HYBRIS – Efficient Specification and Analysis of Hybrid Systems Part II – The Hybrid Low-Level Language Framework HL3

Kirsten Berkenkötter Stefan Bisanz Ulrich Hannemann
Jan Peleska

University of Bremen
P.O.B. 330 440
28334 Bremen, Germany
{kirsten, bisanz, ulrichh, jp}@informatik.uni-bremen.de

Contents

Part II

HL3 – The Hybrid Low-Level Language Framework

Chapter 2

HL3 – Motivation and Concepts

2.1 Some Observations About Existing Formalisms for Hybrid Systems

Though today numerous formalisms and verification approaches are available for hybrid systems (see references in the related-work section below), their application in an industrial "real-world"-context is still rare. According to our analysis, two main causes are responsible for this situation:

- The syntax developed for hybrid formalisms within research communities was too specialized and not supported by conventional software engineering tools available to practitioners.
- While the underlying theories focused on specification semantics and formal verification by theorem proving or model checking, there was insufficient support for the development of optimized code for embedded control systems.

With respect to the first cause we suggest to augment existing well-accepted formalisms of software engineering by new specification constructs describing time-continuous behavior. This has led to the development of the *HybridUML profile* described in Part I of this document.

The second cause is related to both practical and theoretical considerations: From a practitioner's point of view, the effort invested into formal specification and verification – which will certainly be considerably higher than the effort spent on elaborating informal conventional specifications – is only justified if the specifications can be easily transformed into executable systems. In principle, two approaches are available for this task: Stepwise refinement according to the *invent and verify paradigm* relies on software developers to generate a sequence of increasingly concrete refinements of the initial specification and prove that each of these is consistent with the more abstract one according to some refinement relation; the refinement series ends when the concrete code level is reached. The second, so-called *transformational* or *model-driven*, approach aims at "compiling" the specification automatically into executable code, without interactive participation of programmers.

From our experience with industrial projects we do not think that refinement strategies will have a fair chance of becoming the principal development methodology in real-world projects. Though a few notable and highly successful examples exist – such as the Paris Metro control software development performed by Matra using the B-method – there is no indication that these approaches will become widely-spread industrial standard techniques.

In contrast to this, we are convinced that the model-driven development approach allowing to transform semantically well-defined specification models into executable code is highly promising: The effort spent on constructing a complete and consistent specification is rewarded by considerably – if not completely – reducing the programming effort. But also this approach has its pitfalls: If the specification formalism is associated with a *high-level semantics* in the sense that its syntactic constructs are directly associated with denotational or operational mathematical models, the consistency between code and specification still has to be verified; this has the quality of a "1-step refinement approach": The semantics of a general programming language like C, C++, Java or assembler differs from that of the specification formalism, and therefore a proof is necessary that the executable code, interpreted in the appropriate *low-level semantics*, is a correct refinement or "simulation" of the specification model. This task will be rather hard in general and may become infeasible if the code generator has not been built with these verification objectives in mind.

Our suggestion to overcome this problem associated with the transformational approach is based on two main ideas:

- Restrict the infinite variety of possible compilation targets for hybrid specifications according to a *Hybrid Low-Level Language Framework HL3*, so that the compilation target consists of recurring and well-understood design patterns and its semantic interpretation is feasible,
- associate a *transformational semantics* with each high-level specification by the same compilation process used to generated the executable system.

First, the HL3 framework fixes a specific hard real-time runtime environment which avoids uncertainties introduced by using arbitrary operating systems. Second, all specifications written in a given hybrid high-level formalism have to be compiled using a transformation function which generates instances of abstract classes pre-defined by the framework. As a consequence, the variable compilation targets depending on formalism and specification are restricted with respect to software architecture and interfaces to the runtime environment. Therefore the behavioral semantics of the executable target can be given more easily than for an unrestricted compilation into a programming language. If the high-level formalisms have been introduced informally, the transformation defines the semantics as well. If, however, the transformation has only been created in order to translate specifications with given high-level semantics into executable code, the consistency between abstract specification behavior and executable compilation target still has to be verified. Due to the restrictive structure of compilation targets and runtime environment, this proof obligation is at least easier to discharge within the HL3 framework than for arbitrary transformations designed in an intuitive way.

2.2 The HL3 Approach to Model-Based Development

The HL3 framework developed by the authors consists of a re-usable hard real-time runtime environment R and a design pattern P for compilation targets of arbitrary hybrid specifications. Given a high-level formalism H – such as HybridUML – for the description of hybrid systems, transformations Φ_H from high-level specifications S into instances $\Phi_H(S)$ of the HL3 pattern P can be developed. For $(\Phi_H(S), R)$, a formal semantics $\mathcal{S}(\Phi_H(S), R)$ is defined so that the transformation both provides a semantic definition of S and an executable program whose behavior will be consistent with $\mathcal{S}(\Phi_H(S), R)$. Similar to machine code, HL3 should not be used for manual programming, but as a target language for automated transformations. In contrast to machine code, the real-time semantics of HL3 programs can be determined in a direct way, thereby assigning formal meaning to the high-level specification used as the transformation source. This is achieved by using a very limited range of instructions for multi-threading, timing control, and consistent handling of global state in presence of concurrency.

2.3 Related Work

Hybrid systems have been studied extensively in various research communities since the early nineties. The definition and investigation of the Duration Calculus (see [ZRH93, RRS03] and further references given there) provided fundamental contributions to understanding Hybrid Systems. The introduction of Hybrid Automata [Hen96] demonstrated the feasibility of verification by model checking for hybrid specifications. The applicability of hybrid automata to large-scale systems was improved by the introduction of hierarchical hybrid specifications [AGLS01]. Alternative hierarchical approaches closer to the Statecharts formalism have been described in [KMP00] (together with a proof theory) and [BBB$^+$99] (verification by model checking).

We mention GIOTTO [HHK03] as today's most prominent example of a hard real-time language with well-defined semantics. Similar to our HL3 framework, GIOTTO follows the time-triggered systems paradigm described in [Kop97a]. The time-triggered approach is particularly well-suited for real-time programs discretising time-continuous evolutions, since it guarantees bounded timing jitter for periodic schedules. In contrast to this, other approaches to hard real-time focus on the fast response to external interrupts, see [RTA, RTL] for popular real-time variants of the Linux operating system.

HL3 differs from GIOTTO in the following way:

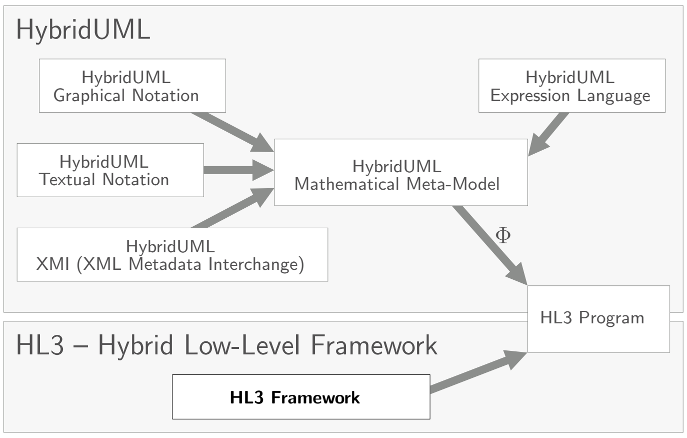
- The framework approach of HL3 offers a "recipe" how to construct transformation for new hybrid high-level formalisms in a systematic routine.

- The communication mechanism of HL3 has been explicitly designed to facilitate the implementation of visibility and atomicity requirements for actions on global observables, as they are often required by high-level formalisms.

- The HL3 runtime environment has been optimized for execution of high-speed multi-CPU cluster architectures.

2.4 Overview of Part II

In the next chapter we define the HL3 Low-Level Framework in a detailed way. A formal operational semantics is given for the execution of HL3 models.


HybridUML
HybridUML Graphical Notation
HybridUML Expression Language
HybridUML Textual Notation
HybridUML Mathematical Meta-Model
HybridUML XMI (XML Metadata Interchange)
Φ
HL3 Program
HL3 – Hybrid Low-Level Framework
HL3 Framework

Chapter 4 defines the starting point for the transformation of HybridUML models to their executable semantics. Instead of using the graphical HybridUML syntax directly, we define a mathematical structure that exactly represents HybridUML models syntactically, but non-graphically. The benefits of this separation of the mathematical model from the graphical representation are that (1) the mathematical model is directly usable for transformation Φ, and that (2) the HybridUML semantics is independent from the graphical notation.

The description of the HybridUML model is completed with the definition of the HybridUML Expression Language (HybEL), defining the syntax of expressions that can be used within HybridUML specifications. Its syntax and intuitive semantics are explained in Chapter 5.

Finally, Chapter 6 provides the specific transformation Φ_{HUML} from HybridUML models to instances of the HL3 design pattern. The transformation is defined formally, therefore the HybridUML executable semantics results. The resulting semantics is the HybridUML simulation semantics – it defines the behavior of a self-contained simulation of the complete HybridUML model.

The transformation results are separated into two parts: (1) Independently from the specific model, HybridUML-specific behavior definitions are given. (2) Corresponding to the specific HybridUML model, the entities of HL3 model, as well as their dependencies, are defined.

Explicit programs are part of the resulting HL3 model, they are defined by HybEL expressions contained in the HybridUML model. The transformation rules for expressions into programs are presented separately.

For the definition of the static structure of the HL3 model, an evaluation semantics of HybEL expressions in the context of given HybridUML models is defined.

Chapter 3

HL3 – Hybrid Low-Level Framework

The *Hybrid Low-Level Framework HL3* is a generic compilation target for hybrid high-level formalisms. It is designed to support the transformation of high-level specifications into executable code, thereby assigning a formal semantics to the generated *HL3 model*. The HL3 model (also called HL3 program) is suitable for hard real-time execution, to be used either for developing embedded applications or for their automated test in hardware-in-the-loop configurations.

In the following, we focus on the *execution* of HL3 models – we define an operational semantics for the execution of HL3 models.

Operational Semantics of HL3. The operational semantics of HL3 models is given by a state transition system (STS). The transition system $sts = (S, s_0, T)$ defines the semantics of a specific HL3 model. It consists of a set S of states, an initial state $s_0 \in S$, and a transition relation $T \subseteq S \times S$.

The utilization of STS instead of labeled transition systems (LTS) emphasizes the fact that HL3 just operates on states. Events – which would occur as labels of an LTS – are considered as a higher level concept, to be "implemented" in HL3 as state changes of dedicated variables.

For the operational semantics of HL3 the state space S is built by a cross product

$$S = CONST \times VAR$$

The sub-vector $c \in CONST$ of states $(c, v) \in S$ represents the HL3 model which results from the transformation of a high-level model into the HL3 framework. It is the same for every state $s \in S$, thus it remains constant under any transition of *sts*. Nevertheless, it is useful to consider c as part of the state space, since application conditions for transitions of the STS may depend on its values.

The dynamic part of the states of *sts* is then given by $v \in VAR$. This is the "conventional" part, encoding control state, valuations, etc. Both $CONST$ and VAR are presented in detail in sections 3.2 and 3.3, respectively.

The details of *sts* are provided in the subsequent sections. They are structured as follows: We start with an informal overview of the HL3 framework in section 3.1, which particularly identifies the entities of HL3 models. Sections 3.2 and 3.3 then formally define the state space S of the operational semantics of HL3 models. The transitions $t \in T$ of the operational semantics, that define the system's behavior, are defined in sections 3.4ff – section 3.4 provides the system's overall behavior, i.e. the scheduling of active entities, and sections 3.5 and 3.6 contain the behavior of active entities themselves.

3.1 HL3 Overview

HL3 provides a re-usable hard real-time processing infrastructure – the *runtime environment* – and a *design pattern* for the formalism- and specification-dependent components to be executed within the runtime environment. A HL3 model then consists of a set of passive objects that constitute the runtime environment, and a set of active objects that implement the design pattern.

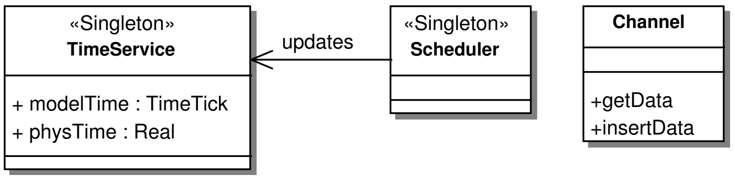


Figure 3.1: Runtime Environment of the HL3 framework.

3.1.1 Runtime Environment

The *HL3 runtime environment* provides pre-defined entities that are available for the instantiation of a HL3 model.

TimeService. At the heart of a real-time system, there is a notion of *time*. The runtime environment provides a *TimeService* which relates the *physical time* to a *model time*. Both are synchronized during the execution of a HL3 model, such that the model execution is based on the model time, which is a discretized view on the physical time.

Physical time is a *global time*, since we assume that HL3 models are executed locally, i.e. on a cluster connected by high-speed local area networks. Therefore, relativistic effects between cluster nodes are neglected. Model time can be obtained by all objects of the model as a pair $t_0.t_1$, with component t_0 representing the discretized physical time. As long as t_0 is kept constant, calculations take place in zero time, from the model's perspective. The second component t_1 is then used to distinguish *causally* related calculations which occur during the same time tick t_0. Consequently, t_1 is always reset when t_0 is incremented.

Channel. In order to model a consistent view on global model data which can be transparently distributed over the (hardware) system, the data structure *Channel* is available. Channels can store several copies of values for different recipients, such that specific values are published at different model times. Typically, on read access, the newest value which is addressed to the respective recipient is obtained. That is, among all data items contained in the channel for which the recipient is within their scope, the most recent entry associated with a visibility time that is less or equal to the current model time, is chosen.

Different requirements on HL3 executions can be satisfied by use of channels: (1) Racing conditions between calculations that are executed in parallel can be avoided, by publishing calculation results in the model future. Since read access always takes place in model presence, there are different values for the same channel, and they do not conflict. (2) Simultaneous calculations from the model view can be executed sequentially, such that the results of the former calculations are published after the last one is finished. (3) For the execution on (cluster) hardware architectures, the distribution of data takes physical time. By choosing an appropriate delay for data publication, a consistent view of data at all (cluster) nodes can be achieved: Every write access to a channel leads to immediate distribution of the data within the whole cluster. As long as the distribution is completed before the data becomes visible, all cluster nodes will have a consistent view on this data.

Further, specific causalities wrt. time, as required by specific high-level formalisms, can be modeled. For example, different publication times can be used for formalisms where changes shall become immediately visible within the local context of an executing entity, but are published later to external ones.

Scheduler. The central instance of the model execution is the *Scheduler*, since it defines the *cooperation* of the active objects from the design pattern. The scheduler defines an execution loop with specific execution phases, it defines the sequence of object executions and their distribution to CPUs (or cluster nodes). For periodic executions, a system period is determined at compile time, such that the execution loop is synchronized with physical time.

Implementations of the HL3 scheduler require a set of reserved CPUs on which the active objects can be executed without interruptions from an underlying operating system.

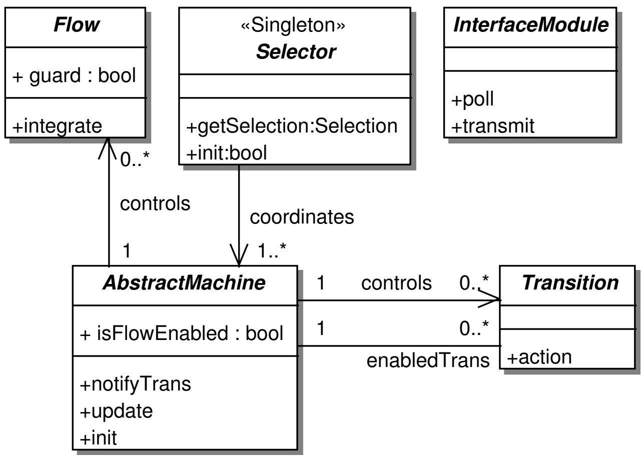


Figure 3.2: Design Pattern of the HL3 framework.

3.1.2 Design Pattern

For hybrid high-level formalisms there are two fundamentally different kinds of behavioral steps which occur within an execution of a model: (1) *Flows* define continuous evolutions of values wrt. time. For example, they can represent algebraic or differential equations from a high-level formalism. (2) Discrete steps represent instantaneous calculations and have no time duration – they are atomic state transformers. Discrete steps are usually called *transitions*. Transitions within HL3 are different from e.g. statechart transitions, in that they represent discrete steps in an *abstract* way. There are neither source or target states for transitions, nor are transitions equipped with signals/events or guard conditions. These are specific concepts of particular high-level formalisms to be implemented in HL3.[1]

The flows and transitions of a high-level formalism are *coordinated* by control behavior, which can be sequential and/or parallel. For the representation of control behavior from the high-level formalism, the HL3 framework provides sequential control components called *abstract machines*. The abstract machines of a HL3 model are always executed concurrently, such that parallel behavior can be modeled, too.

Flows, transitions, and abstract machines are active objects within the HL3 framework. We constrain them such that each flow and each transition is controlled by a specific abstract machine. Further, within a HL3 model a central instance is required which coordinates the set of concurrent abstract machines – the *selector*. Its name is motivated by the fact that its purpose is to make a selection from a set of possible steps, in certain situations. It enforces global behavioral constraints on the concurrent abstract machines, such as the synchronous execution of transitions. Since these constraints do not depend on the concrete high-level specifications, the selector has to be defined only per high-level formalism. Nevertheless, it can be useful to apply different selectors for the same formalism: (1) For application development, a selector will usually resolve nondeterministic transition selection – which may be allowed according to the high-level formalism – to deterministic execution sequences. (2) In contrast to this, a simulation or testing system will require a selector which is capable of producing all transition schedules possible according to the high-level formalism.

For the usage of HL3, it is required that the intended behavior of the applied high-level formalism can be decomposed into these components, i.e. into flows, transitions, abstract machines, and a selector. We expect that this is the case for all reasonable hybrid high-level formalisms.

Additionally, since hybrid specifications are mostly useful when they are connected to a physical environment, a hardware abstraction layer is given which hides driver-specific details and the location of hardware interfaces – so-called *interface modules* encapsulate external interfaces.

The flows, transitions, abstract machines, interface modules, and the selector constitute the entire set of active objects of a HL3 model. Within the HL3 framework their *responsibilities* are defined, but their behavior *is not*. This has to be defined by use of an instantiation rule which depends on the applied high-level formalism. According to this rule, models of the high-level formalism

[1]Further note that HL3 transitions are part of the state space S and are distinct from the transitions $t \in T$ of our HL3 operational semantics.

are then transformed into HL3 models and define specific active HL3 objects. Therefore, the definition of their responsibilities is called *design pattern*.

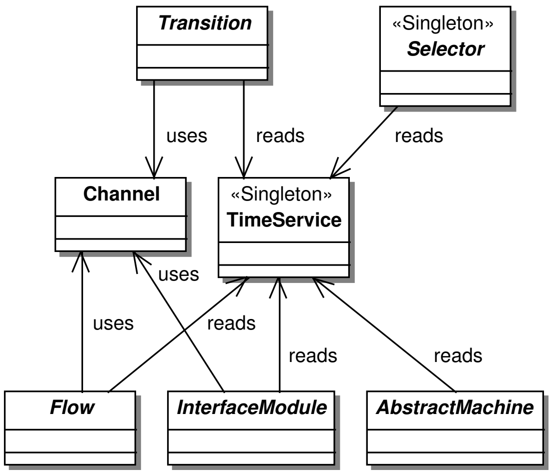


Figure 3.3: Channels and the time service are accessible to the active objects of the design pattern.

The responsibilities of the active objects are:

Flow Since the aim of the HL3 framework is to provide executable models, flows are *discretized*. HL3 flows require integration functions which can be called with regular frequency, such that each flow may be activated with this specific frequency. Its `guard` attribute then determines whether the flow is actually activated or not. The single responsibility of a flow is:

`integrate()` A flow provides an operation `integrate()` which calculates a single discrete step for the approximation of its continuous specification. The calculation's pre-state is retrieved from channels, and the calculation's results (i.e. the post-state) is written (back) to channels, for publication in the future.

Transition The effect of a transition is an inherently discrete calculation:

`action()` The transition's action is the effect of the operation `action()`. It is implemented as a function reading channel data pre-state and setting post state via channels, in the same fashion as flow steps are. In contrast, it is executed exactly once when selected. Further, the publication time depends on the high-level formalism, such that parallel transitions can be modeled, or interleaved ones.

AbstractMachine Abstract machines are more elaborate than flows or transitions. They are assumed to have an *internal* state, which has to be updated at regular intervals. As a result, the *external* state denotes whether flows or transitions are enabled or disabled.

The external state consists of a set of transitions which are enabled, as well as a flag that denotes if the passing of time is admissible wrt. the internal state.

This distinction is made, because flows can only be executed for the complete model, since we assume a *global time*, and flows evolve wrt. time. Nevertheless, the abstract machine may activate or deactivate the set of currently enabled flows, which may change because of its internal state.

The abstract machines' internal state and therefore its behavior depends on the high-level formalism. For example, a concept of partitioning the high-level model state into discrete locations could be encoded within.

The external state is always determined from the internal state, i.e. an abstract machine may indicate whether in the current internal state only transitions, only flows, or one of both may be performed.

We expect that in every conceivable high-level formalism the execution of flows or transitions is mutually exclusive. Otherwise racing conditions might prevent the discrete change of observables due to simultaneous changes by flows. Therefore, the abstract machine's external state can be seen as a constraint for the complete system's execution. An example for high-level formalisms based on the maximal progress concept, or for high-level formalisms allowing the definition of urgent transitions, is the disabling of flows: Whenever an urgent transition is enabled, it is required that the transition has to be executed before time passes. Therefore, the abstract machine prevents the system from taking continuous steps. This implies that (model) time cannot evolve.
The responsibilities of abstract machines are:

init() The abstract machine initializes its internal state.

update() The purpose of this operation is to update the abstract machine's internal state, along with its external state. It is activated for all abstract machines whenever global state encoded in channels has been potentially modified, i.e. after transitions or flows have been performed.
This operation thus determines the current set of enabled transitions and the current value of the flow flag, depending on the internal pre-state as well as the global state.

notifyTrans() The abstract machine updates its internal state, corresponding to the execution of a specific associated transition. For example, if the internal state encodes locations, the execution of a particular transition probably implies a change between locations inside the respective abstract machine, therefore this operation is activated.

InterfaceModule Interface modules are software components which abstract from hardware[2] interfaces.

Interface modules are treated in a similar fashion as flows; from the model perspective, they are interpreted as discretized continuous evolutions of values. They are scheduled with fixed frequency and perform an abstraction from raw data received on hardware interfaces to channel data and vice versa. Since hardware interfaces are sources and/or sinks of data in a natural way, sending and receiving of data is distinguished here:

poll() This reads data from the associated interface and places it into a corresponding channel. Polling of data is associated with publication times,

[2]Although the main purpose of interface modules is the abstraction from *hardware* interfaces, technically also software components can be represented.

such that it is ensured that the data will have been distributed to all recipients before it becomes visible.

`transmit()` This sends data to the associated (hardware) interface. In contrast to `poll()`, each interface module retrieves the current data when scheduled and transmits this data immediately to its hardware interface.

Selector The selector is the single component that coordinates the abstract machines' external states. It selects transitions and/or flows, when it is activated:

`getSelection()` The selector chooses a set of transitions and sets a single flow-enabling flag, on the basis of the external states of the abstract machines. The set of transitions is the result of a selection procedure among all possible transitions offered by the abstract machines in their current state, such that all selected transitions shall be executed subsequently. The selector must therefore ensure, that no conflicting transitions exist within this selection. For high-level formalisms with a notion of nondeterminism, each (sequential) abstract machines may offer more than one transition for selection, and it is the selector's responsibility to make the choice.
Further, the set of transitions can be empty, which means that no discrete step can be taken. Since the possibility of a subsequent flow step is controlled by the flow flag, the selector's choice also defines how to proceed with the execution – by a discrete or flow step.

`init()` The selector may have an internal state, which is initialized with this operation. Additionally, initialization constraints can be checked on the state space; the operation is expected to give a boolean result that denotes if the state space is initially well-formed or not.

The selector is different from the other objects in that it shall not be instantiated for the particular high-level model, but for the high-level *formalism.* Therefore, every model-specific control behavior must be encoded into the abstract machines of the HL3 representation.

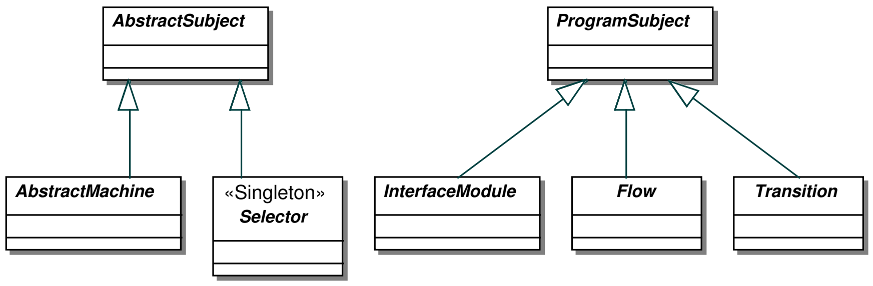


Figure 3.4: Executable objects are grouped into *abstract subjects* and *program subjects.*

The active objects of a HL3 model are also called *subjects.* We distinguish two different kinds of subjects – (1) Abstract machines and the selector can be arbitrarily complex, therefore abstract definitions of their functionality are assumed. They are *abstract subjects*. (2) Flows, transitions, and interface modules are supposed to be less elaborate. Particularly, they have no internal state, such that they can be given by while-programs. These are called *program subjects*.

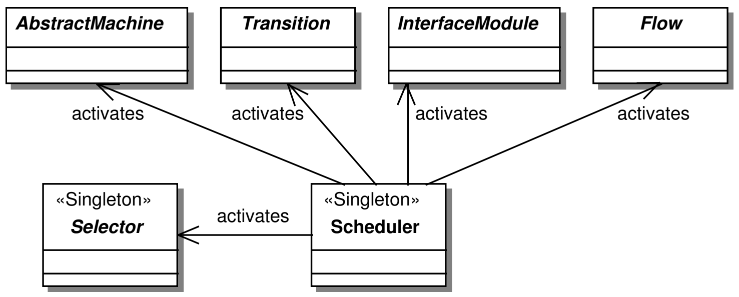


Figure 3.5: The HL3 Scheduler schedules the active objects of the design pattern.

3.2 CONST – Constant State Components

An element $c \in \mathit{CONST}$ of the constant part of the state space S defines one particular HL3 model. It consists of (1) the active and passive entities described in section 3.1, (2) structural aspects of the high-level model, as well as (3) physical constraints specifications.

Formally, the constant state components are given by the set

$$\begin{aligned} \mathit{CONST} \;=\; & \{\mathit{scheduler}\} \times \mathit{SELECTOR} \times \\ & \mathcal{P}(\mathit{AM}) \times \mathcal{P}(\mathit{IFM}) \times \mathcal{P}(\mathit{FLOW}) \times \mathcal{P}(\mathit{TRANS}) \times \\ & \mathcal{P}(\mathit{VAR}) \times \mathcal{P}(\mathit{CHAN}) \times \mathcal{P}(\mathit{PORT}) \times \mathcal{P}(\mathit{LWP}) \times \\ & \mathit{Am}_{\mathit{Trans}} \times \mathit{Am}_{\mathit{Flow}} \times \mathit{Subject}_{\mathit{Var}} \times \\ & \mathit{Chan}_{\mathit{Port}} \times \mathit{InitVal}_{\mathit{Port}} \times \mathit{Subject}_{\mathit{Port}} \times \mathit{SelPort} \times \\ & \mathit{VisibilitySet}_{\mathit{Flow}} \times \mathit{VisibilitySet}_{\mathit{Ifm}} \times \mathit{VisibilitySet}_{\mathit{Trans}} \times \\ & \mathit{Lwp}_{\mathit{Subj}_{abs}} \times \mathit{Program}_{\mathit{Subj}_{prog}} \times \\ & \mathit{SysPeriod} \times \mathit{Period}_{\mathit{Flow}} \times \mathit{Period}_{\mathit{Ifm},\mathit{poll}} \times \mathit{Period}_{\mathit{Ifm},\mathit{tmit}} \end{aligned}$$

The detailed description is given below. For convenience, named projections[3] are given. The following mapping names are available for access to the

[3]Projections are defined as $\pi_i\, X : X_1 \times \ldots \times X_n \rightarrow X_i$ with $(x_1, \ldots, x_n) \mapsto x_i$ for $X = X_1 \times \ldots \times X_n$ and $1 \leq i \leq n$.

coordinates of $c \in CONST$:

$$\begin{aligned}
&sched = \pi_1\, CONST, sel = \pi_2\, CONST, am = \pi_3\, CONST,\\
&ifm = \pi_4\, CONST, flow = \pi_5\, CONST, trans = \pi_6\, CONST,\\
&var = \pi_7\, CONST, chan = \pi_8\, CONST, port = \pi_9\, CONST,\\
&lwp = \pi_{10}\, CONST, am_{trans} = \pi_{11}\, CONST, am_{flow} = \pi_{12}\, CONST,\\
&subject_{var} = \pi_{13}\, CONST, chan_{port} = \pi_{14}\, CONST, initval_{port} = \pi_{15}\, CONST,\\
&subject_{port} = \pi_{16}\, CONST, selport = \pi_{17}\, CONST, vis_{flow} = \pi_{18}\, CONST,\\
&vis_{ifm} = \pi_{19}\, CONST, vis_{trans} = \pi_{20}\, CONST, lwp_{subj} = \pi_{21}\, CONST,\\
&prg_{subj} = \pi_{22}\, CONST, \delta_{period} = \pi_{23}\, CONST, period_{flow} = \pi_{24}\, CONST,\\
&period_{ifm,poll} = \pi_{25}\, CONST, period_{ifm,tmit} = \pi_{26}\, CONST
\end{aligned}$$

Similarly, access to the constant state components c of states $(c, v) \in S$ is given by $const_S = \pi_1\, S$.

Only dedicated parts of the constant state space are accessible from the subjects of the HL3 model, as used in section 3.6. These are

$$CONST_m \quad = \quad Subject_{Var} \times Chan_{Port} \times Subject_{Port}$$

3.2.1 Entities

The entities of a HL3 model are the union of the entities given by the design pattern and runtime environment, as described in section 3.1. Additionally, a set of *light weight processes* is given. This represents the extent of true parallelism that is available for the model's execution.

We define the sets of *possible* entities that can be contained by HL3 models $c \in CONST$. Additionally, we explicitly assume a specific model c that contains specific sets of entities. Therefore, we can tailor the definitions of dependencies and physical constraints in the subsequent sections to the entities of c, and can omit all possible entities that are not part of the specific model.

scheduler identifies the pre-defined HL3 scheduler. There is exactly one scheduler for all particular HL3 models with a fixed behavior, leading to the operational rules given in section 3.4.

SELECTOR is the set of selectors that exist for all possible hybrid systems specification formalisms. The selector $sel(c) \in SELECTOR$ is tailored for the applied high-level formalism, i.e. for the high-level formalism from which the HL3 model $c \in CONST$ originates. We abbreviate it as $selector = sel(c)$.

AM is the set of possible abstract machine identifiers. The set $am(c) \subseteq AM$ is derived from high-level specification, such that the sequential and parallel control aspects from the high-level specification are implemented by these abstract machines. We use $Am = am(c)$ as an abbreviation.

IFM contains interface modules identifiers. They implement (hardware) interfaces according to the high-level specification. The available interface modules of HL3 model $c \in CONST$ are $Ifm = ifm(c)$ with $Ifm \subseteq IFM$.

FLOW are flow identifiers. The flows of $c \in \mathit{CONST}$ which implement the high-level model's time-continuous calculations are $\mathit{Flow} = \mathit{flow}(c)$ with $\mathit{Flow} \subseteq \mathit{FLOW}$.

TRANS are transition identifiers. The transitions that represent the discrete steps of the high-level model are $\mathit{Trans} = \mathit{trans}(c)$ with $\mathit{Trans} \subseteq \mathit{TRANS}$.

VAR is the set of available variable symbols. Each local variable which is used in the HL3 model $c \in \mathit{CONST}$ is contained in $\mathit{Var} = \mathit{var}(c) \subseteq \mathit{VAR}$.

CHAN is the set of channel identifiers. Each global or shared variable from the high-level model $c \in \mathit{CONST}$ is represented by a channel in $\mathit{Chan} = \mathit{chan}(c) \subseteq \mathit{CHAN}$.

PORT provides *port* identifiers for *channel access:* Data is not read or written directly on channels, but through ports. Particularly, subjects that act as data *recipients* are abstracted by the ports they read. Therefore, the addressing of recipients is done in a uniform way. For a HL3 model $c \in \mathit{CONST}$, the available ports are given by $\mathit{Port} = \mathit{port}(c) \subseteq \mathit{PORT}$.

LWP. The HL3 has an execution model which explicitly supports *true parallelism*: An HL3 program is executed on one or more *light weight processes (LWP).* There are specific light weight processes $\mathit{Lwp} = \mathit{lwp}(c) \subseteq \mathit{LWP}$ for models $c \in \mathit{CONST}$. Each LWP runs exclusively on a dedicated CPU, i.e. there is a fixed mapping $\mathit{cpu} : \mathit{Lwp} \rightarrowtail \mathit{Cpu}$. Each LWP executes without any interference of an underlying operating system[4]. Each subject is allocated on one of the available LWPs in order to be executed. If several subjects shall be allocated to the same LWP, only one of them may be active at a time. Active subjects operating on different LWPs are running simultaneously, having simultaneous access to all resources (memory, interfaces etc.). Subjects running on the same LWP only access resources one after another, since only one of them is active at any point in time.

For the subjects, i.e. the active objects of the HL3 model, the distinction into *abstract subjects* and *program subjects* is not encoded explicitly, because it is given implicitly:

$$\begin{aligned}
&\mathit{Subject} = \mathit{Subj}_{abs} \cup \mathit{Subj}_{prog}\\
&\mathit{Subj}_{abs} = \mathit{Am} \cup \{\mathit{selector}\}\\
&\mathit{Subj}_{prog} = \mathit{Flow} \cup \mathit{Trans} \cup \mathit{Ifm}
\end{aligned}$$

3.2.2 Dependencies

For the entities of a HL3 model, static dependencies exist:

[4]An implementation based on the Linux operating system is provided in [Efk05]

$\mathbf{Am_{Trans}}$. Each transition of the model is exclusively controlled by one abstract machine:

$$Am_{Trans} = Trans \longrightarrow Am$$

$\mathbf{Am_{Flow}}$. The same holds for flows – each one is controlled by a dedicated abstract machine:

$$Am_{Flow} = Flow \longrightarrow Am$$

$\mathbf{Subject_{Var}}$. A local variable is only accessible for one subject:

$$Subject_{Var} = Var \longrightarrow Flow \cup Trans \cup Ifm \cup Am$$

$\mathbf{Chan_{Port}}$. A port provides access to exactly one channel. There is no distinction between read and write access, i.e. always read/write access is provided.

$$Chan_{Port} = Port \longrightarrow Chan$$

$\mathbf{InitVal_{Port}}$. For each port, there is an initial data value which is read before any further value is published for this port through the corresponding channel.

$$InitVal_{Port} = Port \longrightarrow Data$$

$\mathbf{Subject_{Port}}$. Each particular port of a channel is assigned to a set of subjects. This defines the subjects which are allowed to access the port.

$$Subject_{Port} = Port \longrightarrow \mathcal{P}(Am \cup Flow \cup Trans \cup Ifm)$$

SelPort. The selector may access ports for special purposes. For example, the resetting of signals from a high-level formalism could be the selector's responsibility.

$$SelPort = \mathcal{P}(Port)$$

$\mathbf{VisibilitySet_{Flow}}$. Each flow has a specific visibility set, which is used to define a maximal set $R \subseteq Port$ of recipients for the calculation results of the flow. When the flow writes data to a channel cn, the actual recipients are the ports $\{p \in Port \mid chan_{port}(c)(p) = cn\} \cap R$.
The time component t of the visibility set acts as a constant publication delay per recipient: Basically, before `integrate()` is executed, the scheduler will choose publication times such that the data will be visible for all recipients as soon as time increases. Then, the scheduler adds the specified delays to these publication times. The resulting visibility set is passed as input parameter for `integrate()`.

$$VisibilitySet_{Flow} = Flow \longrightarrow VisibilitySet$$

$\mathbf{VisibilitySet_{Ifm}}$. The visibility sets which are given for interface modules are similar to the flows' visibility sets, i.e. a recipient set $\{p \in Port \mid chan_{port}(c)(p) = cn\} \cap R$ for the specified recipient set R results, and the given times are regarded as delays.

$$VisibilitySet_{Ifm} = Ifm \longrightarrow VisibilitySet$$

VisibilitySet$_{\mathbf{Trans}}$. The visibility sets associated with transitions are also similar to the flows' and interface modules' visibility sets, but with a single difference: The basic publication times originate from the selector, rather than from the scheduler, and therefore depend on the selection policy of the applied high-level formalism.

$$\mathit{VisibilitySet}_{\mathit{Trans}} = \mathit{Trans} \longrightarrow \mathit{VisibilitySet}$$

Lwp$_{\mathbf{Subj_{abs}}}$. Abstract subjects $s \in \mathit{Subj}_{abs}$ are statically assigned to LWPs:

$$\mathit{Lwp}_{\mathit{Subj}_{abs}} = \mathit{Subj}_{abs} \longrightarrow \mathit{Lwp}$$

This is due to the fact that abstract subjects can have internal state. The HL3 framework does not constrain the implementation of abstract subjects, therefore it is not guaranteed that the internal state is accessible from arbitrary LWPs. In contrast, program subjects do not have internal state and can be scheduled dynamically to any available LWP.

Program$_{\mathbf{Subj_{prog}}}$. Within the HL3 framework, program subjects $s \in \mathit{Subj}_{prog}$ are given in more detail than abstract subjects. Their behavior is explicitly specified by *while-programs*, which are defined wrt. the subject's operations. The available operations are listed in section 3.3.

$$\begin{aligned}\mathit{Program}_{\mathit{Subj}_{prog}} = {} & (\mathit{Flow} \times \mathit{Op}_{\mathit{Flow}}) \cup (\mathit{Trans} \times \mathit{Op}_{\mathit{Trans}}) \cup (\mathit{Ifm} \times \mathit{Op}_{\mathit{Ifm}}) \\ & \longrightarrow \mathit{Program}\end{aligned}$$

The program is given by a program string. The semantics of program strings is discussed in section 3.6.

3.2.3 Physical Constraints

For the representation of a hybrid high-level model in HL3, a *discretization* takes place. This is a necessity, because HL3 models shall be executable on real hardware, which inherently work in a discrete fashion. Therefore, the periods which define the time durations for subsequent calculation steps of continuous evolutions from the model are significant:

SysPeriod. The internal scheduling period $\delta_{period}(c)$ is the smallest time duration which the execution of the HL3 model can observe. Therefore, all continuous calculations, i.e. flow and interface module activations, have to occur at time intervals which are multiples of $\delta_{period}(c)$.
It is a *physical* time duration, i.e.:

$$\mathit{SysPeriod} = \mathit{PhysicalTime}$$

Physical time is defined in section 3.3.

Period$_{\mathbf{Flow}}$. For a flow f, value $\mathit{period}_{\mathit{flow}}(f)$ denotes the multiple of $\delta_{period}(c)$, such that $p_f = \delta_{period}(c) \cdot \mathit{period}_{\mathit{flow}}(f)$ is the flow's scheduling period:

$$\mathit{Period}_{\mathit{Flow}} = \mathit{Flow} \longrightarrow \mathbb{N}$$

$\mathbf{Period_{Ifm,poll}}$. Analogously, function $period_{ifm,poll}$ defines the scheduling period for the polling of data from interface modules:

$$Period_{Ifm,poll} = Ifm \longrightarrow \mathbb{N}$$

$\mathbf{Period_{Ifm,tmit}}$. Function $period_{ifm,tmit}$ defines the scheduling period for the transmitting of data to interface modules:

$$Period_{Ifm,tmit} = Ifm \longrightarrow \mathbb{N}$$

3.3 VAR – Variable State Components

The variable portion VAR of the state space S is structured into the following sub-components:

$$\begin{aligned} VAR \quad = \quad & PhysicalTime \times ModelTime \times FailStatus \times \Sigma_{LWP} \times \\ & Sched \times \Sigma_{Subj_{prog}} \times \Sigma_{Subj_{abs}} \times \Sigma_{Flow} \times \Sigma_{Am} \times \\ & \Sigma_{Var} \times \Sigma_{Chan} \end{aligned}$$

As for $CONST$, named projections are given for VAR:

$$\begin{aligned} & physTime = \pi_1\ VAR, modelTime = \pi_2\ VAR, fail = \pi_3\ VAR, \\ & \kappa_{LWP} = \pi_4\ VAR, sched = \pi_5\ VAR, \kappa_{Subj_{prog}} = \pi_6\ VAR, \\ & \kappa_{Subj_{abs}} = \pi_7\ VAR, \kappa_{Flow} = \pi_8\ VAR, \kappa_{Am} = \pi_9\ VAR, \\ & \sigma_{Var} = \pi_{10}\ VAR, \kappa_{Chan} = \pi_{11}\ VAR \end{aligned}$$

For states $(c, v) \in S$, the projection $var_S = \pi_2\, S$ provides the variable components v.

For use in section 3.6, the portions of the variable state space that can be read or written, respectively, by the subjects of the HL3 model are given:

$$\begin{aligned} VAR_{mread} \quad &= \quad ModelTime \times \Sigma_{Var} \times \Sigma_{Chan} \\ VAR_{mwrite} \quad &= \quad \Sigma_{Var} \times \Sigma_{Chan} \end{aligned}$$

In the following, the components of VAR are defined and explained. Additionally, constraints on the initial state $s_0 = (c, v_0)$ of the state transition system $sts = (S, s_0, T)$ are given, per component.

PhysicalTime – Global Physical Time. Physical time $PhysicalTime$ is modeled by the non-negative real numbers:

$$PhysicalTime = \mathbb{R}_0^+$$

It is *observed* within an execution of a HL3 model, exclusively by the HL3 scheduler. The HL3 subjects cannot evaluate it.

Init State. Physical time is observed relatively to system start: $physTime(v_0) = 0$

ModelTime – Logical HL3 Time. The logical HL3 time is modeled by *time ticks* which are pairs of non-negative integral numbers:

$$ModelTime = \mathbb{N}_0 \times \mathbb{N}_0$$

For convenience, named projections are defined: $t_0 = \pi_1\, ModelTime$, $t_1 = \pi_2\, ModelTime$

Component $t_0(t)$ of $t \in ModelTime$ represents a discretized abstraction of the physical time as visible to the HL3 subjects. It is always ensured that $t_0(modelTime(v)) \leq physTime(v)$, but – depending on the high-level formalism to be encoded in HL3 – $t_0(modelTime(v))$ may be kept constant for some interval of $modelTime(v)$, in order to simulate the execution of transitions in zero time. The second component $t_1(t)$ of the logical HL3 time is used to distinguish causally related events which occur during the same time tick $t_0(t)$. Component $t_1(t)$ is reset when $t_0(t)$ is increased.

Alternatively to $(t_0, t_1) \in ModelTime$, we write $t_0.t_1 \in ModelTime$, because the natural ordering of values $t_0.t_1$ is almost similar to the ordering of real numbers: $a_0.a_1 \leq b_0.b_1$ iff $a_0 < b_0 \vee a_0 = b_0 \wedge a_1 \leq b_1$.
The addition of two time ticks is defined as:

$$\begin{aligned}
+ : &\, ModelTime \times ModelTime \rightarrow ModelTime \\
&(t_0.t_1, 0.u_1) \mapsto t_0.t_1{+}u_1 \\
&(0.t_1, u_0.u_1) \mapsto u_0.t_1{+}u_1 \\
&(t_0.t_1, u_0.u_1) \mapsto t_0{+}u_0.0 \text{ if } t_0 \neq 0 \wedge u_0 \neq 0
\end{aligned}$$

Init State. Model time starts at zero: $modelTime(v_0) = 0.0$

FailStatus – Execution Failure. For all variants of hard real-time applications, it is important to detect violations of timing restrictions. To record that such a violation has happened, we use set

$$FailStatus = \{ok, failed\}$$

An execution starts in state *ok* and performs a transition to *failed* if any violation of timing restrictions occurs. The detailed conditions for failure are modeled in the operational rules in section 3.4.

In addition to timing failure, the model execution may fail within an explicit initialization phase (see description of **Sched** below) to indicate that there is no valid execution for the init state of the HL3 model.

Init State. Initially, every execution is ok: $fail(v_0) = ok$

$\boldsymbol{\Sigma_{LWP}}$ – Light Weight Processes. State component Σ_{LWP} reflects the present activation state of subjects. It is modeled as a mapping from available LWPs to active subjects, such that for each LWP, the currently allocated subject is given. If an LWP $l \in Lwp$ has no subject, then $\Sigma_{LWP}(l) = scheduler$ denotes that the scheduler is in control of l, in order to schedule some inactive subject or to leave l idle.

$$\Sigma_{LWP} = Lwp \longrightarrow Subj_{abs} \cup Subj_{prog} \cup \{scheduler\}$$

Init State. At system start, no subject is running: $\operatorname{ran} \kappa_{LWP}(v_0) = \{scheduler\}$

Sched – The Schedule. When an active subject releases its LWP p, the scheduler will obtain control of p. During its activation time, it will examine the *Sched* state component modeling the current state of the HL3 program schedule.

We assume *symmetric multi-processing* for any implementation of the scheduler, i.e. the logical scheduler is distributed redundantly to all available LWPs. All local schedulers synchronize wrt. the global HL3 program schedule.
The relevant data structure for the operations of the scheduler is

$$\begin{aligned} Sched \;=\; & (\mathcal{P}(Am \times Op_{Am}) \cup \mathcal{P}(Trans \times Op_{Trans}) \\ & \cup \mathcal{P}((Flow \times Op_{Flow}) \cup (Ifm \times Op_{Ifm})) \\ & \cup \{\{(selector, op)\} \mid op \in Op_{selector}\}) \\ & \times sched\_phase \end{aligned}$$

with

$$\begin{aligned} Op_{Am} &= \{init, update, notifyTrans\} \\ Op_{Trans} &= \{action\} \\ Op_{Flow} &= \{integrate\} \\ Op_{Ifm} &= \{poll, transmit\} \\ Op_{selector} &= \{init, getSelection\} \\ Op &= Op_{Am} \cup Op_{Trans} \cup Op_{Flow} \cup Op_{Ifm} \cup Op_{selector} \end{aligned}$$

and

$$\begin{aligned} sched\_phase \;=\; & \{init\_phase, update\_phase, selection\_phase, flow\_phase, \\ & transition\_phase, notify\_phase\} \end{aligned}$$

We have a set $subj_{sched} = \pi_1\, Sched$ which provides pairs (s, op) of subjects and operations, such each subject s has to be scheduled in order to execute its operation op. Further, $phase = \pi_2\, Sched$ indicates which processing phase of the HL3 execution model is active. If $subj_{sched}(sched(v)) = \varnothing$ in a state $(c, v) \in S$, then there is nothing more to do for the current processing phase, and a switch to the next phase is to follow.

Init State. A system execution starts with initialization of abstract machines and the selector: $sched(v_0) = ((Am \cup \{selector\}) \times \{init\}, init\_phase)$

$\Sigma_{\mathbf{Subj_{prog}}}$ – The Execution State of Program Subjects. The execution state models for each operation of a *program subject* the remaining statements to be executed. Following the standard concepts for explaining operational semantics of sequential programs [AO97], the execution state of one sequential unit is modeled as the string $prg \in Program$ of programming statements $s \in Stmt$ still to be performed:

$$Program = \operatorname{seq} Stmt$$

Since we are modeling executions on real hardware, all atomic statements will complete in finite time. There is an interval $[\delta_0, \delta_1]$ for each statement with

$\delta_i \in \mathbb{R}_+$, such that $\delta \in [\delta_0, \delta_1]$ whenever the associated subject is scheduled and the statement is not yet completed. Therefore the time value $\delta \in \mathbb{R}_0^+ \cup \{\infty\}$ is attached to the string of programming statements, denoting the remaining execution time needed to process the actual atomic statement. However, while the object is not running, this time value is set to ∞, until the object is allocated on an LWP.

Finally, a parameter $p \in \mathit{Param}_{prog}$ may be given for the current execution of the program. The only possible parameter for operations of program subjects is a visibility set, which is applied to the operations *action*, *integrate*, and *poll*, on transitions, flows, and interface modules, respectively. The absence of parameters is denoted by λ.

$$\mathit{Param}_{prog} = \mathit{VisibilitySet} \cup \{\lambda\}$$

Then, the execution state of program subjects is defined as

$$\mathit{ProgState} = \mathit{Program} \times (\mathbb{R}_0^+ \cup \{\infty\}) \times \mathit{Param}_{prog}$$
$$\Sigma_{\mathit{Subj}_{prog}} = \mathit{Subj}_{prog} \times \mathit{Op} \longrightarrow \mathit{ProgState}$$

For convenience, the coordinates of *ProgState* are given by $\mathit{string}_{\mathit{ProgState}} = \pi_1\, \mathit{ProgState}$, $\delta_{\mathit{ProgState}} = \pi_2\, \mathit{ProgState}$, and $\mathit{vis}_{\mathit{ProgState}} = \pi_3\, \mathit{ProgState}$.

Init State. At the beginning of a model run, no program subject is running: $\operatorname{ran} \kappa_{\mathit{Subj}_{prog}}(v_0) = \{(\langle\rangle, \infty, \lambda)\}$

$\Sigma_{\mathbf{Subj_{abs}}}$ – The Execution State of Abstract Subjects. The execution state of an *abstract subject* consists of an internal state component $s \in \mathit{IntState}$, as well as a time value $\delta \in \mathbb{R}_0^+ \cup \{\infty\}$ denoting the remaining execution time needed to process its current task, similarly to the execution of program subjects. The value ∞ denotes that the object is not allocated on an LWP.

Possible execution parameters $p \in \mathit{Param}_{abs}$ are a single transition each, which is needed for the *notifyTrans* operations on abstract machines:

$$\mathit{Param}_{abs} = \mathit{Trans} \cup \{\lambda\}$$

The execution state of abstract subjects is given by

$$\mathit{AbsState} = \mathit{IntState} \times (\mathbb{R}_0^+ \cup \{\infty\}) \times \mathit{Param}_{abs}$$
$$\Sigma_{\mathit{Subj}_{abs}} = \mathit{Subj}_{abs} \longrightarrow \mathit{AbsState}$$

For convenience, the coordinates of *AbsState* are given by $\mathit{intState}_{\mathit{AbsState}} = \pi_1\, \mathit{AbsState}$, $\delta_{\mathit{AbsState}} = \pi_2\, \mathit{AbsState}$, and $\mathit{trans}_{\mathit{AbsState}} = \pi_3\, \mathit{AbsState}$.

Init State. At the beginning of a model execution, no abstract subject is running: $\operatorname{ran} \kappa_{\mathit{Subj}_{prog}}(v_0) \subseteq \{(s, \infty, \lambda) \in \mathit{AbsState}\}$. The internal states s are undetermined, but will be initialized during the *init_phase*.

$\Sigma_{\mathbf{Flow}}$ – The Flow State. The flow state specifies for each flow whether it is currently enabled, and, if this is the case, the absolute point in physical time for its next periodic execution.

$$\mathit{FlowState} = \mathbb{R}_0^+ \cup \{\infty\}$$
$$\Sigma_{\mathit{Flow}} = \mathit{Flow} \longrightarrow \mathit{FlowState}$$

For $(c, v) \in \mathit{VAR}$ and $f \in \mathit{Flow}$, $\kappa_{\mathit{Flow}}(v)(f) = \infty$ indicates that the flow is currently disabled and shall not be scheduled. A value $\kappa_{\mathit{Flow}}(v)(f) \in \mathbb{R}_0^+$ indicates that f shall be scheduled at some time $t \in \mathit{PhysicalTime}$ with $\kappa_{\mathit{Flow}}(v)(f) \leq t < \kappa_{\mathit{Flow}}(v)(f) + \delta_{\mathit{period}}(c)$. If the scheduler cannot manage to reserve an LWP for f for such a time, a transition to (c, v') with $\mathit{fail}(v') = \mathit{failed}$ is performed. A transition into the failure state is also performed if the flow is scheduled correctly at time t, but terminates later than $\kappa_{\mathit{Flow}}(v)(f) + \delta_{\mathit{period}}(c)$. That means, that the flow must be also completed within the specified system period.

Init State. In the beginning, all flows are disabled: $\operatorname{ran} \kappa_{\mathit{Flow}}(v_0) = \{\infty\}$

$\Sigma_{\mathbf{Am}}$ – The Abstract Machine State. The abstract machine status indicates its enabledness for continuous or discrete steps:

$$\begin{aligned} \mathit{AmState} &= \mathbb{B} \times \mathcal{P}(\mathit{Trans}) \\ \Sigma_{\mathit{Am}} &= \mathit{Am} \longrightarrow \mathit{AmState} \end{aligned}$$

with named projections $\mathit{flow}_{\mathit{AmState}} = \pi_1\, \mathit{AmState}$ and $\mathit{trans}_{\mathit{AmState}} = \pi_2\, \mathit{AmState}$.

Continuous Step. The abstract machine $m \in \mathit{Am}$ indicates, whether it admits a continuous step of the *complete system*. This decision is made internally and depends on the applied high-level formalism.

Since time passes for all abstract machines Am of the system, a continuous step can only be taken by all abstract machines together. Therefore, each abstract machine indicates its enabledness for this by a boolean flag, such that the conjunction of all flags defines whether a continuous step is possible for the complete system.

Whether the system actually switches to a flow phase is the choice of the scheduler, based on the selector's selection. Which set of flows is executed in such a case is defined by the flow entities themselves.

Discrete Step. The abstract machines' status also contains, whether a discrete step is possible. Unlike continuous steps, discrete steps may be executed for a subset $\mathit{Am}_{\mathit{dstep}} \subseteq \mathit{Am}$ of all abstract machines. A discrete step consists of the execution of a set of transitions.

Transitions t are either enabled or disabled, depending on the current state $(c, v) \in S$. This is chosen internally within the abstract machine $m = \mathit{am}_{\mathit{trans}}(c)(t)$, i.e. the abstract machine that is responsible for the particular transition.

For a composition of a discrete step, each abstract machine indicates which of its transitions are available, i.e. it provides the set of its enabled transitions.

The selector determines which enabled transitions are to be fired within a discrete step. Since abstract machines are designed as sequential components, every admissible selector will select at most one transition per abstract machine. The scheduler will then allocate the actions associated with all selected transitions on an LWP and notify the corresponding abstract machines about the transitions which have been fired.

The abstract interface postulated for an abstract machine m requires that a call to its `update()` method will calculate the set of enabled transitions (as well as the boolean flag for continuous step-enabledness) again. Observe that abstract machines may label more than one transition as enabled if the underlying high-level formalism admits nondeterministic behavior. Moreover, if the formalism contains the concept of urgent transitions, this can be reflected by a non-empty set of enabled transitions in combination with the *disabling* of continuous steps. This causes the selector to select a discrete step, rather than a flow phase.

Init State. Abstract machines do not select anything, initially: $\operatorname{ran} \kappa_{Am}(v_0) = \{(\mathit{false}, \varnothing)\}$

$\mathbf{\Sigma_{Var}}$ – Local Variable Valuations. Valuations of local variables are specified in the conventional way which is also applied when defining operational semantics to sequential programs: A valuation $\sigma : \mathit{Var} \longrightarrow \mathit{Val}$ maps each local variable symbol $x \in \mathit{Var}$ to its current value $\sigma(x) \in \mathit{type}_{\mathit{Var}}(x)$, i.e. the variables are typed, and Val is the union of all these types:

$$\Sigma_{\mathit{Var}} = \mathit{Var} \rightarrow \mathit{Val}$$
$$\mathit{type}_{\mathit{Var}} : \mathit{Var} \rightarrow \mathcal{P}(\mathit{Val})$$

Init State. There is no initial valuation for local variables. Therefore, every program that uses local variables must initialize them explicitly.

$\mathbf{\Sigma_{Chan}}$ – HL3 Channels. The HL3 framework supports a notion of visibility and publication of variable values, based on the concept of HL3 channels. This is modeled as the current state Σ_{Chan} of channels.

$$\mathit{ChanState} = \mathit{ModelTime} \times \mathit{Port} \nrightarrow \mathit{Data}$$
$$\Sigma_{\mathit{Chan}} = \mathit{Chan} \longrightarrow \mathit{ChanState}$$

When inserting a data item into a channel, a *visibility set* is specified for this item. A visibility set is a collection of visibilities, that are time ticks with an associated recipient:

$$\mathit{Visibility} = \mathit{ModelTime} \times \mathit{Port}$$
$$\mathit{VisibilitySet} = \mathcal{P}(\mathit{Visibility})$$

For convenience, we define an operation $\Cap$ that builds the intersection of two visibility sets wrt. the recipients, and adds the corresponding time ticks:

$$\begin{aligned} \Cap : {} & \mathit{VisibilitySet} \times \mathit{VisibilitySet} \rightarrow \mathit{VisibilitySet} \\ & (v_1, v_2) \mapsto \{(t,p) \in \mathit{ModelTime} \times \mathit{Port} \\ & \qquad \mid \exists (t_1, p) \in v_1, (t_2, p) \in v_2 \bullet t = t_1 + t_2\} \end{aligned}$$

When data is retrieved from a channel through a port $p \in \mathit{Port}$, access to either the most recent data item visible for p, or to the second most recent item is provided. Thus, visibility sets $v \in \mathit{VisibilitySet}$ are used to specify *when* data values should become visible to specific recipients.

The types of application-specific data are transparent to the HL3 framework, since the data is never interpreted or changed by operational rules of the framework. Therefore application-specific data are modeled just as sequences of bytes:

$$\mathit{Data} = \mathrm{seq}\ \mathit{Bytes}$$

We assume that there is a unique byte representation for every application-specific data value:

$$\mathit{data}_{\mathit{Val}} : \mathit{Val} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathit{Data}$$
$$\mathit{val}_{\mathit{Data}} = \mathit{data}_{\mathit{Val}}^{-1}$$

New Entry Insertion. A new data value is inserted into a channel through a port, such that for each receiving port from the visibility set, an entry is created with the according visibility time:

$$\begin{aligned}
&\mathit{ChanEntry} = \mathit{Data} \times \mathit{VisibilitySet}\\
&\mathit{insert}_{\mathit{Port}} : \mathit{Port} \times \mathit{ChanEntry} \times \mathit{CONST}_m \times \mathit{VAR}_{\mathit{mread}}\\
&\qquad \rightarrow (\mathit{Chan} \times \mathit{ChanState})\\
&(p_{\mathit{write}}, (\mathit{data}, \mathit{vis}), c, v) \mapsto\\
&\qquad (\mathit{chan}_{\mathit{port}}(c)(p_{\mathit{write}}),\\
&\qquad \kappa_{\mathit{Chan}}(v)(\mathit{chan}_{\mathit{port}}(c)(p_{\mathit{write}})) \oplus\\
&\qquad\quad \{\, (\mathit{tick}, p_{\mathit{read}}) \mapsto \mathit{data} \mid (\mathit{tick}, p_{\mathit{read}}) \in \mathit{vis}\,\})
\end{aligned}$$

Only one data value is stored for each combination (t, p) of visibility time and recipient port. Therefore, racing conditions take effect on writing into the channel, if data is written concurrently for (t, p). This is the desired behavior for the HL3 semantics. It is the responsibility of the applied high-level formalism to avoid this, if it is required.

Current Entry Access. Access to the currently visible entry in a channel is defined by:[5]

$$\begin{aligned}
&\mathit{entry}_{\mathit{Port},\mathit{cur}} : \mathit{Port} \times \mathit{CONST}_m \times \mathit{VAR}_{\mathit{mread}} \nrightarrow \mathit{ModelTime} \times \mathit{Data}\\
&(p, c, v) \mapsto (t_{\mathit{cur}}, \kappa_{\mathit{Chan}}(v)(\mathit{chan}_{\mathit{port}}(c)(p))(t_{\mathit{cur}}, p))\\
&\ \text{with } t_{\mathit{cur}} =\\
&\qquad \mu(t : \mathit{ModelTime} \mid t \leq \mathit{modelTime}(v)\\
&\qquad\quad \wedge (t, p) \in \mathrm{dom}\ \kappa_{\mathit{Chan}}(v)(\mathit{chan}_{\mathit{port}}(c)(p))\\
&\qquad\quad \wedge (\forall\, u : \mathit{ModelTime} \bullet (u, p) \in \mathrm{dom}\ \kappa_{\mathit{Chan}}(v)(\mathit{chan}_{\mathit{port}}(c)(p))\\
&\qquad\qquad \Rightarrow u \leq t \vee \mathit{modelTime}(v) < u))
\end{aligned}$$

[5] We use the notation $q = \mu(x : X \mid p)$ for a set X, a bound variable x and a predicate p over x as a shorthand for $|\{x \in X \mid p\}| = 1 \Rightarrow q \in \{x \in X \mid p\}$. Further, $q = \mu(X)$ is an abbreviation for $q = \mu(x : X \mid \mathit{true})$.

The above mapping retrieves a pair $(t_{cur}, data) \in ModelTime \times Data$ from the channel $chn \in Chan$ that is associated with the given port $p \in Port$. The system's state $(c, v) \in S$ determines the current valuation $\kappa_{Chan}(v)(chn)$ of the channel, as well as the (constant) dependency $chan_{port}(c)(p)$ between port and channel. $(t_{cur}, data)$ is chosen such that t_{cur} is the dedicated time stamp that (1) is not in the (model) future, that (2) actually appears in the channel for the port, and that (3) is the newest of such time stamps.

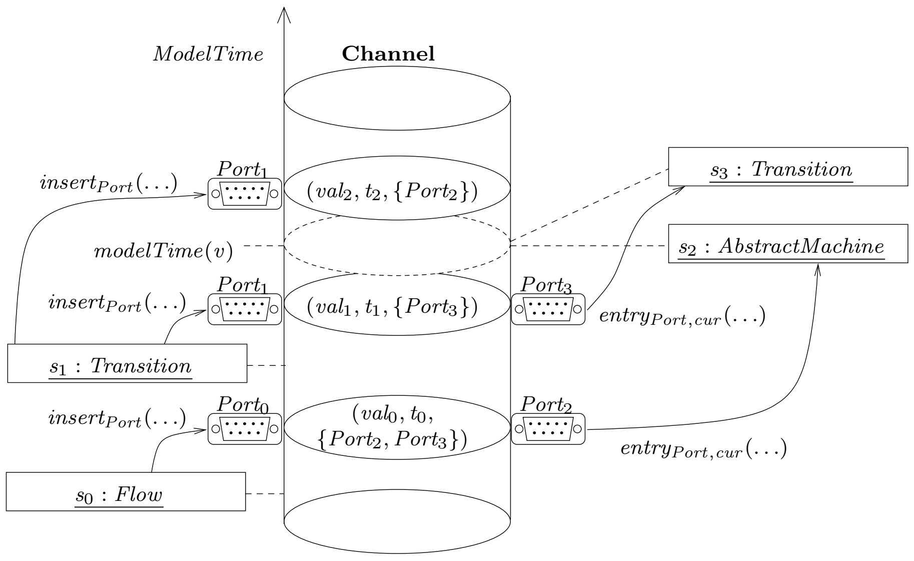


Figure 3.6: Illustration of a HL3 Channel – subjects s_0, s_1 have written values with different visibility sets to the channel, therefore in state $(c, v) \in S$, subjects s_2, s_3 receive different values.

Access to the currently visible data and visibility time is then given by

$$\begin{aligned}
&data_{Port,cur} : Port \times CONST_m \times VAR_{mread} \nrightarrow Data\\
&\quad (p, c, v) \mapsto \pi_2\,(ModelTime \times Data)(entry_{Port,cur}(p, c, v))\\
&tick_{Port,cur} : Port \times CONST_m \times VAR_{mread} \nrightarrow ModelTime\\
&\quad (p, c, v) \mapsto \pi_1\,(ModelTime \times Data)(entry_{Port,cur}(p, c, v))
\end{aligned}$$

Previous Entry Access. In addition to the access to the currently visible entry, channels permit reading of the *previous* entry:

$$\begin{aligned}
&entry_{Port,prev} : Port \times CONST_m \times VAR_{mread} \nrightarrow ModelTime \times Data\\
&(p, c, v) \mapsto (t_{prev}, \kappa_{Chan}(v)(chan_{port}(c)(p))(t_{prev}, p))\\
&\text{ with } T_{prev} = \{t \in ModelTime \mid\\
&\quad \exists\, t_{cur} \in ModelTime \bullet t < t_{cur} \leq modelTime(v)\\
&\qquad \wedge \{(t, p), (t_{cur}, p)\} \subseteq \operatorname{dom} \kappa_{Chan}(v)(chan_{port}(c)(p))\\
&\qquad \wedge (\forall\, u : ModelTime \bullet (u, p) \in \operatorname{dom} \kappa_{Chan}(v)(chan_{port}(c)(p))\\
&\qquad\quad \Rightarrow u \leq t \vee modelTime(v) < u)\}\\
&\quad \wedge T_{prev} \neq \varnothing \Rightarrow t_{prev} = \mu(T_{prev})
\end{aligned}$$

$$\wedge T_{prev} = \emptyset \Rightarrow t_{prev} = tick_{Port,cur}(p, c, v)$$

This mapping provides an entry $(t_{prev}, data)$ such that t_{prev} is the dedicated time stamp from the model presence or past that is the *second newest* of all time stamp occurrences for the given port. Alternatively, iff there is only t_{cur} available (as defined above), this is chosen.
Similarly to $entry_{Port,cur}$, the projections to data and time components are given:

$$\begin{aligned}
&data_{Port,prev} : Port \times CONST_m \times VAR_{mread} \nrightarrow Data\\
&\qquad (p, c, v) \mapsto \pi_2\,(ModelTime \times Data)(entry_{Port,prev}(p, c, v))\\
&tick_{Port,prev} : Port \times CONST_m \times VAR_{mread} \nrightarrow ModelTime\\
&\qquad (p, c, v) \mapsto \pi_1\,(ModelTime \times Data)(entry_{Port,prev}(p, c, v))
\end{aligned}$$

Init State. At system start, every channel contains exactly the initial values for its associated ports, with time stamp zero: $\kappa_{Chan}(v_0) = \{c \mapsto \{(0.0, p) \mapsto initval_{port}(c)(p) \mid p \mapsto c \in chan_{port}\} \mid c \in Chan\}$

3.4 Scheduling Rules

In this section, a high-level view on the system execution is given. First, it is described informally, how the assignment of subjects to LWPs evolves wrt. time, in order to perform their respective tasks – i.e. the *scheduling* is given. This is structured into *scheduling phases* (also called *execution phases*), which are roughly divided into discrete and flow phase, as well as additional housekeeping.

Second, operational rules define the scheduling formally. This includes the switching between complete phases as well as the scheduling of single subjects within phases. The rules that determine the subjects' internal operation are provided in the subsequent sections.

Scheduling Intuition. There are two different aspects of scheduling for a HL3 system: (1) Conceptually, the system resides within an execution loop, that sequentially steps through scheduling phases. Within each iteration, exactly one of *flow phase* or *discrete phase* is chosen. (2) With respect to *physical time*, there is a fixed period $\delta_{period}(c)$ (for all states $(c, v) \in S$) that defines the points in time for which the (discretized) flow calculations have to be started. This restricts and *requires* when flow phases must occur. As long as the system's choice for flow phases within the execution loop is synchronized with the period $\delta_{period}(c)$, i.e. the choice for a flow phase is always in time, the system's timing is ok – $fail(v) = ok$. Otherwise, a timing failure is detected, i.e. $fail(v) = failed$.

Execution Loop. An iteration of the system's execution loop starts (1) in phase *update_phase*, followed by (2) *selection_phase*. Based on the resulting selection, (3) a choice between *flow_phase* and *transition_phase* is made. (4) A *notify_phase* is appended, iff previously a *transition_phase* was chosen, otherwise skipped:

update_phase During this phase, the decision for the next flow or discrete step is made. Since the control logic of the system is coded into the set of available abstract machines $am(c)$, all of them update their *internal state* and provide their resulting choice for discrete and flow steps (alias *abstract machine state*) $s \in AmState$.

selection_phase According to the high-level formalism, a resulting set of transitions and a flag denoting flow-enabledness is determined from the abstract machine's states. From this, either *flow_phase* or *transition_phase* is chosen for the next phase.

flow_phase At the beginning of this phase, the synchronization with the physical time is done – the system *waits* for the expiration of the current system period. Afterwards, all flows $f \in flow(c)$ which are (1) enabled and (2) for which the current time fits to their respective period are recalculated. Interface modules $md \in ifm(c)$ are treated similarly, they are triggered to put/get data to/from their respective interface.

transition_phase This phase executes all discrete actions that were given by the transitions selected in phase *selection_phase*.

notify_phase Following phase *transition_phase*, during this phase each abstract machine for which a transition was selected, is notified. This is necessary, because the firing of a transition $t \in trans(c)$ can have an impact on the internal state of the associated abstract machine $m \in am_{trans}(v)(t)$. Each notified abstract machine then adjusts its internal state correspondingly.

Before the first iteration of the execution loop, i.e. at the start of the model execution, an *init_phase* is executed exactly once:

init_phase Within this phase, the subjects which contain internal state are initialized, such that they have a defined internal state afterwards. The affected subjects are the abstract machines and the selector.

Illustrated Scheduling Example. An example run of a HL3 system for $|lwp(c)| = 3$, i.e. for a system which has three CPUs for parallel execution of subjects, is given in figures 3.7, 3.8, and 3.15. Each figure shows one of three subsequent periods, showing the evolution of physical time from top to bottom. The vertical extent of the boxes denoting the execution of the subjects corresponds to the execution duration of the respective subject. The vertical space between boxes is just for clearness of the presentation, it shall not denote time durations. The horizontal position assigns each execution to one of the available LWPs, the horizontal extent has no meaning. Different phases are separated by dashed lines.

In figure 3.7, a successful execution of two consecutive flow phases as well as of two transition phases in between is shown: (1) The first flow phase shows the scheduling of flows and interface modules. Each flow or interface module is immediately executed once; whenever an idle LWP is found, one of the remaining subjects is assigned and started. (2) Next, a new iteration of the execution loop starts with the update phase. All abstract machines m are scheduled for update, each one on the statically assigned LWP $p = lwp_{subj}(c)(m)$. (3) Then, the selector selects the succeeding phase and determines the subjects to be scheduled within. Since this is a centralized activity, only one LWP is active, the others are idle. (4) The selector has decided to continue with a transition

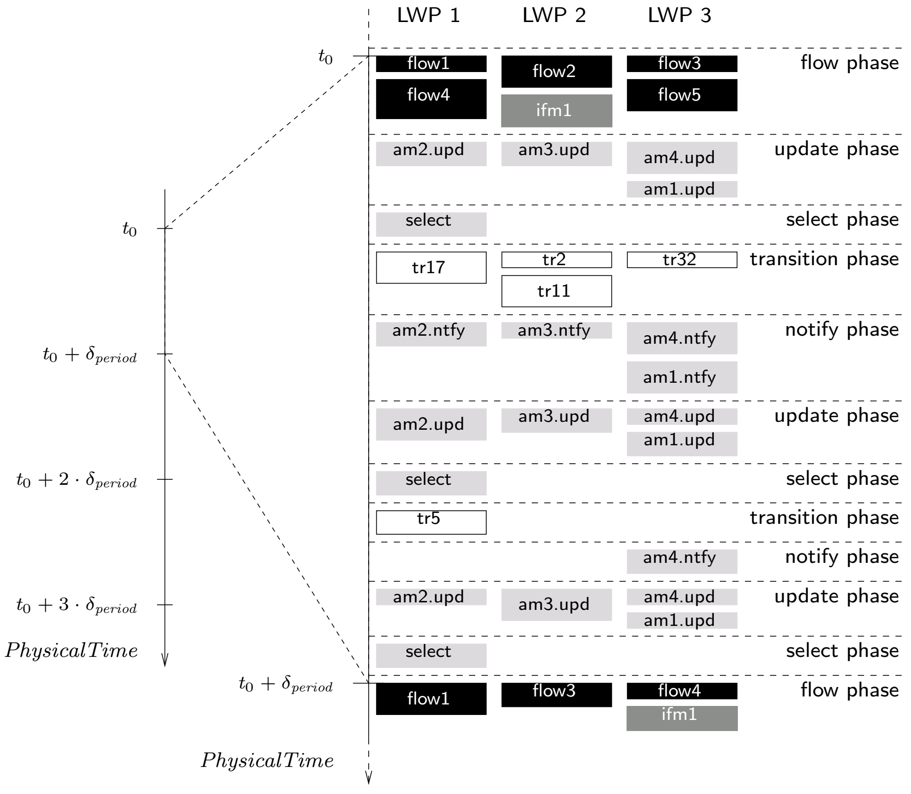


Figure 3.7: Scheduling example: between two consecutive flow phases, two transition phases (along with notification phases) are scheduled.

phase, therefore all actions of the transitions of the selection are executed. Like for flows, each transition is assigned to the first idle LWP found. (5) Afterwards, the notification of abstract machines is done. Again, the assignment of abstract machines and LWPs is fixed. Here, all abstract machines are affected, because for each one, a transition was taken.

Further, a second execution loop iteration with transition phase is shown. Since only one transition is chosen, there is only one abstract machine to notify. The third iteration chooses a flow phase, therefore the timing wrt. the period is ok. Note that this choice is determined from the abstract machines and the selector, i.e. from the model, and cannot be enforced by the HL3 environment.

The successive period is shown in figure 3.8. Here, the first execution loop iteration chooses a flow phase. Therefore, the complete system is idle for the rest of this period, since flows take place in the beginning of the following period.

Finally, in figure 3.15 a scenario is given that violates the timing: the third iteration of the execution loop chooses a transition phase again, which is yet active when the point in time is reached at which the next flow phase should have been scheduled.

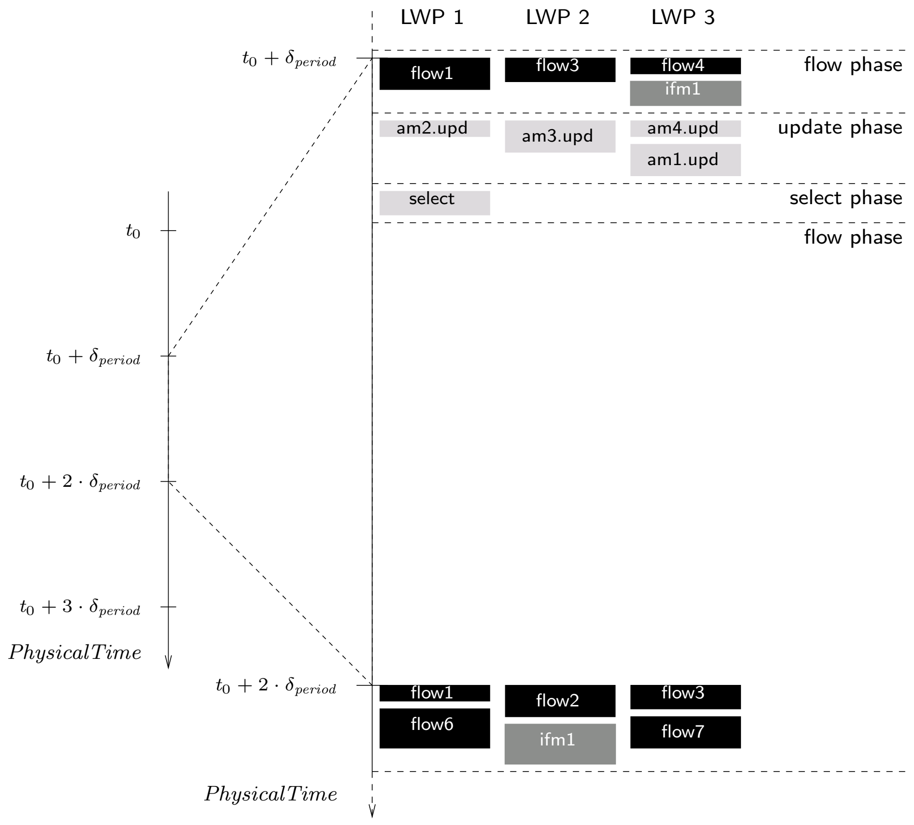


Figure 3.8: Scheduling example: two consecutive flow phases, but no transition phases in between.

3.4.1 Timing

Since the HL3 framework is designed for hard real-time systems, timing is a key issue for the execution of an HL3 model. For arbitrary HL3 models, it cannot be guaranteed that every run meets its timing requirements, because it depends on the transformation from the high-level formalism on the one hand, and on the particular high-level model on the other. Therefore, it is important that during execution, violations of timing requirements are detected.

The purpose of this section is to *identify* the timing constraints that must hold for states $s \in S$ for a successful HL3 model execution. Correspondingly, the state space S is partitioned into *valid states* and *fail states*.

Discretization of Time

For the execution of a HL3 model $c \in \mathit{CONST}$, physical time is *discretized*. In the states of the operational semantics $(c, v) \in S$, the physical time itself is contained as $\mathit{physTime}(v) \in \mathit{PhysicalTime}$, whereas the model time $\mathit{modelTime}(v) \in \mathit{ModelTime}$ is the discretized notion of time. The first component $t_0(t)$ of $t \in \mathit{ModelTime}$ corresponds to physical time in that the operational semantics will set $t_0(\mathit{modelTime}(v)) = \lfloor \mathit{physTime}(v) \rfloor$ at dedicated synchronization points in physical time. The second model time component $t_1(t)$ of $t \in \mathit{ModelTime}$ is not relevant for the considerations of this section and therefore not discussed here.

Along with the discretized evolution of time, continuous calculations represented by flow and interface module operations $(\mathit{Flow} \times \mathit{Op}_{\mathit{Flow}}) \cup (\mathit{Ifm} \times \mathit{Op}_{\mathit{Ifm}})$ have to be computed discretely. An ideal discretization would synchronize the model time and would calculate operations in zero time, exactly for multiples of the system period $\delta_{period}(c)$. Of course, a real discretized execution needs some time duration for calculations, and fortunately, there is time – between each two consecutive multiples $n \cdot \delta_{period}(c)$ and $(n+1) \cdot \delta_{period}(c)$ of the system period, the time duration $\delta_{period}(c)$ can be spent for this.

Physical Time Frame. Therefore, physical time is partitioned into fixed-sized *frames* of duration $\delta_{period}(c)$, beginning with $t = 0$ for system start. Each frame $\mathit{frame}(c, n) = (t_{begin}, t_{end})$ defines a physical time interval $[t_{begin}, t_{end}[$, and frames are numbered by $n \in \mathbb{N}_0$:

$$\begin{aligned} &\mathit{frame} : \mathit{CONST} \times \mathbb{N}_0 \rightarrow \mathit{PhysicalTime} \times \mathit{PhysicalTime} \\ &(c, n) \mapsto (n \cdot \delta_{period}(c), (n+1) \cdot \delta_{period}(c)) \end{aligned}$$

For convenience, we define $\mathit{low} = \pi_1\,(\mathit{PhysicalTime} \times \mathit{PhysicalTime})$ and $\mathit{up} = \pi_2\,(\mathit{PhysicalTime} \times \mathit{PhysicalTime})$ to access the lower and upper bound of the intervals, respectively.

Then, for each time value t, the frame number n for which $t \in [\mathit{low}(\mathit{frame}(c, n)), \mathit{up}(\mathit{frame}(c, n))[$ holds, is given by

$$\begin{aligned} &\mathit{fno}_{time} : \mathit{CONST} \times \mathit{PhysicalTime} \rightarrow \mathbb{N}_0 \\ &(c, t) \mapsto \left\lfloor \frac{t}{\delta_{period}(c)} \right\rfloor \end{aligned}$$

Flow Execution Interval. The major uncertainty for the execution of the HL3 model is the amount of time needed for the calculations within (and between) discretized steps. This cannot be predetermined in general, because abstract machines control dynamically the enabledness of calculations, in a way that depends on the specific high-level model and its specific transformation into HL3.

For every frame, a set of flow operations and interface module operations have to be executed. A superset is predefined by the period factors $\mathit{period}_{\mathit{flow}}(c)$, $\mathit{period}_{\mathit{ifm},\mathit{poll}}(c)$, and $\mathit{period}_{\mathit{ifm},\mathit{tmit}}(c)$, but flows can be dynamically disabled by abstract machines. Depending on the number of available LWPs, on the execution times of the calculations, and of their exact activation times, a *flow execution interval* results for each frame number, such that all corresponding operations are executed within:

$$\begin{aligned} &\mathit{fexecint} : \mathit{CONST} \times \mathbb{N}_0 \rightarrow \mathit{PhysicalTime} \times \mathit{PhysicalTime} \\ &\forall (c, n) \in \mathit{CONST} \times \mathbb{N}_0 \bullet \mathit{low}(\mathit{fexecint}(c, n)) \geq \mathit{low}(\mathit{frame}(c, n)) \end{aligned}$$

Note that the flow execution interval also does not end before its frame:

$$\forall (c, n) \in \mathit{CONST} \times \mathbb{N}_0 \bullet \mathit{up}(\mathit{fexecint}(c, n)) \geq \mathit{low}(\mathit{frame}(c, n))$$

The operational semantics will not execute the operations of a flow execution interval *before* the respective frame has begun: The flow execution interval is embedded into the scheduling phase *flow_phase*, such that their end times coincide. However, the *flow_phase* generally begins earlier than the flow execution interval, such that the system execution *waits* for its beginning.

Model Time Synchronization. Directly after the executions of the operations within the flow execution interval are finished, the operational semantics will synchronize the model time wrt. the physical time. The synchronization points in physical time as they are observed for a particular HL3 model execution, are given for the corresponding frame number:

$$\begin{aligned}&\mathit{synctime} : \mathit{CONST} \times \mathbb{N}_0 \to \mathit{PhysicalTime}\\ &(c, n) \mapsto \mathit{up}(\mathit{fexecint}(c, n))\end{aligned}$$

Successful Discretization. The discretization is successful for the execution of a HL3 model $c \in \mathit{CONST}$, iff for every frame number, the model time synchronization takes place within that frame:

$$\begin{aligned}&\forall (c, n) \in \mathit{CONST} \times \mathbb{N}_0 \bullet\\ &\qquad \mathit{synctime}(c, n) \in [\mathit{low}(\mathit{frame}(c, n)), \mathit{up}(\mathit{frame}(c, n))[\end{aligned}$$

Therefore, a successful model execution will have exactly one time synchronization per frame. Because we assume (see above) that flow execution intervals do not start too early, it suffices to monitor that a least one synchronization exists per frame:

$$\begin{aligned}&\forall (c, n) \in \mathit{CONST} \times \mathbb{N}_0 \bullet \exists\, m \in \mathbb{N}_0 \bullet\\ &\qquad \mathit{synctime}(c, m) \in [\mathit{low}(\mathit{frame}(c, n)), \mathit{up}(\mathit{frame}(c, n))[\end{aligned}$$

Monitoring of Successful Discretization

In order to monitor the success or failure of discretization during a HL3 model execution, we define a constraint on the state space S of the operational semantics. Since we observe states $(c_{ob}, v_{ob}) \in S$ *during model execution*, only those states are accessible that reflect the current physical time t, i.e. $\mathit{physTime}(v_{ob}) = t$. The model time $\mathit{modelTime}(v_{ob})$ then reflects the most recent time from the past at which a time synchronization has taken place.

Because the time synchronization is monitored during the *complete* execution, it is not necessary to consider all past frames. In fact, only for the last elapsed frame, the time synchronization has to be checked. All preceding frames have been checked in the past already. Therefore,

$$\begin{aligned}&\exists\, m \in \mathbb{N}_0 \bullet \mathit{synctime}(c_{ob}, m) \in\\ &\qquad [\mathit{low}(\mathit{frame}(c_{ob}, \mathit{fno}_{time}(\mathit{physTime}(v_{ob})) - 1)),\\ &\qquad\ \mathit{up}(\mathit{frame}(c_{ob}, \mathit{fno}_{time}(\mathit{physTime}(v_{ob})) - 1))[\end{aligned}$$

must always hold.

For successful executions, the model time $\mathit{modelTime}(v_{ob})$ always reflects a synchronization time from either the current frame with number $\mathit{fno}_{time}(\mathit{physTime}(v_{ob}))$, or the last elapsed frame $\mathit{fno}_{time}(\mathit{physTime}(v_{ob})) - 1$.

The current frame is not relevant, because by definition, it is not finished yet, and time synchronization may occur within the frame in the future. However, the detection of a time synchronization for the current frame must not denote a failure, therefore we relax the constraint to

$$\begin{aligned}&\exists\, m \in \mathbb{N}_0 \bullet \mathit{synctime}(c_{ob}, m) \in \\ &\quad [\mathit{low}(\mathit{frame}(c_{ob}, \mathit{fno}_{\mathit{time}}(\mathit{physTime}(v_{ob})) - 1)), \\ &\quad\; \mathit{up}(\mathit{frame}(c_{ob}, \mathit{fno}_{\mathit{time}}(\mathit{physTime}(v_{ob}))))[\end{aligned}$$

Then, when checking this constraint at some time $\mathit{physTime}(v_{ob}) > \mathit{low}(\mathit{frame}(c, \mathit{fno}_{\mathit{time}}(\mathit{physTime}(v_{ob}))))$, i.e. *after* the current frame has begun, it can be satisfied even if the last elapsed frame has failed to synchronize time. But directly at the frame's beginning, i.e. at points in time $\mathit{physTime}(v_{ob}) = \mathit{low}(\mathit{frame}(c, \mathit{fno}_{\mathit{time}}(\mathit{physTime}(v_{ob}))))$, this failure is detected.

In order to check the constraint, the value of $\mathit{modelTime}(v_{ob})$ can be used to check it wrt. the given interval, because its first component directly encodes the existence of a corresponding time synchronization (see above):

$$\begin{aligned}&t_0(\mathit{modelTime}(v_{ob})) \in \\ &\quad [\mathit{low}(\mathit{frame}(c_{ob}, \mathit{fno}_{\mathit{time}}(\mathit{physTime}(v_{ob})) - 1)), \\ &\quad\; \mathit{up}(\mathit{frame}(c_{ob}, \mathit{fno}_{\mathit{time}}(\mathit{physTime}(v_{ob}))))[\end{aligned}$$

We define a mapping that checks this constraint on states $(c, v) \in S$ and explicitly states whether (c, v) is *synchronized* or not. Thereby, the expression is expressed directly, i.e. by resolving the definitions of *frame* and $\mathit{fno}_{\mathit{time}}$:

$$\begin{aligned}&\mathit{chktsync} : S \to \mathbb{B} \\ &(c, v) \mapsto \left\lfloor \frac{\mathit{physTime}(v)}{\delta_{\mathit{period}}(c)} \right\rfloor - 1 \leq \frac{t_0(\mathit{modelTime}(v))}{\delta_{\mathit{period}}(c)} < \left\lfloor \frac{\mathit{physTime}(v)}{\delta_{\mathit{period}}(c)} \right\rfloor + 1\end{aligned}$$

Discretization Examples. Figure 3.9 illustrates a successful synchronization sequence. For all physical times t in a frame $\mathit{fno}_{\mathit{time}}(c, t)$, the model time is either a time of the preceding frame $\mathit{fno}_{\mathit{time}}(c, t) - 1$, or of the current frame $\mathit{fno}_{\mathit{time}}(c, t)$.

In contrast, a scenario is given in figure 3.10, with synchronization missing in frame $\mathit{fno}_{\mathit{time}}(c, t) = 3$. This denotes a model execution that misses a discretization step and therefore fails to execute some continuous calculations in time.

A successful scheduling scenario is shown in figure 3.11: Calculations of all phases are fast enough. Different shades of grey denote the idle part of the flow phase (light) and the flow execution interval (dark). A failure scenario is given in figure 3.14: The flow phase (and flow execution interval) is started in time, but takes too much time.

We have left out the discussion of non-flow phases in this section, because they affect the discretization of time only in a straight-forward way: The set of execution phases *between* two consecutive flow phases can also take too much physical time. This is illustrated in figure 3.13.

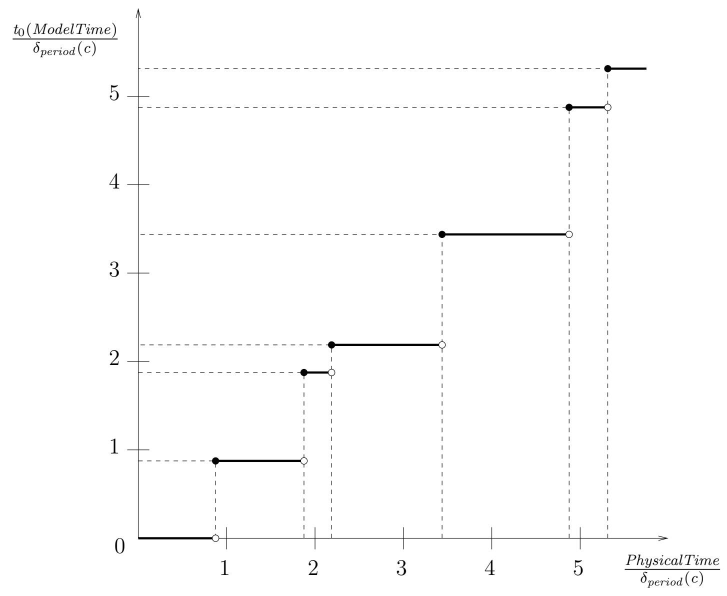


Figure 3.9: Model time must be synchronized within each frame for a successful execution of the HL3 model. The model time *is* the time of synchronization, therefore it identifies the frame of synchronization.

Time-Triggeredness

As a further restriction on admissible runs of a HL3 model, we require that every flow execution interval must be started with the beginning of its frame:

$$\forall (c,n) \in CONST \times \mathbb{N}_0 \bullet low(fexecint(c,n)) = low(frame(c,n))$$

See for example figures 3.12 and 3.15: Wrt. to the given constraints, these scenarios would be successful runs of a HL3 model. Nevertheless, we disallow the delay of flows, because we realize the *time-triggered* execution of flow calculations. See [Kop97b] for a discussion about time-triggered vs. event-triggered systems.

In order to check time-triggeredness, we have a second constraint on states $(c, v) \in S$:

$$chkttrig : S \to \mathbb{B}$$
$$(c,v) \mapsto nexttime_1(c,v) < physTime(v) \Rightarrow phase(sched(v)) = flow\_phase$$

where the function $nexttime_1$ is defined as:

$$nexttime_1 : S \to \mathbb{R}_0^+$$
$$s \mapsto nexttime(1,s)$$
$$nexttime : \mathbb{N} \times S \to \mathbb{R}_0^+$$
$$(k,(c,v)) \mapsto \left(\left\lfloor \frac{t_0(modelTime(v))}{k \cdot \delta_{period}(c)} \right\rfloor + 1 \right) \cdot k \cdot \delta_{period}(c)$$

This calculates the physical time value t_{fexint} at which the subjects of the next flow phase shall be actually executed. t_{fexint} is the smallest multiple of

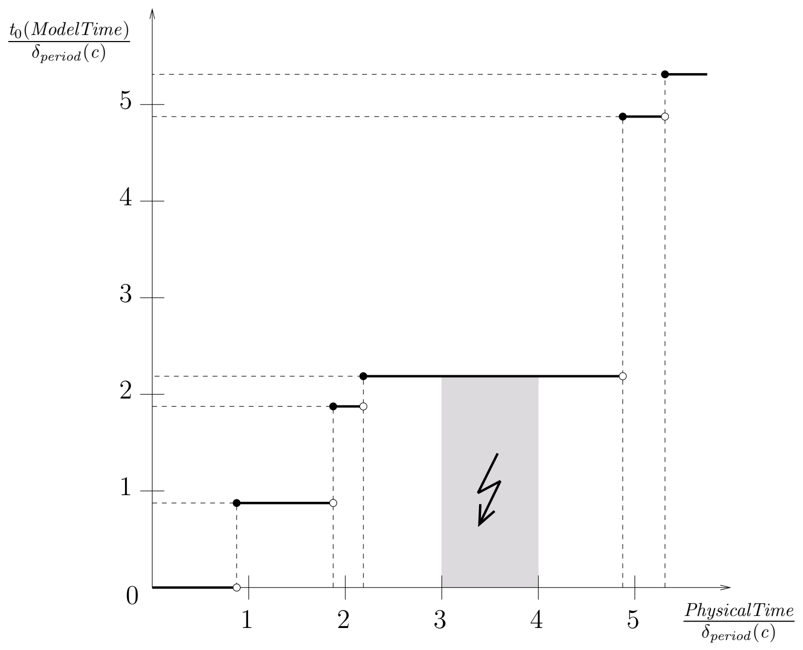


Figure 3.10: Model time synchronization scenario that fails in frame 3: at the beginning of frame 4, the model time is a time of frame 2 and therefore is too old.

the system period $\delta_{period}(c)$ which is bigger than current model time, i.e. the beginning of the next frame.

The constraint *chkttrig* then ensures that as soon as the time t_{fexint} is reached, a flow phase must be active. There is no need to ensure that this is no previous flow phase, because *chktsync* includes this.

Note that *nexttime* is the generalization of $nexttime_1$, which considers multiples of $\delta_{period}(c)$. It is used for the assignment of flows and interface modules to flow phases. *nexttime* is illustrated in figure 3.16.

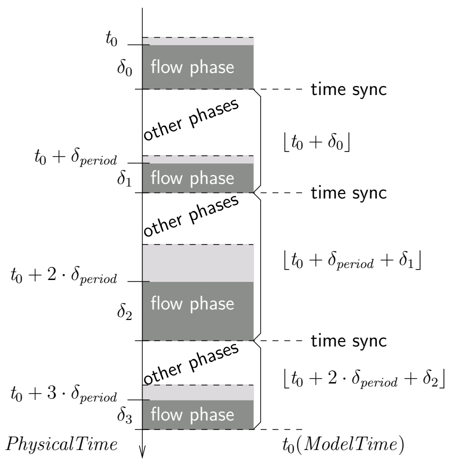
Synchronized and Fail States

Corresponding to the timing constraints given above, we have states $S_{sync} \subset S$ within the state space that are synchronized, i.e. $S_{sync} = \{s \in S \mid chktsync(s) \wedge chkttrig(s)\}$. The remaining states $S_{fail} = S \setminus S_{sync}$ are *fail states*. If the system execution ever passes one of the fail states, the fail flag $fail(v)$ will be set explicitly.

3.4.2 Switching between Execution Phases

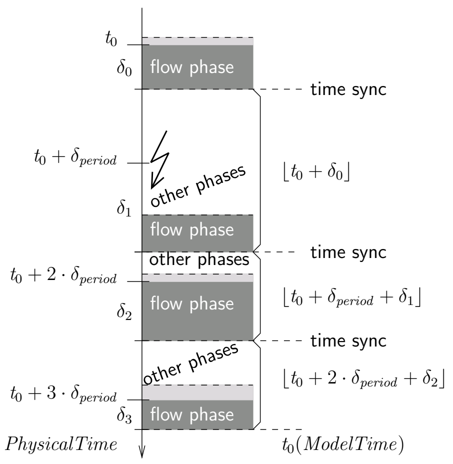
In this section, transitions $t \in T$ of the state transition system are defined, which represent the switching of execution phases. Each of these transitions require that there is nothing more to do for any subject in the pre-state $(c, v) \in S$, i.e. that the schedule is empty and that the scheduler controls every LWP (which means that they are idle):

$$subj_{sched}(sched(v)) = \varnothing \wedge \forall\, p \in lwp(c) \bullet \kappa_{LWP}(v)(p) = scheduler$$



t_0
δ_0
flow phase
time sync
other phases
$\lfloor t_0 + \delta_0 \rfloor$
$t_0 + \delta_{period}$
δ_1
flow phase
time sync
other phases
$\lfloor t_0 + \delta_{period} + \delta_1 \rfloor$
$t_0 + 2 \cdot \delta_{period}$
flow phase
δ_2
time sync
other phases
$\lfloor t_0 + 2 \cdot \delta_{period} + \delta_2 \rfloor$
$t_0 + 3 \cdot \delta_{period}$
δ_3
flow phase
$PhysicalTime$
$t_0(ModelTime)$

Figure 3.11: Successful execution: Model execution adjusts the model time $modelTime(v)$ once for every system period $\delta_{period}(c)$.



t_0
δ_0
flow phase
time sync
$t_0 + \delta_{period}$
other phases
$\lfloor t_0 + \delta_0 \rfloor$
δ_1
flow phase
time sync
other phases
$t_0 + 2 \cdot \delta_{period}$
flow phase
$\lfloor t_0 + \delta_{period} + \delta_1 \rfloor$
δ_2
time sync
other phases
$\lfloor t_0 + 2 \cdot \delta_{period} + \delta_2 \rfloor$
$t_0 + 3 \cdot \delta_{period}$
δ_3
flow phase
$PhysicalTime$
$t_0(ModelTime)$

Figure 3.12: Timing failure: Flow execution is delayed.

The switching of execution phases always takes some (small amount of) time $\delta_{switch} \in \mathbb{R}_+$.

Switching to update_phase. The *update_phase* is preceded by either a *notify_phase*, a *flow_phase*, or the *init_phase*. When all scheduled transitions have been executed and their abstract machines are notified, or all active flows have been executed for one duration step, or all abstract subjects are initialized, respectively, the *update_phase* is started. All abstract machines are scheduled to determine their enabled transitions and flows. Depending on the preceding phase, the model time is adjusted: (1) With a previous *notify_phase*, the visible time remains constant, but only the causality tick is incremented. This provides a zero-time execution of transitions for the model. (2) After a *flow_phase*, the visible time is synchronized with the physical time, such that for the model, time increases. (3) Succeeding the *init_phase*, model time is not adjusted, i.e. it remains zero.

Let $(c, v) \in S$ where for all LWPs $p \in lwp(c)$ we have that $\kappa_{LWP}(v)(p) = scheduler$.

Rule 3.4.1 If $sched(v) = (\emptyset, notify\_phase)$, then we have a transition $(c, v) \longrightarrow (c, v')$ with $v = v'$, except for

$$sched(v') = (am(c) \times \{update\}, update\_phase)$$
$$physTime(v') = physTime(v) + \delta_{switch}$$
$$\text{with } \delta_{switch} \in \mathbb{R}_+$$
$$modelTime(v') = t_0(modelTime(v)).(t_1(modelTime(v)) + 1) \qquad \square$$

Rule 3.4.2 If $sched(v) = (\emptyset, flow\_phase)$ and $physTime(v) \geq nexttime_1(c, v)$, then a transition $(c, v) \longrightarrow (c, v')$ exists which has

$$sched(v') = (am(c) \times \{update\}, update\_phase)$$

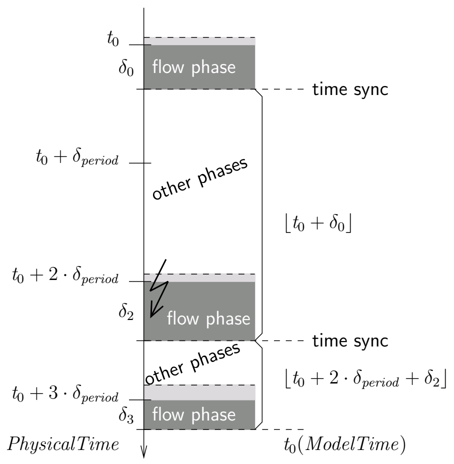


Figure 3.13: Timing failure: Non-flow phases take too long.

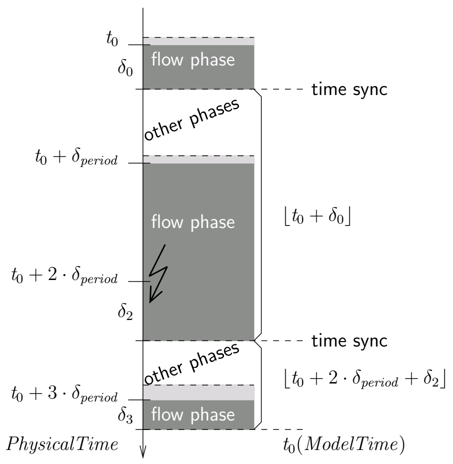


Figure 3.14: Timing failure: Flow phase takes too long.

$$\begin{aligned}
&physTime(v') = physTime(v) + \delta_{switch}\\
&\qquad \text{with } \delta_{switch} \in \mathbb{R}_+\\
&modelTime(v') = \lfloor physTime(v) \rfloor .0
\end{aligned}$$

and otherwise $v = v'$. □

Note that flow phases cannot be left before the physical time for flow execution is reached. This ensures, that empty flow phases, i.e. flow phases which will neither schedule flows nor interface modules, are treated the same way as non-empty ones are.
Finally,

Rule 3.4.3 if $sched(v) = (\emptyset, init\_phase)$, then a transition $(c, v) \longrightarrow (c, v')$ exists which modifies v only by

$$\begin{aligned}
&sched(v') = (am(c) \times \{update\}, update\_phase)\\
&physTime(v') = physTime(v) + \delta_{switch}\\
&\qquad \text{with } \delta_{switch} \in \mathbb{R}_+
\end{aligned}$$
□

Switching to selection_phase. After an *update_phase*, a *selection_phase* follows. Control is taken over by the selector, in order to create a *selection* of discrete or flow steps from the abstract machine's states.

Rule 3.4.4 Let $(c, v) \in S$ where for all LWPs $p \in lwp(c)$ we have that $\kappa_{LWP}(v)(p) = scheduler$, and moreover that $sched(v) = (\emptyset, update\_phase)$. We thus have a transition $(c, v) \longrightarrow (c, v')$ with $v = v'$, except for

$$\begin{aligned}
&sched(v') = (\{(selector, getSelection)\}, selection\_phase)\\
&physTime(v') = physTime(v) + \delta_{switch}\\
&\qquad \text{with } \delta_{switch} \in \mathbb{R}_+
\end{aligned}$$
□

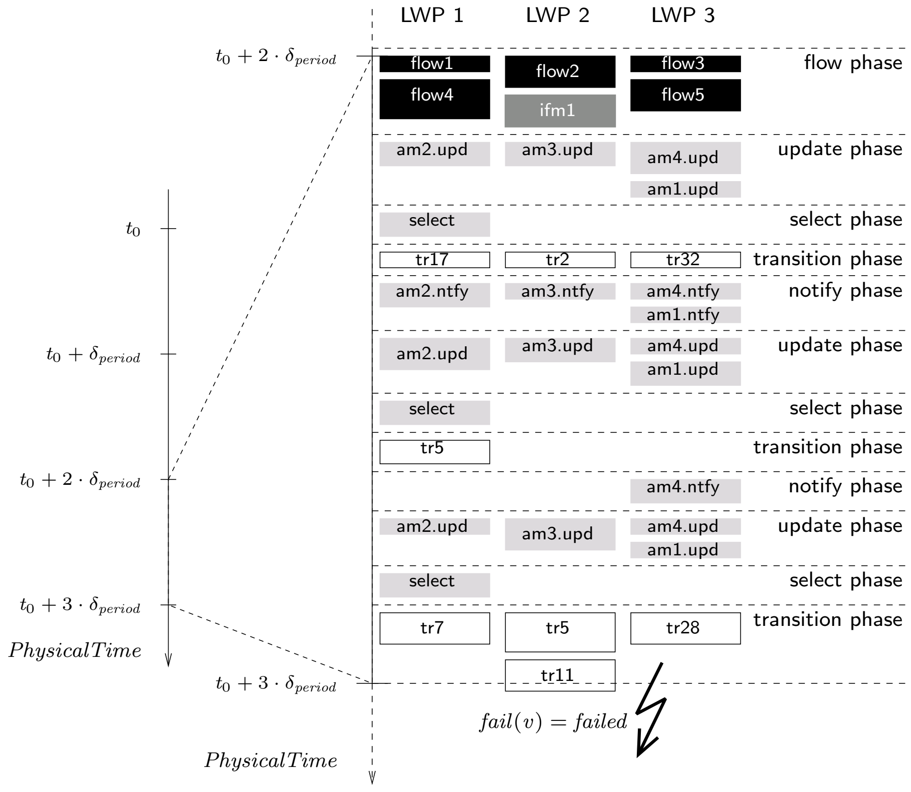


Figure 3.15: Scheduling example: timing failure.

Switching to transition_phase or flow_phase. The transition to *transition_phase* or *flow_phase* depends on the selector's choice. Therefore it is defined with the selection rule in section 3.5.

Switching to notify_phase. A *transition_phase* is always succeeded by a *notify_phase*. All abstract machines for which a transition was taken are scheduled to update their internal states accordingly.

Rule 3.4.5 Let $(c, v) \in S$ where for all LWPs $p \in lwp(c)$ we have that $\kappa_{LWP}(v)(p) = scheduler$, and moreover that $sched(v) = (\varnothing, transition\_phase)$. We thus have a transition $(c, v) \longrightarrow (c, v')$ with $v = v'$, except for

$$\begin{aligned} sched(v') \quad = \quad & (\{m \in am(c) \mid trans_{AbsState}(\kappa_{Subj_{abs}}(m)) \neq \lambda\} \\ & \times \{notifyTrans\}, \\ & notify\_phase) \\ physTime(v') = {} & physTime(v) + \delta_{switch} \\ & \text{with } \delta_{switch} \in \mathbb{R}_+ \end{aligned}$$ □

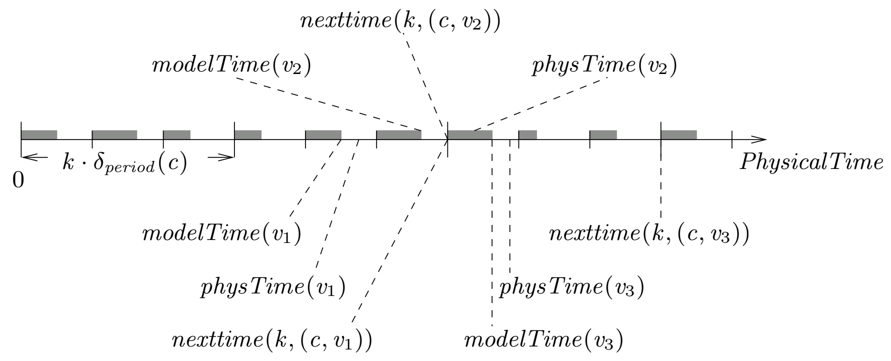


Figure 3.16: Illustration of the mapping *nexttime*, with factor $k = 3$: For every state $(c, v) \in S$ and k, the beginning of the next frame such that $fno_{time}(c, t)$ is a multiple of k is calculated. However, during the flow execution interval of that frame, its own start time results, as for (c, v_2) in this example.

3.4.3 Scheduling within Execution Phases

Within execution phases, single subjects have to be scheduled, depending on the schedule $sched(v)$ of the current state $(c, v) \in S$, and depending on the availability of LWPs $lwp(c)$. Therefore, in this section transitions $t \in T$ are defined that each represent the *allocation* of a single subject on an available LWP. For the allocation of a subject, the corresponding LWP needs a time duration $\delta_{alloc} \in \mathbb{R}_+$ before the execution of the subject's operation begins. We model this by adding δ_{alloc} to the execution duration $\delta_s \in \mathbb{R}_+$ of the operation or program, respectively.

The deallocation is not presented here, since it is a direct result of the termination of subject execution, and thus is included in the definitions of sections 3.5 and 3.6.

The way of how single subjects are allocated is different for abstract subjects and program subjects. Within each execution phase, only either kind of subjects can be executed, therefore we distinguish the allocation rules by these both kinds.

Further, there is a specific rule that lets time pass for idle LWPs. This is done exactly when a *flow_phase* is active, but the next (physical time) frame is not reached yet. Thus, the system waits for the next flow execution interval.

Dynamic LWP Assignment for Program Subjects. Program subjects $s \in Subj_{prog}$ in the schedule $sched(v)$ are scheduled immediately on any idle LWP in order to execute operations $op \in Op$:

Let $(c, v) \in S$ for some LWP $p \in lwp(c)$, such that $\kappa_{LWP}(v)(p) = scheduler$, and moreover that $sched(v) = (sset, sphase)$ with $(s, op) \in sset$. Then subject s is selected and p is assigned to s, and s is immediately removed from the schedule. The program string is reset, i.e. the complete program string for operation op as defined in the HL3 model is assigned, and a remaining duration for the first statement (including the duration needed for allocation) is set. The program's (optional) parameter is not modified.

If additionally, $sphase = transition\_phase$ (which implies $s \in Trans$), then the transition is set as the parameter for the next execution of the associated abstract machine, which will be the notification of the transition's execution. Internal state and duration are kept for the abstract machine.

Rule 3.4.6 Thus, for $sphase = transition\_phase$ we have a transition $(c, v) \longrightarrow (c, v')$ with $v = v'$, except for

$$\begin{aligned}
&\kappa_{LWP}(v') = \kappa_{LWP}(v) \oplus \{p \mapsto s\}\\
&sched(v') = ((sset \setminus \{s\}), sphase)\\
&\kappa_{Subj_{prog}}(v') = \kappa_{Subj_{prog}}(v) \oplus \{(s, op) \mapsto\\
&\qquad (prg_{subj}(c)(s, op), \delta_s + \delta_{alloc}, vis_{ProgState}(\kappa_{Subj_{prog}}(v)(s, op)))\}\\
&\qquad \text{with } \delta_s, \delta_{alloc} \in \mathbb{R}_+\\
&\kappa_{Subj_{abs}}(v') = \kappa_{Subj_{abs}}(v) \oplus \{am_{trans}(c)(s) \mapsto\\
&\qquad (intState_{AbsState}(\kappa_{Subj_{abs}}(v)(am_{trans}(c)(s))),\\
&\qquad \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(am_{trans}(c)(s))),\\
&\qquad s)\}
\end{aligned}$$

□

Rule 3.4.7 For all phases $sphase \neq transition\_phase$, we have a transition $(c, v) \longrightarrow (c, v')$ that does not affect the execution state of abstract machines, i.e. $v = v'$, except for

$$\begin{aligned}
&\kappa_{LWP}(v') = \kappa_{LWP}(v) \oplus \{p \mapsto s\}\\
&sched(v') = ((sset \setminus \{s\}), sphase)\\
&\kappa_{Subj_{prog}}(v') = \kappa_{Subj_{prog}}(v) \oplus \{(s, op) \mapsto\\
&\qquad (prg_{subj}(c)(s, op), \delta_s + \delta_{alloc}, vis_{ProgState}(\kappa_{Subj_{prog}}(v)(s, op)))\}\\
&\qquad \text{with } \delta_s, \delta_{alloc} \in \mathbb{R}_+
\end{aligned}$$

□

There is an additional restriction for $sphase = flow\_phase$ – the above transition is only allowed, if $physTime(v) \geq nexttime_1(c, v)$. That means, within a flow phase, the subjects in the schedule are scheduled as recently as the physical time for the next flow executions has come. Previously, physical time just has to pass.

Waiting for Flow Execution. A flow phase is idle and lets time pass, as long as the physical time for the next flow executions wrt. the system period is not reached yet.

Rule 3.4.8 Let $(c, v) \in S$ with $phase(sched(v)) = flow\_phase$ and $physTime(v) < nexttime_1(c, v)$. Then some time $\delta \in \mathbb{R}_+$ passes, i.e. transitions $(c, v) \longrightarrow (c, v') \in T$ exist with $v = v'$, but

$$\begin{aligned}
&physTime(v') = physTime(v) + \delta\\
&\qquad \text{with } \delta \leq nexttime_1(c, v) - physTime(v)
\end{aligned}$$

□

Static LWP Assignment for Abstract Subjects. An abstract subject $s \in Subj_{abs}$ in the schedule $sched(v)$ is scheduled for operation $op \in Op$ on its statically assigned LWP p, as soon as it is available:

Rule 3.4.9 Let $(c, v) \in S$ for $sched(v) = (sset, sphase)$ with $(s, op) \in sset$, and let $\kappa_{LWP}(v)(lwp_{subj}(c)(s)) = scheduler$. Then p is mapped to subject s, and s is removed from the schedule. The internal state and the subject's (optional) parameter are retained, but a remaining duration for the allocation and execution of the operation is set. We have a transition $(c, v) \longrightarrow (c, v')$ with $v = v'$, except for

$$\begin{aligned}
&\kappa_{LWP}(v') = \kappa_{LWP}(v) \oplus \{lwp_{subj}(c)(s) \mapsto s\}\\
&sched(v') = ((sset \setminus \{s\}), sphase)\\
&\kappa_{Subj_{abs}}(v') = \kappa_{Subj_{abs}}(v) \oplus \{s \mapsto (intState_{AbsState}(\kappa_{Subj_{abs}}(v)(s)),\\
&\qquad \delta_s + \delta_{alloc}, trans_{AbsState}(\kappa_{Subj_{abs}}(v)(s)))\}\\
&\qquad \text{with } \delta_s, \delta_{alloc} \in \mathbb{R}_+
\end{aligned}$$

□

3.5 Abstract Subject Execution

In this section, the execution of abstract subjects is defined by transitions $t \in T$. Abstract subjects are executed in that a specific operation of the corresponding subject is run. The execution of the operation takes some time before it is terminated. The operation's effect on the state space becomes effective with its termination.

Further, for $|lwp(c)| > 1$ it is possible that several abstract subjects are executed in parallel, i.e. time evolves for all active subjects in common. This is defined for all possible abstract subject operations by the *progress rule* given below.

As a side effect of progress, the timing constraints identified in section 3.4.1 may be violated. This is regarded in the progress rule, and leads to the setting of the fail flag.

In contrast to the *progress* of operations, the *effect* of operations depends on the specific operation and the corresponding kind of subject. Therefore, different *termination rules* are provided for the possible operations, i.e. for

$$\begin{aligned}
&op \in \{(Am, init), (Am, update), (Am, notifyTrans), (selector, init),\\
&\qquad (selector, getSelection)\}
\end{aligned}$$

3.5.1 Progress of Abstract Subject Execution

Operations of abstract subjects are executed, as soon as a maximum subset from the schedule is allocated on LWPs, such that there is no idle LWP left or the schedule is empty. Then, operations take some time.

Let $(c, v) \in S$ with $phase(sched(v)) \in \{update\_phase, selection\_phase, notify\_phase\}$. Given that all active subjects have a positive remaining execution duration, i.e. $\forall s \in \operatorname{ran} \kappa_{LWP}(v) \bullet \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(s)) \in \mathbb{R}_+$, and that there are no more idle LWPs left that could be used to execute some subject in the schedule, i.e. $\neg \exists s \in subj_{sched}(sched(v)) \bullet \kappa_{LWP}(v)(lwp_{subj}(c)(s)) = scheduler$. Then arbitrary time durations can elapse, as long as the first subject is not completed, yet.

Consider the potential successor states $Suc \subseteq S$, which are given for all $\delta \leq \delta_{min}$, whereas δ_{min} is the minimum value of the set of all remaining durations $\{\delta \in \mathbb{R}_+ \mid \exists\, s \in \operatorname{dom} \kappa_{Subj_{abs}}(v) \bullet \delta = \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(s))\}$, such that $v = v'$ holds for $(c, v') \in Suc$, except for

$$\begin{aligned}
&\kappa_{Subj_{abs}}(v') = \kappa_{Subj_{abs}}(v) \oplus \{s \mapsto \\
&\qquad (intState_{AbsState}(\kappa_{Subj_{abs}}(v)(s)), \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(s)) - \delta, \\
&\qquad\ trans_{AbsState}(\kappa_{Subj_{abs}}(v)(s)) \\
&\qquad \mid s \in \operatorname{dom} \kappa_{Subj_{abs}}(v) \wedge \delta_{AbsState}(\kappa_{Subj_{abs}}(v)(s)) \in \mathbb{R}_+\} \\
&physTime(v') = physTime(v) + \delta
\end{aligned}$$

Since between (c, v) and (c, v') physical time elapses, time synchronization (see section 3.4.1) is monitored here. The successor states are therefore divided into $Suc_{sync} = Suc \cap S_{sync}$ and $Suc_{fail} = Suc \cap S_{fail}$.

Rule 3.5.1 For all $(c, v') \in Suc_{sync}$, there are transitions $(c, v) \longrightarrow (c, v') \in T$ that let the respective time durations $\delta \leq \delta_{min}$ pass. $\square$

Rule 3.5.2 For the states $(c, v') \in Suc_{fail}$ we have transitions $(c, v) \longrightarrow (c, v'') \in T$ that represent the evolution of time durations in the same way, but also set the fail flag. Therefore, $v' = v''$ holds, with the exception of

$$fail(v'') = failed \qquad \square$$

Note that timing failures are thus detected in finite time, which happens with at most the delay of δ_{min}.

3.5.2 Termination of Abstract Subject Execution

Termination of Abstract Machines' Initialization. Initialization of abstract machines assigns a pre-defined internal state to the corresponding abstract machine, provided by the mapping

$$init_{Am} : Am \to IntState$$

Rule 3.5.3 Then, for states $(c, v) \in S$ with an abstract machine m for which the current execution time has elapsed in the *init_phase*, that is $\delta_{AbsState}(\kappa_{Subj_{abs}}(v)(m)) = 0$ and $phase(sched(v)) = init\_phase$, there is a transition $(c, v) \longrightarrow (c, v') \in T$ with

$$\begin{aligned}
&\kappa_{Subj_{abs}}(v') = \kappa_{Subj_{abs}}(v) \oplus \{m \mapsto (init_{Am}(m), \infty, \lambda)\} \\
&\kappa_{LWP}(v') = \kappa_{LWP}(v) \oplus \{p \mapsto scheduler \mid \kappa_{LWP}(v)(p) = m\} \qquad \square
\end{aligned}$$

such that the internal state is assigned, and the formerly used LWP is released.

Termination of Abstract Machines' Update. The effect of the operation *update* is that the abstract machine (1) has an actual internal state, (2) has an actual abstract machine state, and (3) each of the flows it controls is enabled or disabled, corresponding to the actual internal state. Input for the *update* operation is the previous internal state of the abstract machine, along with the parts of the state space that are visible to subjects, including the current channel valuations.

Formally, the effect is given by a mapping from the internal abstract machine state and the model-accessible portion of the state space S to a succeeding abstract machine state, along with a flow enabling function:

$$
\begin{aligned}
&\mathit{UpdState}_{pre} = \mathit{Am} \times \mathit{IntState} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \\
&\mathit{UpdState}_{post} = \mathit{IntState} \times \mathit{AmState} \times (\mathit{Flow} \nrightarrow \mathbb{B}) \\
&\mathit{update} : \mathit{UpdState}_{pre} \rightarrow \mathit{UpdState}_{post}
\end{aligned}
$$

The resulting components are accessible through

$$
\begin{aligned}
&\mathit{update}_{\mathit{IntState}} : \mathit{UpdState}_{pre} \rightarrow \mathit{IntState} \\
&\qquad s \mapsto \pi_1\ \mathit{UpdState}_{post}(\mathit{update}(s)) \\
&\mathit{update}_{\mathit{AmState}} : \mathit{UpdState}_{pre} \rightarrow \mathit{AmState} \\
&\qquad s \mapsto \pi_2\ \mathit{UpdState}_{post}(\mathit{update}(s)) \\
&\mathit{update}_{\mathit{Flow}} : \mathit{UpdState}_{pre} \rightarrow (\mathit{Flow} \nrightarrow \mathbb{B})\} \\
&\qquad s \mapsto \pi_3\ \mathit{UpdState}_{post}(\mathit{update}(s))
\end{aligned}
$$

Rule 3.5.4 For states $(c, v) \in S$ which have an abstract machine $m \in am(c)$ whose current operation is about to terminate within the update phase, i.e. $\delta_{AbsState}(\kappa_{Subj_{abs}}(v)(m)) = 0$ and $phase(sched(v)) = update\_phase$, we have a transition $(c, v) \longrightarrow (c, v') \in T$ with

$$
\begin{aligned}
&\kappa_{Subj_{abs}}(v') = \kappa_{Subj_{abs}}(v) \oplus \{m \mapsto (\mathit{update}_{\mathit{IntState}}(\mathit{preState}), \infty, \lambda)\} \\
&\kappa_{LWP}(v') = \kappa_{LWP}(v) \oplus \{p \mapsto \mathit{scheduler} \mid \kappa_{LWP}(v)(p) = m\} \\
&\kappa_{Am}(v') = \kappa_{Am}(v) \oplus \{m \mapsto \mathit{update}_{\mathit{AmState}}(\mathit{preState})\} \\
&\kappa_{Flow}(v') = \kappa_{Flow}(v) \oplus \\
&\qquad (\{f \mapsto \infty \mid f \in \mathrm{dom}(\mathit{update}_{\mathit{Flow}}(\mathit{preState})) \\
&\qquad\quad \wedge \neg \mathit{update}_{\mathit{Flow}}(\mathit{preState})(f) \wedge \mathit{am}_{\mathit{flow}}(c)(f) = m\} \\
&\qquad \cup \\
&\qquad \{f \mapsto \mathit{nexttime}_{\mathit{flow}}(f, (c, v)) \mid f \in \mathrm{dom}(\mathit{update}_{\mathit{Flow}}(\mathit{preState})) \\
&\qquad\quad \wedge \mathit{update}_{\mathit{Flow}}(\mathit{preState})(f) \wedge \mathit{am}_{\mathit{flow}}(c)(f) = m\})
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{preState} \quad = \quad & (m, \mathit{intState}_{\mathit{AbsState}}(\kappa_{Subj_{abs}}(v)(m)), \\
& (\mathit{subject}_{var}(c), \mathit{chan}_{port}(c), \mathit{subject}_{port}(c)), \\
& (\mathit{modelTime}(v), \sigma_{Var}(v), \kappa_{Chan}(v)))
\end{aligned}
\qquad \square
$$

Here, (1) the internal state of m is updated, and the remaining execution duration is reset. (2) m is deallocated from its current LWP. (3) The abstract machine state is updated. (4) All flows under control of m are either disabled, or get their new execution time. Here a new value is assigned to indicate when flows are to be scheduled. A flow is supposed to get scheduled at the next instance where its period would allow its execution:

$$
\begin{aligned}
&\mathit{nexttime}_{\mathit{flow}} : \mathit{Flow} \times S \rightarrow \mathbb{R}_0^+ \\
&\qquad (f, (c, v)) \mapsto \mathit{nexttime}(\mathit{period}_{\mathit{flow}}(c)(f), (c, v))
\end{aligned}
$$

Termination of Abstract Machines' Notification. The effect of the operation *notifyTrans* is that the abstract machine has an updated internal state, which regards the previous execution of one of its controlled transitions. Input for the *notifyTrans* operation is the previous internal state of the abstract machine, along with the executed transition.

Formally, the effect is given by a mapping from the internal abstract machine state and the executed transition t to a succeeding internal abstract machine state:

$$\mathit{notify} : \mathit{Am} \times \mathit{IntState} \times \mathit{Trans} \to \mathit{IntState}$$

Rule 3.5.5 For states $(c, v) \in S$ which have an abstract machine $m \in \mathit{am}(c)$ whose current operation is about to terminate within the notify phase, i.e. $\delta_{\mathit{AbsState}}(\kappa_{\mathit{Subj}_{abs}}(v)(m)) = 0$ and $\mathit{phase}(\mathit{sched}(v)) = \mathit{notify\_phase}$, we have a transition $(c, v) \longrightarrow (c, v') \in T$ with

$$\begin{aligned}
&\kappa_{\mathit{Subj}_{abs}}(v') = \kappa_{\mathit{Subj}_{abs}}(v) \oplus \{m \mapsto (\mathit{notify}(m,\\
&\qquad \mathit{intState}_{\mathit{AbsState}}(\kappa_{\mathit{Subj}_{abs}}(v)(m)),\\
&\qquad \mathit{trans}_{\mathit{AbsState}}(\kappa_{\mathit{Subj}_{abs}}(v)(m))),\\
&\qquad \infty, \lambda)\}\\
&\kappa_{\mathit{LWP}}(v') = \kappa_{\mathit{LWP}}(v) \oplus \{p \mapsto \mathit{scheduler} \mid \kappa_{\mathit{LWP}}(v)(p) = m\} \qquad \square
\end{aligned}$$

The execution state of the abstract machine is modified, such that (1) the internal state is updated, (2) the remaining execution duration is reset, (3) and the parameter entry, which was the executed transition, is removed. The corresponding LWP is released, i.e. it is controlled by the scheduler now.

Termination of Selector's Initialization. The initialization of the selector assigns a pre-defined internal state to it, and further determines if the state space is *well-formed*, corresponding to constraints which depend on the selector's high-level formalism. This is given by the mapping

$$\mathit{init}_{sel} : S \to \mathit{IntState} \times \mathbb{B}$$

and the corresponding projections

$$\begin{aligned}
&\mathit{intState}_{\mathit{initsel}} : S \to \mathit{IntState}\\
&\qquad s \mapsto \pi_1\, \mathit{init}_{sel}(s)\\
&\mathit{wellFormed}_{\mathit{initsel}} : S \to \mathbb{B}\\
&\qquad s \mapsto \pi_2\, \mathit{init}_{sel}(s)
\end{aligned}$$

Rule 3.5.6 Then, for states $(c, v) \in S$ for which the selector's execution time has elapsed in the *init_phase*, that is $\delta_{\mathit{AbsState}}(\kappa_{\mathit{Subj}_{abs}}(v)(\mathit{selector})) = 0$ and $\mathit{phase}(\mathit{sched}(v)) = \mathit{init\_phase}$, there is a transition $(c, v) \longrightarrow (c, v') \in T$ with

$$\begin{aligned}
&\kappa_{\mathit{Subj}_{abs}}(v') = \kappa_{\mathit{Subj}_{abs}}(v) \oplus \{\mathit{selector} \mapsto (\mathit{intState}_{\mathit{initsel}}(c, v), \infty, \lambda)\}\\
&\kappa_{\mathit{LWP}}(v') = \kappa_{\mathit{LWP}}(v) \oplus \{p \mapsto \mathit{scheduler} \mid \kappa_{\mathit{LWP}}(v)(p) = \mathit{selector}\}\\
&\mathit{fail}(v') = \begin{cases} \mathit{ok} & \text{if } \mathit{wellFormed}_{\mathit{initsel}}(c, v)\\ \mathit{failed} & \text{else} \end{cases} \qquad \square
\end{aligned}$$

such that the internal state is assigned, the formerly used LWP is released, and the fail flag is set according to the initialization's result.

Termination of Selector's Selection. The actual behavior of the *selector* component depends on both the high-level formalism (and the respective transformation function to HL3) and a user-defined *selection policy* determined by the context which the real-time execution is investigated in. The *selector*'s behavior thus is treated as an abstract subject which takes time for its execution and provides a *selection* upon termination. The selection is mainly determined from the states $\kappa_{Am}(v)$ of the abstract machines, along with its internal state. Additionally, the model time as well as a dedicated set of ports may be accessed.

$$\begin{aligned}
&\mathit{Selection} = \mathbb{B} \times (\mathit{Trans} \nrightarrow \mathit{VisibilitySet})\\
&\mathit{select} : \mathit{ModelTime} \times \mathcal{P}(\mathit{Port}) \times \mathit{IntState} \times \Sigma_{Am}\\
&\qquad \to \mathit{IntState} \times \mathit{Selection} \times \mathit{ChanState}
\end{aligned}$$

The selection consists of (1) a boolean value denoting whether a flow step is possible and (2) a set of transitions that constitute a possible discrete step. The set of transitions is given by the domain of a mapping, which also provides a visibility set for each of the transitions. The time stamps at which the results of the transitions' actions become effective for the respective recipients therefore depend on the high-level formalism. (3) Further, a new internal state results for the selector. (4) As a side-effect, data can be written to channels.
The components of the selection are directly given by

$$\begin{aligned}
&\mathit{sel}_{\mathit{flow}} : \mathit{ModelTime} \times \mathcal{P}(\mathit{Port}) \times \mathit{IntState} \times \Sigma_{Am} \to \mathbb{B}\\
&\qquad (\mathit{tick}, P, s_i, s_{am}) \mapsto \pi_1\,(\pi_2\,\mathit{select}(\mathit{tick}, P, s_i, s_{am}))\\
&\mathit{sel}_{\mathit{trans,vis}} : \mathit{ModelTime} \times \mathcal{P}(\mathit{Port}) \times \mathit{IntState} \times \Sigma_{Am}\\
&\qquad \to (\mathit{Trans} \nrightarrow \mathit{VisibilitySet})\\
&\qquad (\mathit{tick}, P, s_i, s_{am}) \mapsto \pi_2\,(\pi_2\,\mathit{select}(\mathit{tick}, P, s_i, s_{am}))
\end{aligned}$$

Transitions without visibility sets are directly accessible by

$$\begin{aligned}
&\mathit{sel}_{\mathit{trans}} : \mathit{ModelTime} \times \mathcal{P}(\mathit{Port}) \times \mathit{IntState} \times \Sigma_{Am} \to \mathcal{P}(\mathit{Trans})\\
&\qquad (\mathit{tick}, P, s_i, s_{am}) \mapsto \operatorname{dom} \mathit{sel}_{\mathit{trans,vis}}(\mathit{tick}, P, s_i, s_{am})
\end{aligned}$$

The new internal state can be accessed through

$$\begin{aligned}
&\mathit{sel}_{\mathit{intState}} : \mathit{ModelTime} \times \mathcal{P}(\mathit{Port}) \times \mathit{IntState} \times \Sigma_{Am} \to \mathit{IntState}\\
&\qquad (\mathit{tick}, P, s_i, s_{am}) \mapsto \pi_1\,\mathit{select}(\mathit{tick}, P, s_i, s_{am})
\end{aligned}$$

The resulting channel state is given by

$$\begin{aligned}
&\mathit{sel}_{\mathit{chanEntry}} : \mathit{ModelTime} \times \mathcal{P}(\mathit{Port}) \times \mathit{IntState} \times \Sigma_{Am} \to \mathit{ChanState}\\
&\qquad (\mathit{tick}, P, s_i, s_{am}) \mapsto \pi_3\,\mathit{select}(\mathit{tick}, P, s_i, s_{am})
\end{aligned}$$

There are several constraints that must hold for the results of the *select* effect: (1) The set of selected transitions is chosen according to the sets of enabled transitions of the abstract machines. (2) In this set there is at most one associated transition for each abstract machine. (3) The selector is only allowed to allow a flow step, if all abstract machines agree. (4) The selector is not allowed to mischievously block the execution of a HL3 program – in case that no flow is possible, but some abstract machines do have transitions enabled, at least one transition will be selected.

$$\begin{aligned}
&\forall (tick, P, s_i, s_{am}) \in \operatorname{dom} select \bullet \\
&\quad sel_{trans}(tick, P, s_i, s_{am}) \subseteq \bigcup_{m \in \operatorname{dom} s_{am}} trans_{AmState}(s_{am}(m)) \\
&\forall (tick, P, s_i, s_{am}) \in \operatorname{dom} select \bullet \\
&\quad \forall\, t_1, t_2 \in sel_{trans}(tick, P, s_i, s_{am}) \bullet \exists\, m_1, m_2 \in \operatorname{dom} s_{am} \bullet \\
&\qquad t_1 \in trans_{AmState}(s_{am}(m_1)) \wedge t_2 \in trans_{AmState}(s_{am}(m_2)) \\
&\qquad \wedge (t_1 \neq t_2 \Rightarrow m_1 \neq m_2) \\
&\forall (tick, P, s_i, s_{am}) \in \operatorname{dom} select \bullet \\
&\quad sel_{flow}(tick, P, s_i, s_{am}) \Rightarrow \forall\, m \in \operatorname{dom} s_{am} \bullet flow_{AmState}(s_{am}(m)) \\
&\forall (tick, P, s_i, s_{am}) \in \operatorname{dom} select \bullet \\
&\quad \left(\bigcup_{m \in \operatorname{dom} s_{am}} trans_{AmState}(s_{am}(m)) \neq \varnothing \wedge \neg sel_{flow}(s_{am}) \right) \\
&\quad \Rightarrow sel_{trans}(s) \neq \varnothing
\end{aligned}$$

Let $(c, v) \in S$ where within the selection phase, the selector's current operation is about to terminate, i.e. $sched(v) = (\varnothing, selection\_phase)$ and $\delta_{AbsState}(\kappa_{Subj_{abs}}(v)(selector)) = 0$. Further, let the current internal state of the selector $s_i = intState_{AbsState}(\kappa_{Subj_{abs}}(v)(selector))$. Then, the selector's operation *getSelection* is finished, and provides a selection.

Rule 3.5.7 If this selection contains any transitions, that means if $sel_{trans}(s_i, \kappa_{Am}(v)) \neq \varnothing$, we have a transition $t_{disc} = (c, v) \longrightarrow (c, v')$ with $v = v'$, except that the transitions of the selection constitute the scheduled subjects now, and further the visibility sets are set as parameters for the transitions' execution states, in combination with the transitions' static visibility sets (see section 3.3 for operation $\Cap$):

$$\begin{aligned}
&sched(v') = (sel_{trans}(modelTime(v), selport(c), s_i, \kappa_{Am}(v)) \times \{action\}, \\
&\quad transition\_phase) \\
&physTime(v') = physTime(v) + \delta_{switch} \\
&\quad \text{with } \delta_{switch} \in \mathbb{R}_+ \\
&\kappa_{Subj_{prog}}(v') = \kappa_{Subj_{prog}}(v) \oplus \\
&\quad \{(t, action) \mapsto (string_{ProgState}(\kappa_{Subj_{prog}}(v)(t, action)), \\
&\quad\ \ \delta_{ProgState}(\kappa_{Subj_{prog}}(v)(t, action)), vis_{trans}(c)(t) \Cap vis) \\
&\quad\ \ \mid (t \mapsto vis) \in sel_{trans,vis}(modelTime(v), selport(c), s_i, \kappa_{Am}(v))\} \\
&\kappa_{Subj_{abs}}(v') = \kappa_{Subj_{abs}}(v) \oplus \\
&\quad \{selector \mapsto (sel_{intState}(modelTime(v), selport(c), s_i, \kappa_{Am}(v)), \infty, \lambda)\} \\
&\kappa_{Chan}(v') = \kappa_{Chan}(v) \oplus \\
&\quad \{insert_{Port}(p, (data, \{tick, p\}), \\
&\qquad (subject_{var}(c), chan_{port}(c), subject_{port}(c)), \\
&\qquad (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v))))
\end{aligned}$$

$$\begin{aligned}
&\mid (\mathit{tick}, p) \mapsto \mathit{data} \\
&\quad \in \mathit{sel}_{\mathit{chanEntry}}(\mathit{modelTime}(v), \mathit{selport}(c), s_i, \kappa_{Am}(v))\} \qquad \Box
\end{aligned}$$

Additionally, the selector's internal state is updated, and the data values are inserted into the respective channels.

Rule 3.5.8 We have another transition $t_{\mathit{flow}} = (c, v) \longrightarrow (c, v'')$ provided that $\mathit{sel}_{\mathit{flow}}(s_i, \kappa_{Am}(v))$, i.e. that the selection allows a flow step.

$$\begin{aligned}
&\mathit{sched}(v'') = \\
&\quad ((\{f \in \mathit{flow}(c) \mid \kappa_{Flow}(v)(f) = \mathit{nexttime}_1(c, v)\} \times \{\mathit{integrate}\}) \\
&\quad \cup(\{\mathit{md} \in \mathit{ifm}(c) \mid \mathit{nexttime}_{\mathit{ifm},\mathit{poll}}(\mathit{md}, (c, v)) = \\
&\qquad \mathit{nexttime}_1(c, v)\} \times \{\mathit{poll}\}) \\
&\quad \cup(\{\mathit{md} \in \mathit{ifm}(c) \mid \mathit{nexttime}_{\mathit{ifm},\mathit{tmit}}(\mathit{md}, (c, v)) = \\
&\qquad \mathit{nexttime}_1(c, v)\} \times \{\mathit{transmit}\}) \\
&\quad , \mathit{flow\_phase}) \\
&\mathit{physTime}(v') = \mathit{physTime}(v) + \delta_{\mathit{switch}} \\
&\quad \text{with } \delta_{\mathit{switch}} \in \mathbb{R}_+ \\
&\kappa_{\mathit{Subj}_{\mathit{prog}}}(v'') = \kappa_{\mathit{Subj}_{\mathit{prog}}}(v) \oplus \\
&\quad \{(f, \mathit{integrate}) \mapsto (\mathit{string}_{\mathit{ProgState}}(\kappa_{\mathit{Subj}_{\mathit{prog}}}(v)(f, \mathit{integrate})), \\
&\quad\ \delta_{\mathit{ProgState}}(\kappa_{\mathit{Subj}_{\mathit{prog}}}(v)(f, \mathit{integrate})), \mathit{vis}_{\mathit{flow}}(c)(f) \Cap (\mathit{port}(c) \times \{0.1\})) \\
&\quad \mid f \in \mathit{flow}(c) \bullet \kappa_{Flow}(v)(f) = \mathit{nexttime}_1(c, v)\} \\
&\quad \cup\{(\mathit{md}, \mathit{poll}) \mapsto (\mathit{string}_{\mathit{ProgState}}(\kappa_{\mathit{Subj}_{\mathit{prog}}}(v)(\mathit{md}, \mathit{poll})), \\
&\quad\ \delta_{\mathit{ProgState}}(\kappa_{\mathit{Subj}_{\mathit{prog}}}(v)(\mathit{md}, \mathit{poll})), \mathit{vis}_{\mathit{ifm}}(c)(\mathit{md}) \Cap (\mathit{port}(c) \times \{0.1\})) \\
&\quad \mid \mathit{md} \in \mathit{ifm}(c) \bullet \mathit{nexttime}_{\mathit{ifm},\mathit{poll}}(\mathit{md}, (c, v)) \\
&\qquad = \mathit{nexttime}_1(c, v)\} \\
&\kappa_{\mathit{Subj}_{\mathit{abs}}}(v') = \kappa_{\mathit{Subj}_{\mathit{abs}}}(v) \oplus \\
&\quad \{\mathit{selector} \mapsto (\mathit{sel}_{\mathit{intState}}(\mathit{modelTime}(v), \mathit{selport}(c), s_i, \kappa_{Am}(v)), \infty, \lambda)\} \\
&\kappa_{Chan}(v') = \kappa_{Chan}(v) \oplus \\
&\quad \{\mathit{insert}_{\mathit{Port}}(p, (\mathit{data}, \{\mathit{tick}, p\}), \\
&\qquad (\mathit{subject}_{\mathit{var}}(c), \mathit{chan}_{\mathit{port}}(c), \mathit{subject}_{\mathit{port}}(c)), \\
&\qquad (\mathit{modelTime}(v), \sigma_{Var}(v), \kappa_{Chan}(v))) \\
&\quad \mid (\mathit{tick}, p) \mapsto \mathit{data} \\
&\qquad \in \mathit{sel}_{\mathit{chanEntry}}(\mathit{modelTime}(v), \mathit{selport}(c), s_i, \kappa_{Am}(v))\} \qquad \Box
\end{aligned}$$

Here, (1) flows and interface modules are set to the schedule. For interface modules, the fixed periods determine which of them are scheduled, whereas for flows, the flows' execution states denote this. (2) Additionally, visibility sets are generated from the static visibility sets of flows and interface modules, such that the calculation results will be published in the model future. The visibility sets are set as parameters to the flows' and interface modules' execution states. (3) The selector's new internal state is inserted. (4) Data is written to the corresponding channels.

The calculation of the scheduling time ticks for the polling of interface modules as well as the transmission of data to them is defined in the same fashion as the calculation of the scheduling time ticks of flows:

$$\begin{aligned}
&\mathit{nexttime}_{\mathit{ifm},\mathit{poll}} : \mathit{Ifm} \times S \to \mathbb{R}_0^+\\
&\qquad (md, (c, v)) \mapsto \mathit{nexttime}(\mathit{period}_{\mathit{ifm},\mathit{poll}}(c)(f), (c, v))\\
&\mathit{nexttime}_{\mathit{ifm},\mathit{tmit}} : \mathit{Ifm} \times S \to \mathbb{R}_0^+\\
&\qquad (md, (c, v)) \mapsto \mathit{nexttime}(\mathit{period}_{\mathit{ifm},\mathit{tmit}}(c)(f), (c, v))
\end{aligned}$$

Note that (1) one of transitions $\{t_{disc}, t_{flow}\}$ is non-deterministically chosen, if $\mathit{sel}_{\mathit{trans}}(s_i, \kappa_{Am}(v)) \neq \varnothing \wedge \mathit{sel}_{\mathit{flow}}(s_i, \kappa_{Am}(v))$, and that (2) a deadlock occurs, if $\mathit{sel}_{\mathit{trans}}(s_i, \kappa_{Am}(v)) = \varnothing \wedge \neg \mathit{sel}_{\mathit{flow}}(s_i, \kappa_{Am}(v))$. A well-formed model of the applied high-level formalism should avoid the second situation.

3.6 Program Subject Execution

This section defines the behavior of program subjects by transitions $t \in T$. Program subjects are similar to abstract subjects in that they have operations which can be executed, which takes some time before the execution terminates.

In contrast to abstract subjects, the operations of program subjects are given in a more concrete way – by *programs*. Such a program is a *sequence of statements*, therefore the execution of an operation is given by the sequence of executions of these statements.

Each statement execution then has its own duration, and its own effect on the state space on termination of the statement. Therefore, we define the progress of parallel statement executions by one *progress rule* which applies to any set of statements. According to abstract subjects, for $|\mathit{lwp}(c)| > 1$ time evolves *in common* for several operations and therefore for several statements. Here, timing constraints (see section 3.4.1) may be violated, too, leading to the setting of the fail flag.

The *effects* are defined per statement. We provide the corresponding *termination rule* in the following. The termination of a complete program is defined for the empty program string, which results when the last statement in a program is terminated, as well as for an explicit `return` statement, separately.

3.6.1 Progress of Statement Execution

Statements of program subjects are executed, as soon as a maximum subset from the schedule is allocated on LWPs, such that there is no idle LWP left or the schedule is empty. Then, statements take some time.

Let $(c, v) \in S$ with $\mathit{phase}(\mathit{sched}(v)) \in \{\mathit{flow\_phase}, \mathit{transition\_phase}\}$. Suppose that all active subjects have a positive remaining execution duration, i.e. $\forall s \in \operatorname{ran} \kappa_{LWP}(v) \bullet \delta_{\mathit{ProgState}}(\kappa_{\mathit{Subj}_{\mathit{prog}}}(v)(s)) \in \mathbb{R}_+$, and that a maximum subset of program subjects from the schedule is allocated on LWPs, such that there is no idle LWP left or the schedule is empty. That is, $\mathit{scheduler} \notin \operatorname{ran} \kappa_{LWP}(v) \vee \mathit{subj}_{\mathit{sched}}(\mathit{sched}(v)) = \varnothing$. Then some time can pass for the currently active statements to complete.

Rule 3.6.1 Let δ_{min} denote the minimal value of all remaining execution times of active statements captured by $\{\delta \in \mathbb{R}_+ \mid \exists (s, op) \in \operatorname{dom} \kappa_{\mathit{Subj}_{\mathit{prog}}}(v) \bullet \delta =$

$\delta_{ProgState}(\kappa_{Subj_{prog}}(v)(s, op))\}$. Then for all $\delta \leq \delta_{min}$, states $(c, v') \in Suc \subseteq S$ exist with the difference between v and v' of

$$\begin{aligned}
&\kappa_{Subj_{prog}}(v') = \kappa_{Subj_{prog}}(v) \oplus \{\, (s, op) \mapsto \\
&\qquad (string_{ProgState}(\kappa_{Subj_{prog}}(v)(s, op)), \delta_{ProgState}(\kappa_{Subj_{prog}}(v)(s, op)) - \delta \\
&\qquad\ vis_{ProgState}(\kappa_{Subj_{prog}}(v)(s, op))) \mid (s, op) \in \operatorname{dom} \kappa_{Subj_{prog}}(v) \\
&\qquad\quad \wedge \delta_{ProgState}(\kappa_{Subj_{prog}}(v)(s, op)) \in \mathbb{R}_+\} \\
&physTime(v') = physTime(v) + \delta
\end{aligned}$$ □

Similarly to the progress of abstract subject operations, the progress of statements may cause a timing failure. Therefore, the successor states are partitioned into $Suc_{sync} = Suc \cap S_{sync}$ and $Suc_{fail} = Suc \cap S_{fail}$.

Rule 3.6.2 Transitions $(c, v) \longrightarrow (c, v') \in T$ exist for states $(c, v') \in Suc_{sync}$, whereas for $(c, v') \in Suc_{fail}$ we have transitions $(c, v) \longrightarrow (c, v'') \in T$, such that the fail flag is set: $v' = v''$ holds, except for

$$fail(v'') = failed$$ □

3.6.2 Termination of Statement Execution

Suppose that in state $(c, v) \in S$ a program statement $stmt \in Stmt \setminus \{\texttt{return}\}$ is about to terminate, i.e. program subject s is active, $s \in \operatorname{ran} \kappa_{LWP}(v)$, and there is an operation $(s, op) \in \operatorname{dom} \kappa_{Subj_{prog}}(v)$ with

$$\kappa_{Subj_{prog}}(v)(s, op) = (\langle stmt \rangle \frown prg, 0, p_{vis})$$

Then the termination of *stmt* has an effect on the state space, given by the effect function

$$\begin{aligned}
&\epsilon : Program \times Param_{prog} \times Subject \times CONST_m \times VAR_{mread} \\
&\qquad \rightarrow Program \times VAR_{mwrite}
\end{aligned}$$

From a given program, a parameter, as well as the part of the state space that is readable from HL3 models, a remaining program along with the (potentially) modified state space portion with write access from the model results. The effect's resulting components are given by

$$\begin{aligned}
&\epsilon_{prg} : Program \times Param_{prog} \times Subject \times CONST_m \times VAR_{mread} \rightarrow Program \\
&\qquad (prg, param, s, c, v) \mapsto \pi_1\, \epsilon(prg, param, c, v) \\
&\epsilon_{var} : Program \times Param_{prog} \times Subject \times CONST_m \times VAR_{mread} \rightarrow \Sigma_{Var} \\
&\qquad (prg, param, s, c, v) \mapsto \pi_1\, (\pi_2\, \epsilon(prg, param, c, v)) \\
&\epsilon_{chan} : Program \times Param_{prog} \times Subject \times CONST_m \times VAR_{mread} \rightarrow \Sigma_{Chan} \\
&\qquad (prg, param, s, c, v) \mapsto \pi_2\, (\pi_2\, \epsilon(prg, param, c, v))
\end{aligned}$$

Rule 3.6.3 From this, a transition $(c, v) \longrightarrow (c, v') \in T$ is defined with $v = v'$, except for

$$\begin{aligned}
&\sigma_{Var}(v') = \epsilon_{var}(\langle stmt\rangle \frown prg, p_{vis}, s,\\
&\quad (subject_{var}(c), chan_{port}(c), subject_{port}(c)),\\
&\quad (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v)))\\
&\kappa_{Chan}(v') = \epsilon_{chan}(\langle stmt\rangle \frown prg, p_{vis}, s,\\
&\quad (subject_{var}(c), chan_{port}(c), subject_{port}(c)),\\
&\quad (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v)))\\
&\kappa_{Subj_{prog}}(v') = \kappa_{Subj_{prog}}(v) \oplus \{(s, op) \mapsto (\epsilon_{prg}(\langle stmt\rangle \frown prg, p_{vis}, s,\\
&\quad (subject_{var}(c), chan_{port}(c), subject_{port}(c)),\\
&\quad (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v))), \delta, p_{vis})\} \text{ with } \delta \in \mathbb{R}_+ \qquad \square
\end{aligned}$$

The transition updates the local variable valuations and the channel valuations according to the effect of the statement. Further, the remaining program string, which is also part of the effect, is inserted for (s, op).

For different kind of statements, the respective effects differ. In the following subsections, the definition of the effect function ϵ is given incrementally, per statement.

Write Access to Channels

The *writing* of data into a channel is defined by the statement `put`.

`put`. The effect of a statement $\texttt{put}(p_{write}, vis, data)$, where local variables $vis, x \in Var$ hold a visibility set $visset \in VisibilitySet$ and a data value $val \in Val$, and $p_{write} \in Port$ is a port which is accessible by subject s, i.e. $s \in subject_{port}(c)(p_{write})$, is defined by

$$\begin{aligned}
&\epsilon(\langle \texttt{put}(p_{write}, vis, x)\rangle \frown prg, param, s, c, v) = (prg, (\sigma_{Var}(v), \kappa_{Chan}(v) \oplus\\
&\quad \{insert_{Port}(p_{write}, (data_{Val}(\sigma_{Var}(v)(x)), \sigma_{Var}(v)(vis)), c, v)\}))
\end{aligned}$$

On the one hand, the data is written to the channel through the given port, with the attached visibilities. On the other hand, the `put` statement is consumed and removed from the program string.

Read Access to Channels

The *reading* of data from a channel is provided by different statements: (1) `get` retrieves the data value that is currently visible for a specific port. (2) `getTime` gets the publication time stamp for that data value. (3) `getPrevious` provides the second newest data value for a specific port. (4) `getPreviousTime` reads the publication time stamp for the second newest value.

`get`. A `get` statement is an assignment of the form x`:=get`(p_{read}) with a local variable $x \in Var$ of subject s, i.e. $subject_{var}(c)(x) = s$, and a port $p_{read} \in Port$ which is accessible by subject s, i.e. $s \in subject_{port}(c)(p_{read})$.

Then the current data value for the given port is retrieved and assigned to the local variable x, if x is of corresponding type, and the `get` statement is removed from the operation's program string:

$$\epsilon(\langle x\texttt{:=get}(p_{read})\rangle \frown prg, param, s, c, v) =$$
$$(prg, (\sigma_{Var}(v) \oplus \{x \mapsto val_{Data}(data_{Port,cur}(p_{read}, c, v))\}, \kappa_{Chan}(v)))$$

The variable must have a fitting type, that is, $val_{Data}(data_{Port,cur}(p_{read}, c, v)) \in type_{Var}(x)$.

`getTime`**.** Similarly to the retrieval of the current data, the associated publication time stamp can be read. This may be used while calculating integration steps, when the evolution of a value wrt. time is needed. Additionally, the previous value and time also have effect on the calculation; these are given in the succeeding paragraphs.

The effect of a statement $t\texttt{:=getTime}(p_{read})$ is that the time stamp is read and assigned to a (correctly typed) local variable $t \in Var$ of subject s, i.e. $type_{Var}(t) \supseteq ModelTime$ and $subject_{var}(c)(t) = s$. The time stamp is read for a port $p_{read} \in Port$ which is accessible by subject s, i.e. $s \in subject_{port}(c)(p_{read})$. Further, the `getTime` statement is removed from the operation's program string.

$$\epsilon(\langle t\texttt{:=getTime}(p_{read})\rangle \frown prg, param, s, c, v) =$$
$$(prg, (\sigma_{Var}(v) \oplus \{t \mapsto tick_{Port,cur}(p_{read}, c, v)\}, \kappa_{Chan}(v)))$$

`getPrevious`**.** A `getPrevious` statement is an assignment which is very similar to the `get` assignment; the only difference is, that instead of the newest value for the given port, the *second* newest one is retrieved.

For a statement $x\texttt{:=getPrevious}(p_{read})$ with a local variable $x \in Var$ of subject s, i.e. $subject_{var}(c)(x) = s$, and a port $p_{read} \in Port$ accessible by subject s, i.e. $s \in subject_{port}(c)(p_{read})$, the previous data value of the current one for the given port is retrieved and assigned to the local variable x, and the `getPrevious` statement is removed from the operation's program string:

$$\epsilon(\langle x\texttt{:=getPrevious}(p_{read})\rangle \frown prg, param, s, c, v) =$$
$$(prg, (\sigma_{Var}(v) \oplus \{x \mapsto val_{Data}(data_{Port,prev}(p_{read}, c, v))\}, \kappa_{Chan}(v)))$$

The variable must have a fitting type, that is, $val_{Data}(data_{Port,prev}(p_{read}, c, v)) \in type_{Var}(x)$.

`getPreviousTime`**.** The time stamp of the second newest value is accessed in the analogous way as the time stamp of the current value is.

The effect of a statement $t\texttt{:=getPreviousTime}(p_{read})$ is that the time stamp is read and assigned to the a local variable $t \in Var$ of correct type and of subject s, i.e. $type_{Var}(t) \supseteq ModelTime$ and $subject_{var}(c)(t) = s$. The time stamp is read for a port $p_{read} \in Port$ which is accessible by subject s, i.e. $s \in subject_{port}(c)(p_{read})$. Further, the `getPreviousTime` statement is removed from the operation's program string.

$$\epsilon(\langle t\texttt{:=getPreviousTime}(p_{read})\rangle \frown prg, param, s, c, v) =$$
$$(prg, (\sigma_{Var}(v) \oplus \{t \mapsto tick_{Port,prev}(p_{read}, c, v)\}, \kappa_{Chan}(v)))$$

Read Access to Model Time

The statement `getCurrentTime` retrieves the model time for use in the program:

`getCurrentTime`. The effect of a statement t:=`getCurrentTime`() is that the current model time is read and assigned to a local variable $t \in \mathit{Var}$, which is accessible for subject s, i.e. $\mathit{subject}_{var}(c)(t) = s$, supposed that $\mathit{type}_{\mathit{Var}}(t) \supseteq \mathit{ModelTime}$. Anyway, the `getCurrentTime` statement is removed from the operation's program string.

$$\begin{aligned}&\epsilon(\langle t\texttt{:=getCurrentTime()}\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = \\ &\qquad (\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{t \mapsto \mathit{modelTime}(v)\}, \kappa_{\mathit{Chan}}(v)))\end{aligned}$$

Read Access to the Visibility Set Parameter

The statement `getVisParam` retrieves the program's visibility set parameter:

`getVisParam`. The effect of a statement of the form *vis*:=`getVisParam`() is that the visibility set parameter is read and assigned to the local variable $\mathit{vis} \in \mathit{Var}$ of subject s, i.e. $\mathit{subject}_{var}(c)(\mathit{vis}) = s$. The type of *vis* must fit, and a parameter value must be available: $\mathit{type}_{\mathit{Var}}(\mathit{vis}) \supseteq \mathit{VisibilitySet}$ and $\mathit{param} \neq \lambda$. Further, the `getVisParam` statement is removed from the operation's program string:

$$\begin{aligned}&\epsilon(\langle \mathit{vis}\texttt{:=getVisParam()}\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = \\ &\qquad (\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{\mathit{vis} \mapsto \mathit{param}\}, \kappa_{\mathit{Chan}}(v)))\end{aligned}$$

The Operational Rules of Standard Commands

In this section, we briefly list the effects of the conventional statements of while-programs, as they are discussed in detail in [AO97]. We do *not* repeat the semantics of arithmetic and boolean expressions, but assume that they are known, along with the notations used in [AO97].

Assignment. An assignment statement evaluates an expression *exp* and assigns the value to a variable $x \in \mathit{Var}$ of subject s, i.e. $\mathit{subject}_{var}(c)(x) = s$. The statement is removed from the program string.

$$\begin{aligned}&\epsilon(\langle x\texttt{:=}\mathit{exp}\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = \\ &\qquad (\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{x \mapsto \sigma(\mathit{exp})\}, \kappa_{\mathit{Chan}}(v)))\end{aligned}$$

Conditional Statement. A conditional statement chooses one of two (sub-)programs sub_1 and sub_2, depending on the valuation of a boolean expression *bexp*. Then the effect of the statement is defined as

$$\begin{aligned}&\epsilon(\langle \texttt{if (}\mathit{bexp}\texttt{) \{}\mathit{sub}_1\texttt{\} else \{}\mathit{sub}_2\texttt{\}}\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = \\ &\qquad \begin{cases} (\mathit{sub}_1 \frown \mathit{prg}, (\sigma_{\mathit{Var}}(v), \kappa_{\mathit{Chan}}(v))) & \text{if } \sigma_{\mathit{Var}}(v) \models \mathit{bexp} \\ (\mathit{sub}_2 \frown \mathit{prg}, (\sigma_{\mathit{Var}}(v), \kappa_{\mathit{Chan}}(v))) & \text{if } \sigma_{\mathit{Var}}(v) \models \neg\mathit{bexp} \end{cases}\end{aligned}$$

While-Loop Statement. A while-loop repeatedly executes a (sub-)program *sub*, as long as a boolean expression *bexp* evaluates to *true*. Then the effect of the statement is given by

$$\epsilon(\langle \texttt{while (}\mathit{bexp}\texttt{) \{}\mathit{sub}\texttt{\}}\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) =$$
$$\begin{cases} (\mathit{sub} \frown \langle \texttt{while (}\mathit{bexp}\texttt{) \{}\mathit{sub}\texttt{\}}\rangle \frown \mathit{prg}, & \\ \quad (\sigma_{\mathit{Var}}(v), \kappa_{\mathit{Chan}}(v))) & \text{if } \sigma_{\mathit{Var}}(v) \models \mathit{bexp} \\ (\mathit{prg}, (\sigma_{\mathit{Var}}(v), \kappa_{\mathit{Chan}}(v))) & \text{if } \sigma_{\mathit{Var}}(v) \models \neg\mathit{bexp} \end{cases}$$

Set Operations

We introduce special statements to support the handling of set values within programs. These are the adding and retrieving of entries, as well as the clearing of sets and the reading of the set size.

`clear`. For a variable $s \in \mathit{Var}$ that can have the empty set as current value, i.e. with $\varnothing \in \mathit{type}_{\mathit{Var}}(s)$, the operation `clear` assigns it:

$$\epsilon(\langle \texttt{clear}(s)\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) =$$
$$(\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{s \mapsto \varnothing\}, \kappa_{\mathit{Chan}}(v)))$$

`addEntry`. For a variable $s \in \mathit{Var}$ with a current set value, i.e. $\exists\, S \bullet \sigma_{\mathit{Var}}(v)(s) \in \mathcal{P}(S)$, the operation `addEntry` adds the value given by the variable $x \in \mathit{Var}$:

$$\epsilon(\langle \texttt{addEntry}(s, x)\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) =$$
$$(\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{s \mapsto \sigma_{\mathit{Var}}(v)(s) \cup \{\sigma_{\mathit{Var}}(v)(x)\}\}, \kappa_{\mathit{Chan}}(v)))$$

This is only defined, if the new set also fits to the variable's type, that means if $\sigma_{\mathit{Var}}(v)(s) \cup \{\sigma_{\mathit{Var}}(v)(x)\} \in \mathit{type}_{\mathit{Var}}(s)$.

`getEntry`. The reading of entries from a set value is defined by the statement `getEntry`, such that one element of the set of the variable $s \in \mathit{Var}$ is assigned to variable $x \in \mathit{Var}$.

For this, a mapping $\mathrm{anyseq}_{\mathrm{SET}} : \mathcal{P}(\mathit{SET}) \mapsto \mathrm{seq}\, \mathit{SET}$ is supposed to define an arbitrary sequence for all elements of a given finite set $\mathit{set} \subseteq \mathit{SET}$, i.e. $\forall\, \mathit{set} \subseteq \mathit{SET} \bullet |\mathit{set}| \in \mathbb{N}_0 \Rightarrow \mathrm{ran}(\mathrm{anyseq}_{\mathrm{SET}}(\mathit{set})) = \mathit{set} \wedge |\mathrm{anyseq}_{\mathrm{SET}}(\mathit{set})| = |\mathit{set}|$.

Then, the variable $i \in \mathit{Var}$ defines an index of this sequence, which identifies the element to be read:

$$\epsilon(\langle x\texttt{:=getEntry}(s, i)\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = (\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus$$
$$\{x \mapsto \mathrm{anyseq}_{\sigma_{\mathrm{Var}}(\mathrm{v})(\mathrm{s})}(\sigma_{\mathit{Var}}(v)(s))(\sigma_{\mathit{Var}}(v)(i) + 1)\}, \kappa_{\mathit{Chan}}(v)))$$

The variable i must hold an index that maps to a set entry: $\sigma_{\mathit{Var}}(v)(i) \in \mathbb{N}_0 \wedge |\sigma_{\mathit{Var}}(v)(s)| > \sigma_{\mathit{Var}}(v)(i)$. Finally, the resulting set must fit to the type of x: $\mathrm{anyseq}_{\sigma_{\mathrm{Var}}(\mathrm{v})(\mathrm{s})}(\sigma_{\mathit{Var}}(v)(s))(\sigma_{\mathit{Var}}(v)(i) + 1) \in \mathit{type}_{\mathit{Var}}(x)$

`size`. The statement `size` accesses the size of a set value that is held by a variable $s \in \mathit{Var}$, and assigns it to a variable $x \in \mathit{Var}$, provided that $\mathit{type}_{\mathit{Var}}(x) \supseteq \mathbb{N}_0$:

$$\epsilon(\langle x\texttt{:=size}(s)\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) =$$
$$(\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{x \mapsto |\sigma_{\mathit{Var}}(v)(s)|\}, \kappa_{\mathit{Chan}}(v)))$$

Pair Operations

In order to read and modify pairs of values within a program, the operations `left`, `right`, `setLeft`, and `setRight` are defined:

`left`. Read access to the first projection of a pair of values is provided by the statement `left` for a variable $p \in \mathit{Var}$ with $\exists (l,r) \in (L \times R) \bullet \sigma_{\mathit{Var}}(v)(p) = (l,r)$. The value is then stored in a variable $x \in \mathit{Var}$ of corresponding type, i.e. with $l \in \mathit{type}_{\mathit{Var}}(x)$:

$$\begin{aligned}&\epsilon(\langle x\texttt{:=left}(p)\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = \\ &\quad (\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{x \mapsto \pi_1\, \sigma_{\mathit{Var}}(v)(p)\}, \kappa_{\mathit{Chan}}(v)))\end{aligned}$$

`right`. Read access to the second projection of a pair of values is provided analogously to `left`, with the corresponding assumptions.

$$\begin{aligned}&\epsilon(\langle x\texttt{:=right}(p)\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = \\ &\quad (\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{x \mapsto \pi_2\, \sigma_{\mathit{Var}}(v)(p)\}, \kappa_{\mathit{Chan}}(v)))\end{aligned}$$

`setLeft`. Write access to the first projection of a pair of values is provided by the statement `setLeft` for variables $p, x \in \mathit{Var}$:

$$\begin{aligned}&\epsilon(\langle \texttt{setLeft}(p,x)\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = \\ &\quad (\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{p \mapsto (\sigma_{\mathit{Var}}(v)(x), \pi_2\, \sigma_{\mathit{Var}}(v)(p))\}, \kappa_{\mathit{Chan}}(v)))\end{aligned}$$

The previous value of p must be a pair already: $\exists (l,r) \in (L \times R) \bullet \sigma_{\mathit{Var}}(v)(p) = (l,r)$. The new value must fit to the type of p again: $(\sigma_{\mathit{Var}}(v)(x), \pi_2\, \sigma_{\mathit{Var}}(v)(p)) \in \mathit{type}_{\mathit{Var}}(p)$.

`setRight`. Write access to the second projection of a pair of values is provided by the statement `setRight` for variables $p, x \in \mathit{Var}$:

$$\begin{aligned}&\epsilon(\langle \texttt{setRight}(p,x)\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = \\ &\quad (\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{p \mapsto (\pi_1\, \sigma_{\mathit{Var}}(v)(p), \sigma_{\mathit{Var}}(v)(x))\}, \kappa_{\mathit{Chan}}(v)))\end{aligned}$$

The previous value of p must be a pair already, and the new value must fit to the type of p again: $\exists (l,r) \in (L \times R) \bullet \sigma_{\mathit{Var}}(v)(p) = (l,r)$ and $(\pi_1\, \sigma_{\mathit{Var}}(v)(p), \sigma_{\mathit{Var}}(v)(x)) \in \mathit{type}_{\mathit{Var}}(p)$.

Non-Determinism

The statement `random` provides non-determinism by assigning a random natural number (or 0) to a local variable.

`random`. A `random` statement is an assignment $x\texttt{:=random}()$ of a local variable $x \in \mathit{Var}$, which is accessible for subject s, i.e. $\mathit{subject}_{\mathit{var}}(c)(x) = s$. The assigned value is an arbitrary natural number or 0. The `random` statement is consumed.

$$\begin{aligned}&\exists\, \mathit{val} \in \mathbb{N}_0 \bullet \epsilon(\langle x\texttt{:=random()}\rangle \frown \mathit{prg}, \mathit{param}, s, c, v) = \\ &\quad (\mathit{prg}, (\sigma_{\mathit{Var}}(v) \oplus \{x \mapsto \mathit{val}\}, \kappa_{\mathit{Chan}}(v)))\end{aligned}$$

The variable x must be of corresponding type: $\mathit{val} \in \mathit{type}_{\mathit{Var}}(x)$

3.6.3 Program Termination

The termination of a complete program, and therefore of the associated operation, takes place when either the program string is empty, or the explicit `return` is reached. Since the termination itself may also take some (small amount of) time, it is treated in a similar way as statements are.

Suppose that in state $(c, v) \in S$ an operation $(s, op) \in \operatorname{dom} \kappa_{Subj_{prog}}(v)$ with an empty program string or a program string beginning with `return` has an elapsed execution time:

$$\begin{aligned}&\kappa_{Subj_{prog}}(v)(s, op) = (prg, 0, p_{vis})\\&\text{with } prg = \langle\rangle \vee head(prg) = \texttt{return}\end{aligned}$$

As a result, the remaining program string is empty:

$$\begin{aligned}&\epsilon(prg, param, s, c, v) =\\&\qquad(\langle\rangle, (\sigma_{Var}(v), \kappa_{Chan}(v)))\end{aligned}$$

Further, the corresponding LWP is released, i.e. it is controlled by the scheduler now. Additionally, the execution time is unset, and the visibility set parameter is removed.

Rule 3.6.4 Therefore, we have a transition $(c, v) \longrightarrow (c, v') \in T$ with $v = v'$, except for

$$\begin{aligned}&\kappa_{Subj_{prog}}(v') = \kappa_{Subj_{prog}}(v) \oplus \{(s, op) \mapsto (\epsilon_{prg}(prg, p_{vis}, s,\\&\qquad(subject_{var}(c), chan_{port}(c), subject_{port}(c)),\\&\qquad(modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v))), \infty, \lambda)\}\\&\kappa_{LWP}(v') = \kappa_{LWP}(v) \oplus \{p \mapsto scheduler \mid \kappa_{LWP}(v)(p) = s\} \qquad \square\end{aligned}$$

3.7 Properties of the HL3 Framework

In this section, we discuss observations on the behaviour of the HL3 framework.

3.7.1 Timeliness of the HL3 Execution

The following invariant holds for all (valid) states $(c, v) \in S$ which are reachable in a HL3 execution. The first component of the model time is never ahead of the integral part of a global physical time value:

$$\lfloor physTime(v) \rfloor \geq t_0(modelTime(v))$$

This is proven by an inductive argument: In the initial state v_0 we have $modelTime(v_0) = 0.0$ and $physTime(v_0) = 0$. For all rules transferring a state (c, v) to (c, v') except for Rules 3.4.1 and 3.4.2, we have that $modelTime(v') = modelTime(v)$ while $physTime(v') \geq physTime(v)$, thus preserving the invariant. In case of Rule 3.4.1, still $t_0(modelTime(v')) = t_0(modelTime(v))$ holds. In case of Rule 3.4.2, we indeed have a new value for $t_0(modelTime(v'))$, but since $modelTime(v') = \lfloor physTime(v) \rfloor.0$, we conclude immediately $\lfloor physTime(v') \rfloor \geq t_0(modelTime(v'))$.

For all valid states $(c, v) \in S_{sync}$ we also have that

$$t_0(modelTime(v)) \geq \lfloor physTime(v) \rfloor - \delta_{period}(c)$$

which is immediately implied by condition *chktsync*.

3.7.2 Scheduler Guarantees

In order to get a faithful execution of a high-level model, the underlying system must not corrupt this execution by either introducing delays or causing blocking of the system not related to the model itself.

No unbounded delay None of the rules may allow a process to occupy a CPU for an unbounded time, i.e., we have to show that any rule application requires at most some maximal duration δ_{max}.
This is guaranteed by the rules of Section 3.4.2 for $\delta_{switch} \leq \delta_{max}$ where δ_{switch} is some bounded duration which depends on the implementation . A similar argument holds for the rules of Section 3.4.3 except for Rule 3.4.8: while execution time of these rules is included into the duration of the respective execution subjects, it is still bounded as long as $\delta_{alloc} \leq \delta_{max}$, again with δ_{alloc} depending on implementation details. For Rule 3.4.8, we observe that the waiting time δ is always bounded by the system period.

The progress rule of Section 3.5.1 has a duration δ bounded by some δ_{min} calculated from the (again bounded) durations for the calculation of abstract subjects.

The duration of the termination rules for abstract machines in Section 3.5.2 is 0, as termination and release of an LWP is seen as last operation of that abstract machine execution. Thus, these rules can be applied without any delay. The duration of the termination rules for the *selector* is again bounded by some δ_{switch}, a value which depends on the implementation.

No Zeno-execution on scheduler rules The HL3 rules define the simulation of a hybrid system. But as these are defined to incorporate the behavior of the underlying computer system, certain observations on hybrid systems do not carry over to HL3. A phenomenon of hybrid automata is the existence of *Zeno-executions*, where infinitely many transitions are followed in a finite time interval. While there are various reasons for a model to allow Zeno-executions, physical systems are not Zeno [JLSE99].

The HL3 rules do not allow an infinite number of transitions within a fixed duration. This can be seen as follows: Most of the rules in Section 3.4 and 3.5 already have some explicit $\delta > 0$ for execution itself.For those rules without an explicit duration we have to analyze if it is possible to apply them in an arbitrary number without being forced to apply some rule with a minimal duration after an upper bound n.

This is true for Rule 3.4.6, as it requires $sset(v)$ to be non-empty and $\kappa_{LWP}(v)(p) = scheduler$. The rule removes one element of this list *sset* and assigns the associated program subject to that LWP. Once no more elements are in *sset* or no free LWP is available, the HL3 model has to apply a rule which executes a program statement of a transition - which takes time. The same argument holds for Rule 3.4.7.

For the rules of Section 3.5.2 on termination of abstract machines' operations, we distinguish the cases whether some abstract subjects are still listed to be scheduled, i.e., whether $sset(v) \neq \varnothing$. In this case the next applied transition is either the progress rule (taking time), or Rule 3.4.7, which also leads to a situation that requires time to elapse. In case $sset(v) = \varnothing$, the next applied transition is either the progress rule (taking time), or some rule of Section 3.4.2, as all abstract machines have finished execution in the respective phase.

If the high-level model transformed to HL3 exhibits some Zeno behaviour related to model time, the resulting HL3 execution will reach a *fail* state. Assume a model with a Zeno-execution. In the HL3 model then a transition phase exists, where an infinite number of transition are to be executed. As the execution of the transformed transitions do take time, the duration of that particular transition phase exceeds $\delta_{period}(c)$, leading to an application of Rule 3.6.2 which sets the *fail* flag to *failed*. Thus, occurence of a Zeno-execution can be detected. HL3 currently does not offer to continue the simulation using some extension model for Zeno-executions as suggested in [JLSE99].

Blocking analysis An execution of a HL3 model may end in a *deadlock*, i.e., a safe state where no rule can be applied, when the selector component observes that no transition of the high-level model is enabled and no time-passing execution of flow is allowed. In this case, the HL3 model correctly reflects a deadlock situation of the high-level model.

For all other safe states $(c, v) \in S_{sync}$ which result from the application of a HL3 rule, there is no deadlock possible, as some rule can continue execution by construction of the phase cycle model.

An execution of a HL3 model may reach a state (c, v) with $\mathit{fail}(v) = \mathit{failed}$.The $\mathit{fail}(v)$ flag can be set to *failed* by Rule 3.5.6 if there is no initial valuation of the model. Again, this accurately reflects a failure of the high-level model.

A HL3 execution can end in a *fail* state by exceeding the allowed duration of its respective execution phase, postulated by Rules 3.5.2 and 3.6.2. In this case, the implementation may exhibit a failure of the simulation for an admissible model. Among the possible causes for this we have (1) the actual scheduling algorithm, distributing program subjects in an unfavorable fashion. While the execution of the high-level operations is possible within a certain duration, the associated sequence of operations of program subjects exceeds this limit. (2) the HL3 scheduler accumulates small delays associated with scheduler-related transitions, which again can make the difference between a successful execution and a failed one. (3) the actual execution duration for activities of subjects allows only for a limited number of operation, insufficient to complete an update or flow phase in time. (4) insufficient number of CPUs preventing the scheduler to process all operations within a certain phase in time.

3.7.3 Program Execution

As long as program subjects terminate regulary, they do not harm the HL3 execution. Any change of the program state is controlled by the HL3 environment.

Termination Provided the involved program segments do not deadlock, have no infinite loop, and no run-time error occurs, execution of a program statement within the HL3framework terminates in bounded time.

This is a consequence of the following observations:
• The duration a program requires for termination (i.e., returning CPU control, Rule 3.6.4) is included in the excecution time of its last statement, which is less than some duration $\delta_{max} \in \mathbb{R}_+$.
• All atomic statements (assignments, guard evaluation, channel access, etc.) take some duration less than δ_{max}, and the remaining program text has less statements than the text before the transition.
• Only the **while** statement extends the program text. But since the number n_{loop} of loops is finite, its execution in terms of program statements is equal to the execution of its program body as often as the number of loops n_{loop}. As the excecution duration of the program body is bounded, so is the excecution duration of the entire **while** statement.

Note that in case of a program blocking, entering an infinite loop, or encountering a run-time error, the HL3 model will eventually reach a *fail* state, as the respective execution phase exceeds the limit of that its frame.

HL3 Control A program subject can change its program to be executed (as string) only if this subject occupies a CPU in between.

Given a program subject $P \in Subj_{prog}$, we consider an execution $(c, v) \to^* (c, v')$ such that $string_{ProgState}(\kappa_{Subj_{prog}}(v)(P, op)) \neq string_{ProgState}(\kappa_{Subj_{prog}}(v')(P, op))$. Note that only the Rule 3.6.3 actually changes the program string, hence v'', v''' exist such that $(c, v) \to^* (c, v'')$, $(c, v''') \to^* (c, v')$ and the transition $(c, v'') \to (c, v''')$ has been made by an application of Rule 3.6.3. A prerequisite for application of that rule is the fact that $P \in \operatorname{ran} \kappa_{LWP}(v'')$, i.e., P is running on some CPU.

3.7.4 Data Consistency

A $get()$ operation at model time $t_0.t_1$ never reads data to be published at $t'_0.t'_1$ with $t'_0 > t_0$ or $t'_0 = t_0 \wedge t'_1 > t_1$. This is discussed already in Section 3.3.

3.7.5 Timeliness for Flows and Transitions

A successful execution complies to the constraints of flows which are enabled, the HL3 model immediately enforces the timely calculation for continous evaluations that are permanently enabled.

A flow which is always enabled, will eventually occupy a CPU. Assume a flow f which is enabled for all states - as a simple example this could represent a clock. In terms of the HL3 model, this means that from the current state (c, v) on, this flow marked with a time value indicating when this flow is due to be executed, i.e. we have $nexttime_{flow}(f, (c, v)) \neq \infty$. Note that this value is set only as a result of the update phase in Rule 3.5.4 and that the value does not change until the intended scheduling time is reached (for all time instances t in the interval $[n \cdot k \cdot \delta_{period}(c), (n + 1) \cdot k \cdot \delta_{period}(c)[$ the same value of $nexttime_{flow}(f, (c, v))$ is calculated). With $t_{sched} = nexttime_{flow}(f, (c, v))$, we get f selected for execution by Rule 3.5.8 whenever this rule is invoked in the interval $[t_{sched} - \delta_{period}(c), t_{sched}[$, as for this interval $\kappa_{Flow}(v)(f) = t_{sched}$ holds. It remains to show that Rule 3.5.8 is actually applied within the interval $[t_{sched} - \delta_{period}(c), t_{sched}[$. Note that we consider only successful executions, i.e., executions that do not reach a *fail* state. For these admissible runs, we discussed in Section 3.4.1 that the predicate *chkttrig* holds for all states, ensuring that the flow phase is active after t_{sched}. Since the flow phase can only be invoked by an application of Rule 3.5.8, this moment of application has to be in the interval above.

Enforced transitions are taken without delay in model time. A typical technique to enforce taking a certain transition at a particular time instant within the model disables the option to let time pass within the model at this moment unless the transition is taken. This enforcement of transitions is mapped by the HL3 model in Rules 3.5.7 and 3.5.8, where $\neg sel_{flow}(s_i, \kappa_{Am}(v))$ disables Rule 3.5.8, while Rule 3.5.7 ensures that some transition is taken. Rule 3.5.7 may be invoked several times before the enforced transition is actually selected, yet there is no change of model time $t_0(modelTime(v))$, as this is changed only at the end of a flow phase (Rule 3.4.2), which can only be reached via an invocation of Rule 3.5.8.

Chapter 4

HybridUML Mathematical Meta-Model

This chapter defines a syntax for HybridUML specifications – the Mathematical Meta-Model, which is the HybridUML meta-model in terms of UML. In contrast to the illustration given in section **??**, the syntax is defined non-graphically, but formally in the mathematical sense. The main reason for this is that the focus of this thesis is on the transformation Φ from HybridUML specifications into an executable system, which is defined formally itself. The syntax is therefore given in a form that is appropriate as input for the transformation.

Further, the separation of the meta-model from its graphical representation is the usual UML approach. As a consequence, the transformation semantics of HybridUML models is independent from the graphical representation.

The correlation between the Mathematical Meta-Model and the graphical notation of section **??** is straightforward, therefore it is not given explicitly in this thesis.

The initial HybridUML presentation [BBHP03] applies the standard UML 2.0 procedure to define a specialization of UML – the *profile mechanism.* HybridUML is defined as a profile, i.e. the UML meta-model is modified by the application of stereotypes. The profile definition itself is given by means of the Meta Object Facility (MOF) [OMG06], which is the meta-model language of UML.[1]

The profile consists of a mixture of graphical notations, OCL expressions, and natural language. This is the recommended approach to adapt UML, but the resulting profile is not mathematically formal. An additional drawback is that the resulting meta-model is at least as large as the original UML meta-model, since all modifications are realized by the *addition* of stereotypes (which include constraints, textual descriptions, etc.).

Therefore, we define the Mathematical Meta-Model, i.e. the HybridUML meta-model explicitly, using mathematical definitions. The Mathematical Meta-Model is intended to be equivalent to the profile definition to a large extent. However, in addition to small technical differences, the concept of event-based communication is integrated.

[1]The MOF features used for the UML meta-model constitute roughly a subset of UML itself.

In order to make it more comprehensible, the Mathematical Meta-Model is augmented by descriptions of the intended purpose of the respective entities, i.e. *intuitive semantics* are given.

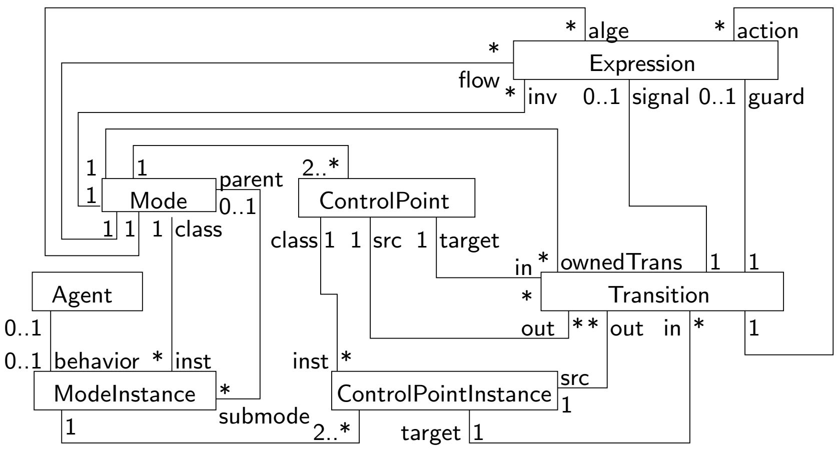


Figure 4.1: HL3 Part of the Mathematical Meta-Model that provides the behavioral specification, illustrated as UML class diagram.

HybridUML Specifications. A HybridUML specification is a tuple

$$\begin{aligned} spec \;=\; & (A, AI, ai_{tl}, V, \sigma_V, S, P_V, PI_V, C_V, P_S, PI_S, \\ & C_S, DT, M, MI, CP, CPI, T, Exp) \end{aligned}$$

with a set of agents A, a set of agent instances AI, a dedicated top-level agent instance $ai_{tl} \in AI$, a set of properties V, a set of property values σ_V, a set of signals S, a set of variable ports P_V, a set of variable port instances PI_V, a set of variable connectors C_V, a set of signal ports P_S, a set of signal port instances PI_S, a set of signal connectors C_S, a set of datatypes DT, a set of modes M, a set of mode instances MI, a set of control points CP, a set of control point instances CPI, a set of transitions T, and a set of expressions Exp.

4.1 Structural Specification

In this section, the part of HybridUML specifications that defines the *structure* of the system is given. This contains everything that is represented in *class diagrams* and *composite structure diagrams*, as it was illustrated in section **??**.

Agent. Agents are the main structural building block of a HybridUML specification. An Agent is a *class* in the usual sense – it represents a set of similar objects, which contain data as well as (optional) behavior. These objects make up an object tree, in that each object either (1) forms a part of the *structural hierarchy* of the system, or (2) provides the context for a state machine which defines sequential *behavior*. The latter are the leafs of the object tree. Technically, an agent is either a *basic agent* or a *composite agent*. A basic agent has exactly one state machine called *mode* that defines its behavior, but has no parts (i.e. contained agent instances). A composite agent contains parts, but has no own behavior:

$$\begin{aligned}
&\mathit{behavior}_A : A \rightarrow \mathcal{P}(MI)\\
&\mathit{part}_A : A \rightarrow \mathcal{P}(AI)\\
&\forall\, a \in A \bullet\\
&\qquad \mathit{part}_A(a) = \varnothing \Leftrightarrow |\mathit{behavior}_A(a)| = 1\\
&\qquad \wedge \mathit{part}_A(a) \neq \varnothing \Leftrightarrow \mathit{behavior}_A(a) = \varnothing
\end{aligned}$$

Agents contain variables, parameters, and signals:

$$\begin{aligned}
&\mathit{var}_A : A \rightarrow \mathcal{P}(V)\\
&\mathit{param}_A : A \rightarrow \mathcal{P}(V)\\
&\mathit{sig}_A : A \rightarrow \mathcal{P}(S)
\end{aligned}$$

Variables are the objects' variables in the usual sense.[2]

Parameters are special private and read-only variables with a multiplicity of one. They are distinct from other variables in that they are available during the construction of the system's static structure, rather than afterwards.

$$\begin{aligned}
&\forall\, a \in A \bullet \mathit{param}_A(a) \subseteq \mathit{var}_A(a)\\
&\forall\, a \in A, v \in V \bullet v \in \mathit{param}_A(a) \Rightarrow \mathit{vis}_V(v) = \mathit{priv}\\
&\forall\, a \in A, v \in V \bullet v \in \mathit{param}_A(a) \Rightarrow \mathit{acc}_V(v) = \mathit{ro}\\
&\forall\, a \in A, v \in V \bullet v \in \mathit{param}_A(a) \Rightarrow \mathit{mult}_V(v) = 1
\end{aligned}$$

There is a special "ID" parameter for each agent:

$$\forall\, a \in A \bullet v_{id,a} \in \mathit{param}_A(a)$$

Signals are incidents without time duration.[3] Basic agents can synchronize by sending or receiving signals, respectively. The sending (or raising) of a signal is non-blocking: Raised signals are multicasted immediately to all agents that potentially receive the signal. The current internal state of each basic agent determines, whether the signal is actually received or silently lost.

Each variable, parameter, and signal is exclusively contained by an agent, i.e.:

$$\begin{aligned}
&\forall\, v \in V, a_1, a_2 \in A \bullet\\
&\qquad (v \in \mathit{var}_A(a_1) \wedge v \in \mathit{var}_A(a_2) \Rightarrow a_1 = a_2)\\
&\forall\, p \in V, a_1, a_2 \in A \bullet\\
&\qquad (p \in \mathit{param}_A(a_1) \wedge p \in \mathit{param}_A(a_2) \Rightarrow a_1 = a_2)\\
&\forall\, s \in S, a_1, a_2 \in A \bullet\\
&\qquad (s \in \mathit{sig}_A(a_1) \wedge s \in \mathit{sig}_A(a_2) \Rightarrow a_1 = a_2)
\end{aligned}$$

[2]There are different terms which are commonly used for *variables* within classes, at least *attribute*, *property*, or *member*. We will use them synonymously in this thesis.

[3]A more common term is *event*, but for UML conformity, we prefer *signal*.

An agent may have ports for its variables and signals, in order to share them with its environment as well as with its contained subagent instances:

$$port_{A,Var} : A \rightarrow \mathcal{P}(P_V)$$
$$port_{A,Sig} : A \rightarrow \mathcal{P}(P_S)$$

Each port represents a variable or signal of the agent:

$$\forall\, p \in P_V, a \in A \bullet$$
$$p \in port_{A,Var}(a) \Rightarrow var_{P_V}(p) \in var_A(a)$$
$$\forall\, p \in P_S, a \in A \bullet$$
$$p \in port_{A,Sig}(a) \Rightarrow sig_{P_S}(p) \in sig_A(a)$$

As with variables and signals, the ports are exclusively contained by one agent each:

$$\forall\, p \in P_V, a_1, a_2 \in A \bullet$$
$$(p \in port_{A,Var}(a_1) \wedge p \in port_{A,Var}(a_2) \Rightarrow a_1 = a_2)$$
$$\forall\, p \in P_S, a_1, a_2 \in A \bullet$$
$$(p \in port_{A,Sig}(a_1) \wedge p \in port_{A,Sig}(a_2) \Rightarrow a_1 = a_2)$$

If an agent is composite, it can have variable and signal connectors. Connectors connect ports and port instances, in order to define shared variables and shared signals inside the agent.

$$conn_{A,Var} : A \rightarrow \mathcal{P}(C_V)$$
$$conn_{A,Sig} : A \rightarrow \mathcal{P}(C_S)$$
$$\forall\, a \in A \bullet$$
$$part_A(a) = \varnothing \Rightarrow conn_{A,Var}(a) = \varnothing \wedge conn_{A,Sig}(a) = \varnothing$$

Each agent has a set of initial state constraints. These constraints must hold for each of the agent's objects after construction of the static structure of the system. Otherwise, there is no defined execution of the complete system.

$$initState_A : A \rightarrow \mathcal{P}(Exp)$$

AgentInstance. An agent instance represents a subset of the objects of a dedicated agent. Therefore, agent instances are mapped to the corresponding agent:

$$agent_{AI} : AI \rightarrow A$$

A multiplicity expression defines the number of objects that are represented:

$$mult_{AI} : AI \rightarrow Exp$$

The multiplicity has to be at least one. If the agent instance is defined within the context of a parent agent (which is almost always the case), then the number of objects is further multiplied by the number of duplicates of the parent agent.

For connecting the agent instance's properties or signals with its environment, the corresponding variable or signal ports of the corresponding agent are instantiated for the agent instance:

$$\begin{aligned}&portIns_{AI,Var} : AI \rightarrow \mathcal{P}(PI_V)\\&portIns_{AI,Sig} : AI \rightarrow \mathcal{P}(PI_S)\end{aligned}$$

Of course, no other ports can be instantiated, i.e.:

$$\begin{aligned}&\forall\, pi \in PI_V, ai \in AI \bullet\\&\qquad pi \in portIns_{AI,Var} \Rightarrow port_{PI_V}(pi) \in port_{A,Var}(agent_{AI}(ai))\\&\forall\, pi \in PI_S, ai \in AI \bullet\\&\qquad pi \in portIns_{AI,Sig} \Rightarrow port_{PI_S}(pi) \in port_{A,Sig}(agent_{AI}(ai))\end{aligned}$$

Each port instance is connected to exactly one agent instance:

$$\begin{aligned}&\forall\, pi \in PI_V, ai_1, ai_2 \in AI \bullet\\&\qquad (pi \in portIns_{AI,Var}(ai_1) \wedge pi \in portIns_{AI,Var}(ai_2) \Rightarrow ai_1 = ai_2)\\&\forall\, pi \in PI_S, ai_1, ai_2 \in AI \bullet\\&\qquad (pi \in portIns_{AI,Sig}(ai_1) \wedge pi \in portIns_{AI,Sig}(ai_2) \Rightarrow ai_1 = ai_2)\end{aligned}$$

Each parameter of the agent gets exactly one value specification for the agent instance. This is determined and assigned during the creation of the static structure of the system.

$$\begin{aligned}&paramVal_{AI} : AI \rightarrow \mathcal{P}(\sigma_V)\\&\forall\, ai \in AI \bullet \forall\, p \in param_A(agent_{AI}(ai)) \bullet\\&\qquad |\{v \in paramVal_{AI}(ai) \mid var_{\sigma_V}(v) = p\}| = 1\end{aligned}$$

The initial state space can be constrained on agent instance level in the same way as on agent level:

$$initState_{AI} : AI \rightarrow \mathcal{P}(Exp)$$

The conjunction of all initial state constraints from the agent instance as well as from the corresponding agent defines the initial state space for the agent instance's objects.

Property. Properties (alias variables[2]) of agents are typed:

$$type_V : V \rightarrow DT$$

Each variable has a multiplicity, a visibility, and an access policy:

$$\begin{aligned}&mult_V : V \rightarrow Exp\\&vis_V : V \rightarrow \{priv, pub\}\\&acc_V : V \rightarrow \{ro, rw\}\end{aligned}$$

Every variable is implicitly an array, such that there are $n \geq 1$ copies of it. This is defined by its multiplicity. The visibility is either *public*, i.e. the variable can be connected to variables of other agents, or *private* to prevent this. The access policy defines whether the variable is writable or constant.

DataType. Different kinds of datatypes are available for HybridUML – (1) primitive datatypes, (2) structured datatypes and (3) enumeration types:

$$\begin{aligned}
&\mathit{kind}_{DT} : DT \to \{\mathit{prim}, \mathit{struc}, \mathit{enum}\}\\
&DT_{\mathit{prim}} = \{t \in DT \mid \mathit{kind}_{DT}(t) = \mathit{prim}\}\\
&DT_{\mathit{struc}} = \{t \in DT \mid \mathit{kind}_{DT}(t) = \mathit{struc}\}\\
&DT_{\mathit{enum}} = \{t \in DT \mid \mathit{kind}_{DT}(t) = \mathit{enum}\}
\end{aligned}$$

The primitive types are predefined, these are (1) *boolean*, representing values $b \in \mathbb{B}$, (2) *integer*, representing values $i \in \mathbb{Z}$, and (3) *real*, which represents values $r \in \mathbb{R}$. Additionally, the type (4) *analog real* is used to distinguish variables which can be modified continuously from discrete real-valued variables.

$$\begin{aligned}
&\forall\, t \in DT \bullet\\
&\qquad \mathit{kind}_{DT}(t) = \mathit{prim} \Rightarrow t \in \{\mathit{bool}, \mathit{int}, \mathit{real}, \mathit{anaReal}\}
\end{aligned}$$

Enumeration types are defined within the HybridUML model. They have exclusive literals, i.e. each enumeration literal is mapped to exactly one enumeration type. There are no other data types with enumeration literals.

$$\begin{aligned}
&\mathit{dt}_L : L \to DT\\
&\mathit{lit}_{DT} : DT \to \mathcal{P}(L)\\
&\qquad t \mapsto \{l \in L \mid \mathit{dt}_L(l) = t\}\\
&\forall\, t \in DT \bullet\\
&\qquad \mathit{kind}_{DT}(t) \neq \mathit{enum} \Rightarrow \mathit{lit}_{DT}(t) = \varnothing
\end{aligned}$$

Structured data types are also defined within the model. Each structured data type contains a list of properties:

$$\mathit{varseq}_{DT} : DT \to \operatorname{seq} V$$

The unsorted shorthand notation for structured data types is:

$$\begin{aligned}
&\mathit{var}_{DT} : DT \to \mathcal{P}(V)\\
&t \mapsto \{v \in V \mid \exists\, i \in \mathbb{N} \bullet (i, v) \in \mathit{varseq}_{DT}(t)\}
\end{aligned}$$

Only structured data types contain properties:

$$\begin{aligned}
&\forall\, t \in DT \bullet\\
&\qquad \mathit{kind}_{DT}(t) \neq \mathit{struc} \Rightarrow \mathit{var}_{DT}(t) = \varnothing
\end{aligned}$$

Properties within structured data types are unique:

$$\begin{aligned}
&\forall\, p \in V, t \in DT, i_1, i_2 \in \mathbb{N} \bullet\\
&\qquad ((i_1, p) \in \mathit{varseq}_{DT}(t) \wedge (i_2, p) \in \mathit{varseq}_{DT}(t) \Rightarrow i_1 = i_2)\\
&\forall\, p \in V, t_1, t_2 \in DT \bullet\\
&\qquad (p \in \mathit{var}_{DT}(t_1) \wedge p \in \mathit{var}_{DT}(t_2) \Rightarrow t_1 = t_2)
\end{aligned}$$

Properties are either part of a structured data type, or part of an agent:

$$\begin{aligned}&\forall\, p \in V, t \in DT \bullet\\&\qquad p \in var_{DT}(t) \Rightarrow \nexists\, a \in A \bullet (p \in var_A(a) \vee p \in param_A(a))\\&\forall\, p \in V, a \in A \bullet\\&\qquad (p \in var_A(a) \vee p \in param_A(a)) \Rightarrow \nexists\, t \in DT \bullet p \in var_{DT}(t)\end{aligned}$$

PropertyValue. Properties within the HybridUML model can have a value specification. This value specification determines the initial value of the property. For constant properties, particularly for parameters, the initial value thus is the constant value of the property. It is given by an expression:

$$exp_{\sigma_V} : \sigma_V \rightarrow Exp$$

Each property value specification is mapped to its property:

$$var_{\sigma_V} : \sigma_V \rightarrow V$$

There are no values for the special "ID" parameters:

$$\forall\, a \in A \bullet\ \nexists\, val \in \sigma_V \bullet var_{\sigma_V}(val) = v_{id,a}$$

VariablePort. A variable port provides a property, such that the property can be connected with other properties, by means of variable connectors. Each variable port represents exactly one property:

$$var_{P_V} : P_V \rightarrow V$$

The property must be publicly visible:

$$\forall\, p \in P_V \bullet vis_V(var_{P_V}(p)) = pub$$

The variable port has a separate access policy that can restrict the access of a read/write variable to read-only:

$$acc_{P_V} : P_V \rightarrow \{ro, rw\}$$

Read/write ports for read-only properties are not allowed:

$$\forall\, p \in P_V \bullet acc_V(var_{P_V}(p)) = ro \Rightarrow acc_{P_V}(p) = ro$$

VariablePortInstance. Variable port instances correspond to variable ports. Each variable port instance represents a variable port for a particular agent instance. Thus, variable port instances are mapped to their variable port:

$$port_{PI_V} : PI_V \rightarrow P_V$$

There can be several variable port instances that correspond to a variable port, particularly a variable port instance may represent only a subset of the indices which are defined by the property's multiplicity. An expression defines the set of indices which are represented by the variable port instance:

$$indices_{PI_V} : PI_V \rightarrow Exp$$

VariableConnector. A variable connector connects an arbitrary number of variable port instances and up to one variable port:

$$\begin{aligned}
&\mathit{portIns}_{C_V} : C_V \to \mathcal{P}(PI_V)\\
&\mathit{port}_{C_V} : C_V \to \mathcal{P}(P_V)\\
&\forall\, c \in C_V \bullet |\mathit{port}_{C_V}(c)| \leq 1
\end{aligned}$$

The data types of the port instances and of the port must match. This ensures that the resulting shared variable has a well-defined type.

$$\begin{aligned}
&\forall\, c \in C_V \bullet \forall\, p_1, p_2 \in \mathit{port}_{PI_V}(\mathit{portIns}_{C_V}(c)) \cup \mathit{port}_{C_V}(c) \bullet\\
&\qquad \mathit{type}_V(\mathit{var}_{P_V}(p_1)) = \mathit{type}_V(\mathit{var}_{P_V}(p_2))
\end{aligned}$$

All connected port instances and the port must be local to the agent that owns the connector. The port instances are from the contained subagent instances and therefore connect their properties. The (optional) local port is a port of the agent itself, and thus make the internally connected variables externally available for connection, through agent instances of this agent.

$$\begin{aligned}
&\forall\, c \in C_V, a \in A \bullet c \in \mathit{conn}_{A,\mathit{Var}}(a) \Rightarrow\\
&\qquad ((\forall\, p \in \mathit{port}_{C_V}(c) \bullet p \in \mathit{port}_{A,\mathit{Var}}(a))\\
&\qquad \wedge\\
&\qquad (\forall\, pi \in \mathit{portIns}_{C_V}(c), ai \in AI \bullet\\
&\qquad\ \ pi \in \mathit{portIns}_{AI,\mathit{Var}}(ai) \Rightarrow ai \in \mathit{part}_A(a)))
\end{aligned}$$

There are different kinds of connectors:

$$\mathit{kind}_{C_V} : C_V \to \{\mathit{ptp}, \mathit{mult}\}$$

They differ in how indices of variables are mapped. A *point-to-point* connector matches each index individually and thus acts as a set of separate connectors. A *multicast* connector acts as a single connector that connects every two indices of every two variables.

Signal. HybridUML signals are themselves not typed, but have a parameter list that is defined by a list of data types. Signals are always processed at discrete points in time, therefore the special type *anaReal* is inappropriate here.

$$\mathit{paramTypes}_S : S \to \mathrm{seq}(DT \setminus \{\mathit{anaReal}\})$$

Similarly to properties, a signal specification defines an array of $n \geq 1$ copies. Therefore, signals also have a multiplicity:

$$\mathit{mult}_S : S \to \mathit{Exp}$$

SignalPort. A signal port provides a signal for connection with other signals, in the same way as variable ports provide properties. Each signal port acts for one signal:

$$\mathit{sig}_{P_S} : P_S \to S$$

The signal port has an access policy that defines whether the signal can be sent or received through it:

$$acc_{P_S} : P_S \rightarrow \{recv, send\}$$

In contrast to variable ports, signal ports control the access to signals exclusively, since signals do not have access policies themselves. However, there must be no signal that is received *and* sent, therefore all ports of a signal have the same access policy:

$$\forall\, p_1, p_2 \in P_S \bullet sig_{P_S}(p_1) = sig_{P_S}(p_2) \Rightarrow acc_{P_S}(p_1) = acc_{P_S}(p_2)$$

SignalPortInstance. Signal port instances correspond to signal ports, in the same way that variable port instances correspond to variable ports:

$$port_{PI_S} : PI_S \rightarrow P_S$$

According to variable port instances, an expression defines the set of the port indices which are represented by this port instance:

$$indices_{PI_S} : PI_S \rightarrow Exp$$

SignalConnector. A signal connector connects an arbitrary number of signal port instances and up to one signal port:

$$\begin{aligned} &portIns_{C_S} : C_S \rightarrow \mathcal{P}(PI_S) \\ &port_{C_S} : C_S \rightarrow \mathcal{P}(P_S) \\ &\forall\, c \in C_S \bullet |port_{C_S}(c)| \leq 1 \end{aligned}$$

The parameter lists of the signals which are represented by the port instances and of the port must match, such that a signal and particularly its actual parameters can be successfully transmitted over the connector:

$$\begin{aligned} &\forall\, c \in C_S \bullet \forall\, p_1, p_2 \in port_{PI_S}(portIns_{C_S}(c)) \cup port_{C_S}(c) \bullet \\ &\quad paramTypes_S(sig_{P_S}(p_1)) = paramTypes_S(sig_{P_S}(p_2)) \end{aligned}$$

As with variable connectors, all connected port instances and the port must be local to the agent that owns the connector, in order to connect port instances from the contained subagent instances internally, and to optionally provide them for external connection:

$$\begin{aligned} &\forall\, c \in C_S, a \in A \bullet c \in conn_{A,Sig}(a) \Rightarrow \\ &\quad ((\forall\, p \in port_{C_S}(c) \bullet p \in port_{A,Sig}(a)) \\ &\quad \wedge \\ &\quad (\forall\, pi \in portIns_{C_S}(c), ai \in AI \bullet \\ &\quad\; pi \in portIns_{AI,Sig}(ai) \Rightarrow ai \in part_A(a))) \end{aligned}$$

The kinds of connectors coincide with variable connector kinds, such that point-to-point and multicast connections are distinguished for signal connectors, too:

$$kind_{C_S} : C_S \rightarrow \{ptp, mult\}$$

4.2 Behavioral Specification

This section provides the part of the HybridUML specification syntax that defines the system's *behavior*. Graphically, the behavior was specified by *statechart diagrams* in section **??**.

Mode. A mode acts as a definition of behavior. Modes are hierarchical *state machines*, in that they can contain submodes. In contrast to plain UML, each submode is always a state machine itself, therefore we do not distinguish between *state machine* and *state*. Further, we prefer the term *mode*, since for hybrid systems, a state usually describes the combination of *control state* and *data state*, the latter including the current point in time. Therefore, residing in a specific control state for a positive time duration involves a *set of states*, which are subsumed to a so-called *mode*.[4] In order to create the behavioral hierarchy, a mode may contain submode instances:

$$submode_M : M \to \mathcal{P}(MI)$$

Every submode instance is at most part of one mode:

$$\forall\, mi \in MI, m_1, m_2 \in M \bullet \\ (mi \in submode_M(m_1) \wedge mi \in submode_M(m_2) \Rightarrow m_1 = m_2)$$

Discrete behavior is modeled by transitions, which conceptually transfer control from a currently active mode to a new mode. Technically, modes are equipped with *control points* which act as exit or entry for transitions:

$$cp_M : M \to \mathcal{P}(CP)$$

Each control point is exclusively contained by one mode:

$$\forall\, c \in CP, m_1, m_2 \in M \bullet \\ (c \in cp_M(m_1) \wedge c \in cp_M(m_2) \Rightarrow m_1 = m_2)$$

There are two special control points for each mode – *default entry* and *default exit*:

Default Exit Transitions starting from the default exit point can fire independently of the mode's internal state, and thus are so-called *group transitions*.[5] In contrast, transitions starting from other exit points can only be taken, if control is explicitly transferred to that exit point before.

Default Entry Transitions entering the default entry point resume the *history* of that mode, i.e. the submodes which were active before a preceding interrupt are reactivated. If no history is available, an initialization transition activates an initial mode. However, mode entry via normal entry points always requires explicit control transfer to a submode.

[4]These terms particularly comply to [AGLS01]. In [Hen96], instead of *mode* the term *control mode* is used.

[5]Group transitions are called *interrupt transitions*, alternatively.

$$\begin{aligned}
&de_M : M \rightarrow CP\\
&\forall\, m \in M \bullet (de_M(m) \in cp_M \wedge kind_{CP}(de_M(m)) = entry)\\
&dx_M : M \rightarrow CP\\
&\forall\, m \in M \bullet (dx_M(m) \in cp_M \wedge kind_{CP}(dx_M(m)) = exit)
\end{aligned}$$

A mode then can contain transitions that connect its own control points and control points of submodes. Transitions between submodes switch control within the mode, whereas transitions to or from the mode itself prepare the loss of control or handle the gain of control of the mode, respectively.

$$trans_M : M \rightarrow \mathcal{P}(T)$$

No transition is contained by more than one mode:

$$\begin{aligned}
&\forall\, t \in T, m_1, m_2 \in M \bullet\\
&\qquad (t \in trans_M(m_1) \wedge t \in trans_M(m_2) \Rightarrow m_1 = m_2)
\end{aligned}$$

Continuous behavior is given by algebraic and flow constraints that define continuous evolutions of variables when the mode is active. Both are given by expressions, see chapter 5 for details on them, including the distinction between them.

$$\begin{aligned}
&flow_M : M \rightarrow \mathcal{P}(Exp)\\
&alge_M : M \rightarrow \mathcal{P}(Exp)
\end{aligned}$$

Invariant constraints define, whether the mode may be currently active or not. This can particularly disable the continuous behavior and thus enforce discrete behavior. Technically, invariants are given by boolean expressions, which are discribed in chapter 5.

$$inv_M : M \rightarrow \mathcal{P}(Exp)$$

ModeInstance. A mode instance is a concrete occurrence of a mode. The discrimination of modes and mode instances is provided solely for re-use of mode specifications. In the graphical examples of section **??**, modes are mostly defined implicitly along with a single mode instance specification. An example of seperate mode and mode instance definition is given in Fig. **??** and Fig. **??**.

$$mode_{MI} : MI \rightarrow M$$

Accordingly, the mode's control points are also instantiated. For each control point of the mode, exactly one control point instance exists for the mode instance:

$$\begin{aligned}
&cpi_{MI} : MI \rightarrow \mathcal{P}(CPI)\\
&\forall\, mi \in MI \bullet \forall\, c \in cp_M(mode_{MI}(mi)) \bullet\\
&\qquad \exists\, ci \in cpi_{MI}(mi) \bullet cp_{CPI}(ci) = c\\
&\forall\, mi \in MI \bullet \forall\, ci_1, ci_2 \in cpi_{MI}(mi) \bullet\\
&\qquad (cp_{CPI}(ci_1) = cp_{CPI}(ci_2) \Rightarrow ci_1 = ci_2)\\
&\forall\, mi \in MI, ci \in CPI \bullet\\
&\qquad ci \in cpi_{MI}(mi) \Rightarrow cp_{CPI}(ci) \in cp_M(mode_{MI}(mi))
\end{aligned}$$

A control point instance is at most part of one mode instance:

$$\begin{aligned}&\forall\, ci \in CPI, mi_1, mi_2 \in MI \bullet\\&\quad (ci \in cpi_{MI}(mi_1) \wedge ci \in cpi_{MI}(mi_2) \Rightarrow mi_1 = mi_2)\end{aligned}$$

ControlPoint. Control points define the entries to modes and exits from modes. *Exit points* act as sources of transitions, whereas *entry points* act as as transition targets.

$$kind_{CP} : CP \rightarrow \{entry, exit\}$$

ControlPointInstance. A control point instance is a concrete occurrence of a control point and is part of a mode instance:

$$cp_{CPI} : CPI \rightarrow CP$$

Transition. A transition represents a discrete behavioral step. It implicitly connects mode instances or a mode and a submode instance, by connecting their control point instances or a control point with a control point instance. Three combinations are allowed:

(1) Connecting an entry point e of mode m with an entry point instance e_{sub} of a submode instance m_{sub}: When control is transferred to m via e, then m_{sub} gains control through e_{sub}.
(2) Connecting an exit point instance x_1 of a submode m_1 with an entry point instance e_2 of a submode m_2: When the transition fires, control is transferred from m_1 to m_2.
(3) Connecting an exit point instance x_{sub} of a submode instance m_{sub} with a non-default exit point x of the parent mode m: When the transition fires, control is transferred to x, such that outgoing transitions from x can fire and m loses control.

Note that transitions thus are not only used to switch control *between* modes, but also determine how control is assigned *within* modes.

$$\begin{aligned}&src_T : T \rightarrow \{CP \cup CPI\}\\&tar_T : T \rightarrow \{CP \cup CPI\}\\&\forall\, t \in T \bullet\\&\qquad src_T(t) \in CP \wedge kind_{CP}(src_T(t)) = entry\\&\qquad \wedge tar_T(t) \in CPI \wedge kind_{CP}(cp_{CPI}(tar_T(t))) = entry\\&\quad \vee\\&\qquad src_T(t) \in CPI \wedge kind_{CP}(cp_{CPI}(src_T(t))) = exit\\&\qquad \wedge tar_T(t) \in CPI \wedge kind_{CP}(cp_{CPI}(tar_T(t))) = entry\\&\quad \vee\\&\qquad src_T(t) \in CPI \wedge kind_{CP}(cp_{CPI}(src_T(t))) = exit\\&\qquad \wedge tar_T(t) \in CP \wedge kind_{CP}(tar_T(t)) = exit\end{aligned}$$

If either source or target is a control point, then the control point must be contained by the same mode as the transition itself:

$$\forall\, t \in T, m \in M \bullet$$

$$\begin{aligned}
&\quad src_T(t) \in CP \wedge t \in trans_M(m) \Rightarrow src_T(t) \in cp_M(m) \\
&\forall\, t \in T, m \in M \bullet \\
&\quad tar_T(t) \in CP \wedge t \in trans_M(m) \Rightarrow tar_T(t) \in cp_M(m)
\end{aligned}$$

If either source or target is a control point instance, then the control point instance must be contained by a submode instance of the mode that contains the transition:

$$\begin{aligned}
&\forall\, t \in T, m \in M, mi \in MI \bullet \\
&\quad t \in trans_M(m) \wedge src_T(t) \in cpi_{MI}(mi) \Rightarrow mi \in submode_M(m) \\
&\forall\, t \in T, m \in M, mi \in MI \bullet \\
&\quad t \in trans_M(m) \wedge tar_T(t) \in cpi_{MI}(mi) \Rightarrow mi \in submode_M(m)
\end{aligned}$$

The target of a transition must not be a default exit point:

$$\begin{aligned}
&\forall\, t \in T, m \in M \bullet \\
&\quad tar_T(t) \in CP \wedge t \in trans_M(m) \Rightarrow tar_T(t) \neq dx_M(m)
\end{aligned}$$

A transition can be equipped with a trigger, which is the specification of a signal to be received. If the signal is currently active (and the guard expression is satisfied, and the source control point has control), then the transition (or a concurrently enabled one) *must* be taken.

$$\begin{aligned}
&sig_T : T \rightarrow \mathcal{P}(Exp) \\
&\forall\, t \in T \bullet \\
&\quad |sig_T(t)| \leq 1
\end{aligned}$$

A transition can have a guard, which is the specification of a boolean expression. If the guard is satisfied (and the source control point has control), and either (1) there is no trigger specification or (2) the trigger's signal is active, then the transition is enabled. In absence of a trigger, the transition *can* be taken. Particularly, as long as the invariant constraints of the source mode instance are fulfilled, the system may reside in that mode instance and may continue continuous behavior.

$$\begin{aligned}
&grd_T : T \rightarrow \mathcal{P}(Exp) \\
&\forall\, t \in T \bullet \\
&\quad |grd_T(t)| \leq 1
\end{aligned}$$

Each transition has a list of actions to be taken when the transition fires. Actions, as well as triggers and guards, are given by expressions and are discussed in chapter 5.

$$act_T : T \rightarrow \operatorname{seq} Exp$$

Chapter 5

HybridUML Expression Language

The HybridUML Expression Language (HybEL) defines the syntax of expressions $exp \in Exp_{HybEL}$ that can be used within HybridUML specifications. The diagrammatic part of the specification embeds textual parts which define constraints and assignments for the behavior as well as for the structure of the system.

HybridUML is not restricted to be used with the HybridUML Expression Language; the application of different expression languages is possible. Syntactically, the set *Exp* of expressions contained in a given HybridUML specification can be any set of expressions according to an arbitrary expression language. Semantically, however, the choice of expression language has a heavy impact on the transformation Φ defined in this thesis. For any HybridUML specification with arbitrary expressions (of the chosen language), the transformation explicitly defines its semantics. This directly depends on the chosen expression language.

Therefore, HybEL is customized as a small expression language that fits HybridUML, such that

- it is expressive enough for the design of embedded applications,
- it supports HybridUML's main concepts, particularly the calculation of time-continuous values,
- and that it is concise in order to facilitate a comprehensible semantics by transformation.

The most obvious alternative is the Object Constraint Language (OCL) [OMG], which is UML's standard expression language. In many respects, OCL is more expressive than HybridUML Expression Language, e.g. it contains powerful collection operations providing set comprehension, as well as universal and existential quantifiers on sets of arbitrary objects. We evaluated the combination of HybridUML and OCL by means of the BART case study, with focus on validation and verification [BZL04]. The applied OCL-based validation and verification concept is described in [Ric02].

However, many features of OCL are useless for HybridUML specifications, such as navigation along associations, since HybridUML excludes a considerable amount of UML features, e.g. associations. In contrast, OCL does not fulfill every demand on an expression language for HybridUML. For example, OCL is explicitly designed to be effect-free and thus does not provide assignment expressions. As a consequence, in order to use OCL with the transformation concept of this thesis, extensive customizations would be required, consisting of restrictions of as well as extensions to the OCL standard. To our opinion, this would reduce the main benefit of using OCL significantly. Therefore, we prefer the custom-made expression language HybEL, and assume $Exp \subseteq Exp_{HybEL}$ for this thesis.

The rest of this chapter provides the following: (1) The relation of HybEL expressions to a given HybridUML specification is described in section 5.1 – *roles* of expressions are identified, leading to the definition of the *context* of expressions. This implies a conceptual separation of full HybEL expressions and embedded *identifier expressions.* Further it is motivated that HybEL expressions do not have a complete semantics on their own, but only within the complete transformation Φ of HybridUML specifications. (2) Section 5.2 provides the syntax and the *intermediate semantics* of identifier expressions, which is a data structure that acts as input for the transformation Φ. (3) The syntax and intermediate semantics of full HybEL expressions is given in section 5.3. (4) Finally, section 5.4 defines an incomplete semantics which is used for the syntax of HybEL expressions in section 5.3.1.

For a less detailed overview of the syntax of the HybridUML Expression Language, appendix **??** contains an EBNF grammar, which omits some of the details given in this chapter.

5.1 Context of Expressions

This section defines the *context of expressions.* As a prerequisite, *roles of expressions* within a given HybridUML specification are identified. Implicitly, these were already given in chapter 4, but are repeated here explicitly in order to point out the application of expressions within the specification. From this, a list of required language features is determined. The dependencies of expressions on the given HybridUML specification are identified.

Roles of Expressions. For use with the *structural* specification part, expressions are attached to properties (including parameters), signals, variable ports, signal ports, agents, and agent instances:

Initial Property Values The expression $exp = exp_{\sigma_V}(val)$ of a property value $val \in \sigma_V$ is an expression that determines an initial value for the associated property $v = var_{\sigma_V}(val)$. The value is calculated from literals, variables, and operations, such that the resulting type is $t = type_V(v)$. For properties v of agents (i.e. $\exists\, a \in A \bullet v \in var_A(a)$), the expression may refer to variables v_p of parent agents, i.e. for which holds: $\exists\, a, a_p \in A, ai \in AI \bullet (v_p \in var_A(a_p) \wedge ai \in part_A(a_p) \wedge agent_{AI}(ai) = a \wedge v \in var_A(a))$. In contrast, for properties v of structured data types (i.e. $\exists\, t \in DT_{struc} \bullet v \in var_{DT}(t)$), no variables are available.

Init State Constraints Constraints on the initial state of the system can be specified by boolean expressions $exp \in initState_{AI}(ai)$ per agent instance ai or by boolean expressions $exp \in initState_{A}(a)$ per agent a, in order to model a *set* of admissible start states, rather than a single one.[1] The boolean result is determined from literals, variables, and operations on them. Variables v_a from the attached agent are accessible, that are $v_a \in var_A(a)$ or $v_a \in var_A(agent_{AI}(ai))$, respectively.

Property Multiplicities For each property v a multiplicity expression $exp = mult_V(v)$ is given. It defines the number n of copies for this property, such that an array of n different values $v[0]..v[n-1]$ results. Each expression exp must be of type int and must evaluate to a natural number. It is calculated from numerical literals, operations, and variables, which are – for properties of agents, i.e. $\exists\, a \in A \bullet v \in var_A(a))$ – the variables v_a of the agent containing v itself, i.e. for which hold: $\exists\, a \in A \bullet \{v, v_a\} \subseteq var_A(a)$. No variables are accessible for the calculation of variables of structured data types, i.e. if $\exists\, t \in DT_{struc} \bullet v \in var_{DT}(t)$.

Signal Multiplicities For each signal s a multiplicity expression $exp = mult_S(s)$ is given, similarly to property multiplicities.

Agent Instance Multiplicities An agent instance ai is equipped with an integer-typed multiplicity expression $exp = mult_{AI}(ai)$ that defines the number n of copies that it represents, in the same way that properties and signals are. For the calculation of the integer value, variables v_p from the parent agent are available – for which hold: $\exists\, a_p \in A \bullet (v_p \in var_A(a_p) \wedge ai \in part_A(a_p))$.

Property Index Specifications Since variable port instances pi may represent a subset instead of all copies of the associated property v, an expression $exp = indices_{PI_V}(pi)$ is given that defines a set *indices* of integer-typed values. With a property multiplicity n, the resulting set of represented property copies is $\{v[i] \mid i \in \{0\,..\,n-1\} \cap indices\}$. For the calculation, literals, operations, and variables from the parent agent can be used. Thus, the variables v_p with $\exists\, a, a_p \in A, ai \in AI \bullet (v_p \in var_A(a_p) \wedge ai \in part_A(a_p) \wedge agent_{AI}(ai) = a \wedge v \in var_A(a))$ can be referred to.

Signal Index Specifications Signal index specifications $exp = indices_{PI_S}(pi)$ are similar to variable index specifications.

Within the *behavioral* specification part, expressions are attached to either modes or transitions. The modes' expressions are:

Flow Constraints A flow constraint $f \in flow_M(m)$ for some mode m is an expression that is suitable to define a continuous evolution of a variable. Therefore it is an expression that calculates a numeric value and assigns it to a variable v of type $type_V(v) = anaReal$. This can be a conventional assignment, i.e. the variable is directly assigned, and the value is determined from numeric literals, variables, and operations. Alternatively, derivatives of variables can be used either for the calculation or for the assignment, i.e. either for the calculation, a derivative is read, or the value of a variable is assigned indirectly by assigning its derivative.

[1]The distinction between "Initial Property Values" and "Init State Constraints" is motivated in chapter 6.

Algebraic Constraints An algebraic constraint $a \in alge_M(m)$ for some mode m is, like a flow constraint, an expression that calculates a numeric value and assigns it to a variable v of type $type_V(v) = anaReal$. The difference is that no derivatives can be used with algebraic constraints.

Invariant Constraints An invariant constraint $i \in inv_M(m)$ for some mode m is an expression that defines iff the mode may be active. It provides a boolean value, which is determined from variables and literals of any type, potentially by the application of suitable operations.

The expressions attached to transitions are:

Trigger The trigger $sig_T(t)$ of a transition t is an expression that defines exactly one signal s to be received, in order to enable t. If the signal carries parameter values, a list of writable variables has to be defined which store these values on signal reception. The variable types must correspond to the signal's parameter definition $paramTypes_S(s)$.

Guard The guard $grd_T(t)$ of a transition t is a boolean expression that enables or disables t. Similarly to invariant constraints, variables, literals, and operations on them calculate the boolean result. If guard and trigger are both present, their conjunction enables t.

Action The actions $act_T(t)$ of a transition define a sequence of assignments and signal raise statements that are to be executed when the transition fires. An assignment calculates a new value for some variable v of any type. The value is given by an expression of appropriate type, which consists of literals, variables, and operations on them. A signal raise statement defines a signal to be sent. A list of expressions must be provided if the signal is supposed to carry parameter values. The expression types must correspond to the signal's parameter definition $paramTypes_S(s)$, such that the signal's parameters are determined.

Because behavioral specifications, i.e. top-level mode instances mi and all their (recursively) included submode instances and transitions do not have own variables or signals, but are associated with agent instances, they refer to the variables v and signals s provided by the corresponding agent instance ai, i.e. $behavior_A(agent_{AI}(ai)) = mi \wedge v \in var_A(agent_{AI}(ai))$ or $behavior_A(agent_{AI}(ai)) = mi \wedge s \in sig_A(agent_{AI}(ai))$, respectively.

Required HybEL Features. From the possible roles of expressions within the HybridUML specification, a list of elements results that the expression language must provide. From this feature list, the language is developed. The required features are:

1. Literals of all types $t \in DT$.

2. Access to variables v of all types $type_V(v) \in DT$. Read and write access are distinguished.

3. Access to derivatives of variables v of type $t = anaReal$. Read and write access are distinguished.

4. Common numeric and boolean operations, like basic arithmetics and comparison of values. Operations depend on the number and type of their arguments.

5. Special operations that define finite sets of integers $iset \in \mathcal{P}(int)$.

6. Assignments to writable variables of all types $t \in DT$.

7. Assignments to derivatives of writable variables of type $t = anaReal$.

8. Access to signals s. Receive and send access are distinguished.

9. Reception of signals, incl. parameter specifications.

10. Sending of signals, incl. parameter specifications.

11. Index expressions for the determination of variable or signal indices.

Dependency to the HybridUML Specification. The available variables and signals within a particular expression depend on the expression's role, and are given by a dedicated agent, which is an associated agent, a parent agent, or an embedding agent. Further, the absence of a corresponding agent leads to an empty set of available variables and signals.

In contrast, the availability of literals, operations, assignments, as well as signal receive and send statements is independent of the embedding specification. Thus, it is exactly the variables and signals that constitute the expression's dependency to the HybridUML specification.

Context of Expressions. The *context of expressions* is the set of available variables and signals. Since the access policy of variables and signals is relevant within expressions, but signals themselves have none, it is added explicitly:

$$CTX = \mathcal{P}(V \cup V_{local}) \times \mathcal{P}(S \times \{recv, send\})$$

The contained variables and signals can be accessed by

$$\begin{aligned}
&var_{CTX} : CTX \to \mathcal{P}(V \cup V_{local}) \\
&\qquad (var, sig) \mapsto var \\
&sig_{CTX} : CTX \to \mathcal{P}(S \times \{recv, send\}) \\
&\qquad (var, sig) \mapsto sig
\end{aligned}$$

The definition of syntactically correct expressions (given in the following sections) is relative to the given context, thus the context encapsulates the expressions' dependencies to the HybridUML specification.

For one technical reason, the context is not defined by the agent which provides the variables and signals: There are (sub-)expressions that are defined within a different context than that of an agent – bound variables $v_{local} \in V_{local}$ introduced by quantified expressions extend the context for contained expressions, and identifiers embedded into structured data types are defined in the context of the data type.

Two-Level Syntax. As a consequence from the dependency of expressions on variables and signals, the HybEL syntax is partitioned into *identifier expressions* and full *hybel expressions.* Identifier expressions define valid identifiers which represent variables or signals from the given context. This includes sub-identifiers $id_{sdt}.id_{sub}$ of structured data types, as well as indexed identifiers $id[\langle exp_{idx}\rangle]$. Full hybel expressions define complete expressions that (can) contain identifier expressions. Note that this is not a two-level definition in the strict sense, but a mutually recursive definition: The index expressions exp_{idx} within the identifier expressions are full hybel expressions themselves.[2]

The syntax of identifier expressions is given in section 5.2.1, the full hybel identifier syntax is defined in section 5.3.1.

Semantics. There are two different ways in which HybEL expressions are interpreted:

Evaluation Semantics The expressions from the structural specification part are *evaluated* during the transformation Φ from HybridUML specifications into the executable system, when the static structure of the system is created. Effectively, only a subset of the available HybEL expressions is available for the structural specification, because only expressions of type $t \in \{int, \mathcal{P}(int)\}$ are applied there.
The mapping *eval* provides the associated values. As described in chapter 6, agent instances and therefore also the attached variables are duplicated, and may have different values assigned. This implies that the evaluation of expressions which refer to variables depends on the particular duplicate; thus *eval* in only defined within the transformation Φ, rather than independently. A stand-alone, but less powerful variant $eval_{\varnothing}$ which omits variables is given in section 5.4. It is sufficient for the syntax definition of hybel expressions in section 5.3.1.

Transformation Semantics Expressions from the behavioral specification part as well as property and signal multiplicities from the structural specification part are *transformed* by Φ into (parts of) the executable system and evaluated at run-time. For this, the full extent of HybEL is exploited. Obviously, this also depends on the transformation Φ, and is discussed in section 6.5 of chapter 6.

In this chapter, *intermediate semantics* are defined for identifier expressions (section 5.2.2) and for full hybel expressions (section 5.3.2). From this, the *evaluation semantics* as well as the *transformation semantics* are defined.

5.2 Identifier Expressions

Identifier expressions are subexpressions of HybEL expressions that identify either a variable or a signal within a given context.

For the given HybridUML specification, there is a set *Id* of available identifiers. These are associated with the variables and signals from the specification. In addition to the variables $v \in V$ and signals $s \in S$, there are local variables

[2]Technically, this is handled by separate parser runs which omit index expressions at first.

$v_{local} \in V_{local}$ with $V \cap V_{local} = \varnothing$ which act as bound variables within expressions. Local variables are always of type *int*. Each variable $v \in V \cup V_{local}$ or signal $s \in S$ of the specification has an identifier $id \in Id$:

$$\begin{aligned}
&id_V : V \rightarrow Id\\
&id_{V_{local}} : V_{local} \rightarrow Id\\
&id_{V,V_{local}} = id_V \cup id_{V_{local}}\\
&id_S : S \rightarrow Id
\end{aligned}$$

The sets of identifiers for variables and signals are disjoint:

$$\operatorname{ran} id_V \cap \operatorname{ran} id_{V_{local}} \cap \operatorname{ran} id_S = \varnothing$$

Each identifier of a variable $v \in V$ and each identifier of a signal $s \in S$ is unique within the embedding agent or data type, respectively:

$$\begin{aligned}
&\forall\, a \in A \bullet \forall\, v_1, v_2 \in var_A(a) \bullet id_V(v_1) = id_V(v_2) \Rightarrow v_1 = v_2\\
&\forall\, t \in DT \bullet \forall\, v_1, v_2 \in var_{DT}(a) \bullet id_V(v_1) = id_V(v_2) \Rightarrow v_1 = v_2\\
&\forall\, a \in A \bullet \forall\, s_1, s_2 \in sig_A(a) \bullet id_S(s_1) = id_S(s_2) \Rightarrow s_1 = s_2
\end{aligned}$$

Each local variable has a unique identifier within Id, i.e. the mapping $id_{V_{local}}$ is injective.

Available identifiers within a certain context are provided by respective mappings:

$$\begin{aligned}
&id_{CTX,var} : CTX \rightarrow \mathcal{P}(Id)\\
&\qquad c \mapsto \{id \in Id \mid \exists\, v \in var_{CTX}(c) \bullet id_{V,V_{local}}(v) = id\}\\
&id_{CTX,sig} : CTX \rightarrow \mathcal{P}(Id)\\
&\qquad c \mapsto \{id \in Id \mid \exists (s, acc) \in sig_{CTX}(c) \bullet id_V(s) = id\}\\
&id_{CTX,sig,recv} : CTX \rightarrow \mathcal{P}(Id)\\
&\qquad c \mapsto \{id \in Id \mid \exists (s, recv) \in sig_{CTX}(c) \bullet id_V(s) = id\}\\
&id_{CTX} : CTX \rightarrow \mathcal{P}(Id)\\
&\qquad c \mapsto id_{CTX,var}(c) \cup id_{CTX,sig}(c)
\end{aligned}$$

5.2.1 Syntax of Identifier Expressions

The syntax of identifier expressions is given by the mapping

$$syn_{id} : CTX \times CTX \rightarrow \mathcal{P}(\operatorname{seq} \Sigma_{Exp})$$

whereas the alphabet Σ_{Exp} contains all variable and signal identifiers of the HybridUML specification. Additionally, terminal symbols for the separation of structured data type's identifiers, as well as terminal symbols for index expressions of indexed variables or signals are provided:

$$\operatorname{ran} id_V \cup \operatorname{ran} id_{V_{local}} \cup \operatorname{ran} id_S \cup \{., [,]\} \subset \Sigma_{Exp}$$

The full definition will be given in section 5.3.1.

The syntax of identifier expressions is affected by two contexts: (1) The local context provides the set of variables and signals for which the current identifier must match. (2) A top-level context is needed for embedded index expressions: within structured data types, embedded identifiers can be equipped with index expressions – these have to be evaluated within the context of the outer structured data type identifier, rather than in the local context. Valid identifier expressions are:

(1) Simple identifiers, identifying a variable or signal: id
(2) Indexed identifiers, identifying a variable or signal with a given index: $id[17]$
(3) Signal identifier with written index. This identifies a signal to be received with any index wrt. its multiplicity. When the signal is received at index i, then i is stored in the variable that is determined by the index expression. For example: $id_{sig}[:= id_{int,rw}]$
(4) Sub-identifier from a simple variable of structured data type, e.g.: $id_{sdt}.id_{sub}$
(5) Sub-identifier from an indexed variable of structured data type, e.g.: $id_{sdt}[17].id_{sub}$

$$\begin{aligned}
& syn_{id}(c, c_{tl}) = \\
& \qquad \{\langle id\rangle \mid id \in id_{CTX}(c)\} \\
& \quad \cup \quad \{\langle id, [\rangle ^\frown exp_{idx} ^\frown \langle]\rangle \mid id \in id_{CTX}(c) \wedge exp_{idx} \in syn_{int}(c_{tl})\} \\
& \quad \cup \quad \{\langle id, [\rangle ^\frown exp_{idx} ^\frown \langle]\rangle \mid id \in id_{CTX,sig,recv}(c) \wedge exp_{idx} \in syn_{idxass}(c_{tl})\} \\
& \quad \cup \quad \{\langle id, .\rangle ^\frown exp_{sub} \mid id \in id_{CTX,var}(c) \wedge \exists\, v \in var_{CTX}(c) \bullet \\
& \qquad\quad (id_{V,V_{local}}(v) = id \wedge exp_{sub} \in syn_{id}((var_{DT}(type_V(v)), \varnothing), c_{tl})\} \\
& \quad \cup \quad \{\langle id, [\rangle ^\frown exp_{idx} ^\frown \langle], .\rangle ^\frown exp_{sub} \mid \\
& \qquad\quad id \in id_{CTX,var}(c) \wedge exp_{idx} \in syn_{int}(c_{tl}) \wedge \exists\, v \in var_{CTX}(c) \bullet \\
& \qquad\quad (id_{V,V_{local}}(v) = id \wedge exp_{sub} \in syn_{id}((var_{DT}(type_V(v)), \varnothing), c_{tl})\}
\end{aligned}$$

The syntax of index expressions is defined in section 5.3.1, given by syn_{int} and syn_{idxass}, respectively. It is defined as a subset of complete HybEL expressions.

As a shorthand, for equal local context and top-level context, we use

$$\begin{aligned}
& syn_{id} : CTX \to \mathcal{P}(\operatorname{seq} \Sigma_{Exp}) \\
& \qquad c \mapsto syn_{id}(c, c)
\end{aligned}$$

5.2.2 Intermediate Semantics of Identifier Expressions

Expressions $exp \in \operatorname{seq} \Sigma_{Exp}$ which represent variables or signals are mapped to trees of *identifier items*. Such a tree is the *intermediate semantics* of an identifier expression. In contrast, the final semantics is given in chapter 6, in the context of integrated transformation Φ of HybridUML specifications and contained HybEL expressions.

Identifier items consist of a *type* and a *value*:

$$IdItem = IdItemType \times IdItemVal$$

The item type is either the type of the variable or signal, or a special type that identifies a contained index expression. Signal types distinguish signals to be sent and signals to be received.

$$IdItemType = DT \cup \{recvSig, sendSig, indexExp\}$$

The possible item values for items that represent an identifier are either variables $v \in V \cup V_{local}$ or signals $s \in S$, with an attached access policy. In contrast, an item of role *indexExp* represents a full-blown HybEL expression that defines an index wrt. the variable's or signal's multiplicity. Consequently, this expression's value is a complete hybel item tree.[3]

$$\mathit{IdItemVal} = (V \times \{\mathit{ro}, \mathit{rw}\}) \cup (S \times \{\mathit{recv}, \mathit{send}\}) \cup \mathrm{tree}_{\mathrm{o}}\, \mathit{HybelItem}$$

For convenience, mappings to type and access policy are given:

$$\begin{aligned}
&\mathit{type}_{\mathit{IdItem}} : \mathit{IdItem} \rightarrow \mathit{IdItemType}\\
&\qquad (t, v) \mapsto t\\
&\mathit{acc}_{\mathit{IdItem}} : \mathit{IdItem} \rightarrow \{\mathit{ro}, \mathit{rw}, \mathit{recv}, \mathit{send}, \lambda\}\\
&\qquad (t, (v, \mathit{acc})) \mapsto \mathit{acc};\ (v, \mathit{acc}) \in V \times \{\mathit{ro}, \mathit{rw}\}\\
&\qquad (t, (s, \mathit{acc})) \mapsto \mathit{acc};\ (s, \mathit{acc}) \in S \times \{\mathit{recv}, \mathit{send}\}\\
&\qquad \mathit{item} \mapsto \lambda\,;\ \text{else}
\end{aligned}$$

For a tree of identifier items, the type is determined from the contained items, which is the innermost for structured data types $t \in DT_{struc}$, otherwise the outermost.

$$\begin{aligned}
&\mathit{type}_{\mathit{IdTree}} : \mathrm{tree}_{\mathrm{o}}\, \mathit{IdItem} \rightarrow \mathit{IdItemType}\\
&\qquad (\mathit{itm}, \langle(\mathit{itm}_1, \mathit{sub}_1), \ldots, (\mathit{itm}_n, \mathit{sub}_n)\rangle) \mapsto \mathit{type}_{\mathit{IdItem}}(\mathit{itm}_n)\\
&\qquad\ ;\ n \geq 1 \wedge \mathit{type}_{\mathit{IdItem}}(\mathit{itm}) \in DT_{struc} \wedge \mathit{type}_{\mathit{IdItem}}(\mathit{itm}_n) \in DT\\
&\qquad (\mathit{itm}, \mathit{sub}) \mapsto \mathit{type}_{\mathit{IdItem}}(\mathit{itm});\ \text{else}
\end{aligned}$$

The access policy for a tree of identifier items is recursively composed of the contained items for structured data types $t \in DT_{struc}$, otherwise the outermost policy is used.

$$\begin{aligned}
&\mathit{acc}_{\mathit{IdTree}} : \mathrm{tree}_{\mathrm{o}}\, \mathit{IdItem} \rightarrow \{\mathit{ro}, \mathit{rw}, \mathit{recv}, \mathit{send}, \lambda\}\\
&\qquad (\mathit{itm}, \langle t_1, \ldots, t_n\rangle) \mapsto \mathit{rw}\\
&\qquad\ ;\ n \geq 1 \wedge \mathit{type}_{\mathit{IdItem}}(\mathit{itm}) \in DT_{struc}\\
&\qquad\quad \wedge \mathit{acc}_{\mathit{IdItem}}(\mathit{itm}) = \mathit{rw} \wedge \mathit{acc}_{\mathit{IdTree}}(t_n) = \mathit{rw}\\
&\qquad (\mathit{itm}, \langle t_1, \ldots, t_n\rangle) \mapsto \mathit{ro}\\
&\qquad\ ;\ n \geq 1 \wedge \mathit{type}_{\mathit{IdItem}}(\mathit{itm}) \in DT_{struc}\\
&\qquad\quad \wedge(\mathit{acc}_{\mathit{IdItem}}(\mathit{itm}) = \mathit{ro} \vee \mathit{acc}_{\mathit{IdTree}}(t_n) = \mathit{ro})\\
&\qquad (\mathit{itm}, \mathit{sub}) \mapsto \mathit{acc}_{\mathit{IdItem}}(\mathit{itm});\ \text{else}
\end{aligned}$$

The mapping to identifier item trees depends on the given context, as the syntax definition does. Two different contexts are provided here, again as a prerequisite for full identifier expressions: contained index expressions will be mapped corresponding to their embedding identifier expression's *top-level* context, whereas sub-identifiers are related to the context provided by the containing data type. For each pair of contexts, there is a mapping from expressions

[3]Ordered trees are recursively defined as pairs of one node and one sequence of subtrees: $\mathrm{tree}_{\mathrm{o}}\, X = X \times \mathrm{seq}(\mathrm{tree}_{\mathrm{o}}\, X)$

to corresponding identifier item trees:

$$\begin{aligned}
&IT = \bigcup_{(c_1,c_2)\in CTX\times CTX} (syn_{id}(c_1,c_2) \rightarrow \mathrm{tree_o}\ \mathit{IdItem}) \\
&it : CTX \times CTX \rightarrow IT
\end{aligned}$$

Simple variables $v \in V$ are represented by a single identifier item that corresponds to the respective type each:

$$\begin{aligned}
&it(c, c_{tl})(\langle id \rangle) = ((type_V(v), (v, acc_V(v))), \langle\rangle) \\
&\qquad ;\ v \in var_{CTX}(c) \cap V \wedge id_V(v) = id
\end{aligned}$$

Simple signals $s \in S$ are also represented by single identifiers; the access policy is distinguished here, because signals to be received and signals to be sent are fundamentally different:

$$\begin{aligned}
&it(c, c_{tl})(\langle id \rangle) = ((recvSig, (s, recv)), \langle\rangle) \\
&\qquad ;\ (s, recv) \in sig_{CTX}(c) \wedge id_S(s) = id \\
&it(c, c_{tl})(\langle id \rangle) = ((sendSig, (s, send)), \langle\rangle) \\
&\qquad ;\ (s, send) \in sig_{CTX}(c) \wedge id_S(s) = id
\end{aligned}$$

Bound identifiers are implicitly of type *int*, and they are always read-only:

$$\begin{aligned}
&it(c, c_{tl})(\langle id \rangle) = ((int, (v, ro)), \langle\rangle) \\
&\qquad ;\ v \in var_{CTX}(c) \cap V_{local} \wedge id_{V_{local}}(v) = id
\end{aligned}$$

Simple identifiers with an attached index expression are mapped similarly to simple identifiers without index expression, but from the index expression, a subtree is created:

$$\begin{aligned}
&it(c, c_{tl})(\langle id, [\rangle \frown exp_{idx} \frown \langle]\rangle) = \\
&\qquad ((type_V(v), (v, acc_V(v))), \langle it(c, c_{tl})(\langle [\rangle \frown exp_{idx} \frown \langle]\rangle)\rangle) \\
&\qquad ;\ v \in var_{CTX}(c) \cap V \wedge id_V(v) = id \\
&it(c, c_{tl})(\langle id, [\rangle \frown exp_{idx} \frown \langle]\rangle) = \\
&\qquad ((recvSig, (s, recv)), \langle it(c, c_{tl})(\langle [\rangle \frown exp_{idx} \frown \langle]\rangle)\rangle) \\
&\qquad ;\ (s, recv) \in sig_{CTX}(c) \wedge id_S(s) = id \\
&it(c, c_{tl})(\langle id, [\rangle \frown exp_{idx} \frown \langle]\rangle) = \\
&\qquad ((sendSig, (s, send)), \langle it(c, c_{tl})(\langle [\rangle \frown exp_{idx} \frown \langle]\rangle)\rangle) \\
&\qquad ;\ (s, send) \in sig_{CTX}(c) \wedge id_S(s) = id
\end{aligned}$$

An identifier expression containing a period defines a sub-identifier that represents a variable from a structured data type. This can be accompanied by an index expression. From the sub-identifier, a subtree is created:

$$\begin{aligned}
&it(c, c_{tl})(\langle id, [\rangle \frown exp_{idx} \frown \langle], .\rangle \frown exp_{suf}) = \\
&\qquad ((type_V(v), (v, acc_V(v))), \\
&\qquad \langle it(c, c_{tl})(\langle [\rangle \frown exp_{idx} \frown \langle]\rangle), it((var_{DT}(type_V(v)), \langle\rangle), c_{tl})(exp_{suf})\rangle) \\
&\qquad ;\ v \in var_{CTX}(c) \cap V \wedge id_V(v) = id
\end{aligned}$$

$$\begin{aligned}&it(c, c_{tl})(\langle id, .\rangle \frown exp_{suf}) = \\ &\quad((type_V(v), (v, acc_V(v))), \\ &\quad\ \langle it((var_{DT}(type_V(v)), \langle\rangle), c_{tl})(exp_{suf})\rangle) \\ &\quad ;\ v \in var_{CTX}(c) \cap V \wedge id_V(v) = id\end{aligned}$$

An index expression defines a contained hybel item tree, which is interpreted in the attached top-level context:

$$it(c, c_{tl})(\langle [\rangle \frown exp_{idx} \frown \langle]\rangle) = ((indexExp, ht(c_{tl})(exp_{idx})), \langle\rangle)$$

The mapping ht is given in section 5.3.2; it is referenced here because ht and it are defined mutually recursively.

As a shorthand, for equal local context and top-level context, we use

$$\begin{aligned}&it : CTX \to IT \\ &\quad c \mapsto it(c, c)\end{aligned}$$

5.3 HybEL Expressions

Alphabet. The alphabet of Exp_{HybEL} consists of terminal symbols that denote variable and signal identifiers (as discussed in section 5.2), as well as operations, assignments, and literals. Its full definition is:

$$\begin{aligned}\Sigma_{Exp} = \{&true, false, \forall, \in, \{, \}, \bullet, (,), \exists, \neg, +, -, \cdot, /, \hat{}, \wedge, \vee, ==, \neq \\ &<, \leq, >, \geq, .., ., ,, :=, :\in, |, ', [,], 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ &\cup L \cup \operatorname{ran} id_V \cup \operatorname{ran} id_{V_{local}} \cup \operatorname{ran} id_S\end{aligned}$$

5.3.1 Syntax of HybEL Expressions

The syntax of HybEL expressions is given by a mapping

$$syn : CTX \to \mathcal{P}(\operatorname{seq} \Sigma_{Exp})$$

which is partitioned into several sub-mappings. These represent different kinds of expressions, reflecting the applications of expressions within a HybridUML specification as discussed at the beginning of this chapter: (1) Constant expressions, providing values of types $t \in DT$. (2) Differential expressions evaluating derivatives of variables of type *anaReal*. (3) Assignment expressions that modify the valuation of a variable. (4) Integer set expressions that calculate finite sets of integers. (5) Special index expressions that assign an index value. (6) Signal raise statements that send signals. (7) Trigger expressions that receive signals and potentially assign variables from the signal's parameters.

$$\begin{aligned}syn(c) = {}&syn_{const}(c) \cup syn_{diff}(c) \cup syn_{ass}(c) \cup syn_{iset}(c) \cup syn_{idxass}(c) \\ &\cup syn_{sigraise}(c) \cup syn_{trigger}(c)\end{aligned}$$

The set of all HybEL expressions which can occur within a given HybridUML specification is then $Exp_{HybEL} = \operatorname{ran} syn$.

Constant Expressions. Constant expressions calculate a value of dedicated type, without modifying variables. They are distinguished by the resulting type:

$$\begin{array}{lll} & \multicolumn{2}{l}{syn_{const} : CTX \rightarrow \mathcal{P}(\mathrm{seq}\,\Sigma_{Exp})} \\ c & \mapsto & syn_{bool}(c) \cup syn_{int}(c) \cup syn_{real}(c) \cup \\ & & \displaystyle\bigcup_{t \in DT_{enum}} syn_{enum}(c,t) \cup \bigcup_{t \in DT_{struc}} syn_{struc}(c,t) \end{array}$$

Boolean expressions are constant expressions which calculate boolean results. They can be defined by:
(1) Boolean literals.
(2) Identifiers of boolean variables.
(3) Unary operation: negation.
(4) Binary operations on boolean operands.
(5) Binary operations on numeric operands.
(6) Binary operations on enumeration-typed operands.
(7) Quantified boolean expression: A bound expression is evaluated for a finite set of integer values. The conjunction ($\forall$) or disjunction ($\exists$) determines the result. A bound variable is used for access to the integer values. Example: $\forall\, i \in \{1..9\} \bullet (x[i] \leq 5)$

$$\begin{array}{lll} & \multicolumn{2}{l}{syn_{bool} : CTX \rightarrow \mathcal{P}(\mathrm{seq}\,\Sigma_{Exp})} \\ c & \mapsto & \{true, false\} \\ & \cup & \{exp \in syn_{id}(c) \mid type_{IdTree}(it(c)(exp)) = bool\} \\ & \cup & \{\langle\neg\rangle \frown exp \mid exp \in syn_{bool}(c)\} \\ & \cup & \{exp_l \frown \langle\Diamond\rangle \frown exp_r \mid exp_l, exp_r \in syn_{bool}(c) \wedge \Diamond \in \{\wedge, \vee, ==, \neq\}\} \\ & \cup & \{exp_l \frown \langle\Diamond\rangle \frown exp_r \mid \\ & & \quad exp_l, exp_r \in syn_{int}(c) \cup syn_{real} \wedge \Diamond \in \{<, \leq, >, \geq, ==, \neq\}\} \\ & \cup & \{exp_l \frown \langle\Diamond\rangle \frown exp_r \mid \exists\, t \in DT_{enum} \bullet \\ & & \quad (exp_l, exp_r \in syn_{enum}(c,t)) \wedge \Diamond \in \{==, \neq\}\} \\ & \cup & \{\langle q, id, \in, \{\rangle \frown exp_{iset} \frown \langle\}, \bullet, (\rangle \frown exp_{bound} \frown \langle)\rangle \mid \\ & & \quad q \in \{\forall, \exists\} \wedge id \in \mathrm{ran}\, id_{V_{local}} \wedge \langle id\rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\ & & \quad \wedge \exists\, v \in V_{local} \bullet (exp_{bound} \in syn_{bool}(var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)) \\ & & \quad \wedge id = id_{V_{local}}(v))\} \end{array}$$

Integer expressions are constant expressions of integer type:
(1) Integer literals.
(2) Identifiers of integer variables.
(3) Binary operations on numeric operands.

$$\begin{array}{lll} & \multicolumn{2}{l}{syn_{int} : CTX \rightarrow \mathcal{P}(\mathrm{seq}\,\Sigma_{Exp})} \\ c & \mapsto & \{s \frown \langle d_1, \ldots, d_n\rangle \mid s \in \{\langle\rangle, \langle -\rangle\} \wedge \forall\, i \in \{1..n\} \bullet \\ & & \quad d_i \in \{0,1,2,3,4,5,6,7,8,9\}\} \\ & \cup & \{exp \in syn_{id}(c) \mid type_{IdTree}(it(c)(exp)) = int\} \\ & \cup & \{exp_l \frown \langle\Diamond\rangle \frown exp_r \mid exp_l, exp_r \in syn_{int}(c) \wedge \Diamond \in \{+, -, \cdot, /, \hat{}\ \}\} \end{array}$$

Real expressions are constant expressions that provide real-valued results:

(1) Real-valued literals.
(2) Identifiers of real or analog real variables.
(3) Binary operations on numeric operands.

$$\begin{array}{ll} & \mathit{syn}_{\mathit{real}} : \mathit{CTX} \to \mathcal{P}(\operatorname{seq} \Sigma_{\mathit{Exp}}) \\ c \mapsto & \{s \frown \langle d_1, \ldots, d_k, ., d_{k+1}, d_n \rangle \mid s \in \{\langle\rangle, \langle - \rangle\} \wedge \forall\, i \in \{1..n\} \bullet \\ & \quad d_i \in \{0,1,2,3,4,5,6,7,8,9\}\} \\ \cup & \{\mathit{exp} \in \mathit{syn}_{\mathit{id}}(c) \mid \mathit{type}_{\mathit{IdTree}}(\mathit{it}(c)(\mathit{exp})) \in \{\mathit{real}, \mathit{anaReal}\}\} \\ \cup & \{\mathit{exp}_l \frown \langle \Diamond \rangle \frown \mathit{exp}_r \mid (\mathit{exp}_l, \mathit{exp}_r) \in (\mathit{syn}_{\mathit{int}}(c) \times \mathit{syn}_{\mathit{real}}(c)) \cup \\ & \quad (\mathit{syn}_{\mathit{real}}(c) \times \mathit{syn}_{\mathit{int}}(c)) \cup (\mathit{syn}_{\mathit{real}}(c) \times \mathit{syn}_{\mathit{real}}(c)) \\ & \quad \wedge \Diamond \in \{+, -, \cdot, /, \hat{}\}\} \end{array}$$

Enumeration type expressions are either literals or identifiers of enumeration type. They are distinguished by their concrete data types:

$$\begin{array}{ll} & \mathit{syn}_{\mathit{enum}} : \mathit{CTX} \times \mathit{DT}_{\mathit{enum}} \to \mathcal{P}(\operatorname{seq} \Sigma_{\mathit{Exp}}) \\ (c, t) \mapsto & L \\ \cup & \{\mathit{exp} \in \mathit{syn}_{\mathit{id}}(c) \mid \mathit{type}_{\mathit{IdTree}}(\mathit{it}(c)(\mathit{exp})) = t\} \end{array}$$

Structured data type expressions are either structured literals or identifiers of structured data type. For type compatibility of literals, helper mappings are defined. The mapping *typeseq*$_{DT}$ maps structured data types to their respective list of contained types, which are derived from their properties – each property is first expanded to n copies according to its multiplicity, then all property copies are concatenated:

$$\begin{array}{l} \mathit{typeseq}_{DT} : \mathit{CTX} \times \mathit{DT} \to \operatorname{seq} \mathit{DT} \\ \quad (c, t) \mapsto s_1 \frown \ldots \frown s_{|\mathit{varseq}_{DT}(t)|} \\ \quad ; \mathit{kind}_{DT}(t) = \mathit{struc} \wedge \forall\, i \in \{1..|\mathit{varseq}_{DT}(t)|\} \bullet \\ \quad (\mathit{eval}_{\varnothing}(c)(\mathit{mult}_V(\mathit{varseq}_{DT}(t)(i))) \in \mathbb{N} \wedge s_i = \\ \quad \mathit{unroll}_{DT \times \mathbb{N}}((\mathit{type}_V(\mathit{varseq}_{DT}(t)(i)), \\ \quad \mathit{eval}_{\varnothing}(c)(\mathit{mult}_V(\mathit{varseq}_{DT}(t)(i))))) \\ \quad (c, t) \mapsto \langle\rangle; \text{else} \end{array}$$

The expansion of properties according to their multiplicities is defined by the mapping

$$\begin{array}{l} \mathit{unroll}_{DT \times \mathbb{N}} : \mathit{DT} \times \mathbb{N} \to \operatorname{seq} \mathit{DT} \\ \quad (t, n) \mapsto s_1 \frown \ldots \frown s_n;\ \forall\, i \in \{1..n\} \bullet s_i = \langle t \rangle \end{array}$$

The syntax of structured data type expressions then is defined as the union of all literals and identifiers of variables which are compatible to a given type t.[4] Note that structured literals can contain arbitrary constant expressions (of appropriate type), not only literals themselves.
(1) An empty literal fits to structured data types t which have no properties.

[4]Note that the semantics $\mathit{eval}_{\varnothing}$ (from section 5.4) is used in this definition. The parsing algorithm therefore ensures syntactical correctness of structured data type literals using a separate parser invocation.

(2) Literals with exactly one property fit to type t, iff t contains exactly one property of the same type. Examples are: $\{17\}$, $\{id_{sdt}.p_1\}$, $\{\{3.7, x\}\}$
(3) Literals with several properties fit to type t, iff t has the same number and types of properties. Examples are: $\{3.7, x\}$, $\{\{x, id_{sdt}.p_2\}, 99, id_{sdt}.p_2\}$
(4) Identifiers of type t fit to type t.

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{syn_{struc} : CTX \times DT_{struc} \to \mathcal{P}(\mathrm{seq}\,\Sigma_{Exp})} \\
(c,t) & \mapsto & \{\langle\{,\}\rangle \mid |typeseq_{DT}(c,t)| = 0\} \\
 & \cup & \{\langle\{\rangle \frown exp \frown \langle\}\rangle \mid |typeseq_{DT}(c,t)| = 1 \wedge \\
 & & ((typeseq_{DT}(c,t)(1) = bool \Rightarrow exp \in syn_{bool}(c)) \\
 & & \wedge(typeseq_{DT}(c,t)(1) = int \Rightarrow exp \in syn_{int}(c)) \\
 & & \wedge(typeseq_{DT}(c,t)(1) \in \{real, anaReal\} \Rightarrow exp \in syn_{real}(c)) \\
 & & \wedge(typeseq_{DT}(c,t)(1) \in DT_{enum} \\
 & & \ \Rightarrow exp \in syn_{enum}(c, typeseq_{DT}(c,t)(1))) \\
 & & \wedge(typeseq_{DT}(c,t)(1) \in DT_{struc} \\
 & & \ \Rightarrow exp \in syn_{struc}(c, typeseq_{DT}(c,t)(1))))\} \\
 & \cup & \{\langle\{\rangle \frown exp_1 \frown \langle,\rangle \frown \ldots \frown \langle,\rangle \frown exp_n \frown \langle\}\rangle \mid n = |typeseq_{DT}(c,t)| \\
 & & \wedge \forall\, i \in \{1..n\} \bullet \\
 & & ((typeseq_{DT}(c,t)(i) = bool \Rightarrow exp_i \in syn_{bool}(c)) \\
 & & \wedge(typeseq_{DT}(c,t)(i) = int \Rightarrow exp_i \in syn_{int}(c)) \\
 & & \wedge(typeseq_{DT}(c,t)(i) \in \{real, anaReal\} \Rightarrow exp_i \in syn_{real}(c)) \\
 & & \wedge(typeseq_{DT}(c,t)(i) \in DT_{enum} \\
 & & \ \Rightarrow exp_i \in syn_{enum}(c, typeseq_{DT}(c,t)(i))) \\
 & & \wedge(typeseq_{DT}(c,t)(i) \in DT_{struc} \\
 & & \ \Rightarrow exp_i \in syn_{struc}(c, typeseq_{DT}(c,t)(i))))\} \\
 & \cup & \{exp \in syn_{id}(c) \mid type_{IdTree}(it(c)(exp)) = t\}
\end{array}
$$

Differential Expressions. Differential expressions are special constant expressions that refer to derivatives of variables:
(1) Derivative specification for an identifier of an analog real variable v. This shall provide the current slope of v wrt. time, i.e. v is interpreted as a function $v : time \to anaReal$, such that the expression v' denotes $\dot{v}(t)$.
(2) Operations which themselves contain differential expressions. This allows to combine derivative access and usual variable access.

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{syn_{diff} : CTX \to \mathcal{P}(\mathrm{seq}\,\Sigma_{Exp})} \\
c & \mapsto & \{exp \frown \langle'\rangle \mid exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = anaReal\} \\
 & \cup & \{exp_l \frown \langle\Diamond\rangle \frown exp_r \mid (exp_l, exp_r) \in (syn_{diff}(c) \times syn_{diff}(c)) \cup \\
 & & (syn_{diff}(c) \times syn_{real}(c)) \cup (syn_{real}(c) \times syn_{diff}(c)) \cup \\
 & & (syn_{diff}(c) \times syn_{int}(c)) \cup (syn_{int}(c) \times syn_{diff}(c)) \\
 & & \wedge \Diamond \in \{+, -, \cdot, /, \hat{}\}\}
\end{array}
$$

Assignment Expressions. Assignments *modify* the valuation of variables. They consist of an identifier of a writable variable of type t and an expression that calculates a compatible value. Assignments are distinguished by the type of variables that are assigned. Additionally, assignments to and from derivatives of variables are treated separately:

$$\begin{aligned}
&\mathit{syn}_{\mathit{ass}} : \mathit{CTX} \to \mathcal{P}(\operatorname{seq}\Sigma_{\mathit{Exp}})\\
c \quad \mapsto \quad & \mathit{syn}_{\mathit{ass,bool}}(c) \cup \mathit{syn}_{\mathit{ass,int}}(c) \cup \mathit{syn}_{\mathit{ass,real}}(c) \cup \mathit{syn}_{\mathit{ass,anaReal}}(c) \cup\\
& \bigcup_{t \in DT_{\mathit{enum}}} \mathit{syn}_{\mathit{ass,enum}}(c,t) \cup \bigcup_{t \in DT_{\mathit{struc}}} \mathit{syn}_{\mathit{ass,struc}}(c,t)\\
& \cup \mathit{syn}_{\mathit{ass,diff}}(c)
\end{aligned}$$

Boolean assignments assign a writable boolean variable from a boolean expression. This may be quantified by an *assignment group*, which provides a finite set of integers for which the assignment is done. For example, $\forall\, i \in \{0..1, 5..6\} := (\mathit{id}_{\mathit{bool}}[i] := x[i] \geq 37)$ is a shortcut for the assignments $\mathit{id}_{\mathit{bool}}[0] := x[0] \geq 37$, $\mathit{id}_{\mathit{bool}}[1] := x[1] \geq 37$, $\mathit{id}_{\mathit{bool}}[5] := x[5] \geq 37$, and $\mathit{id}_{\mathit{bool}}[6] := x[6] \geq 37$.

$$\begin{aligned}
&\mathit{syn}_{\mathit{ass,bool}} : \mathit{CTX} \to \mathcal{P}(\operatorname{seq}\Sigma_{\mathit{Exp}})\\
c \quad \mapsto \quad & \{\mathit{exp}_l \frown \langle{:=}\rangle \frown \mathit{exp}_r \mid \mathit{exp}_l \in \mathit{syn}_{\mathit{id}}(c)\\
& \quad \wedge \mathit{type}_{\mathit{IdTree}}(\mathit{it}(c)(\mathit{exp}_l)) = \mathit{bool} \wedge \mathit{acc}_{\mathit{IdTree}}(\mathit{it}(c)(\mathit{exp}_l)) = \mathit{rw}\\
& \quad \wedge \mathit{exp}_r \in \mathit{syn}_{\mathit{bool}}(c)\}\\
\cup \quad & \{\langle \forall, \mathit{id}, \in, \{\rangle \frown \mathit{exp}_{\mathit{iset}} \frown \langle \}, :=, (\rangle \frown \mathit{exp}_{\mathit{bound}} \frown \langle)\rangle \mid\\
& \quad \mathit{id} \in \operatorname{ran} \mathit{id}_{V_{\mathit{local}}} \wedge \langle \mathit{id}\rangle \notin \mathit{syn}_{\mathit{id}}(c) \wedge \mathit{exp}_{\mathit{iset}} \in \mathit{syn}_{\mathit{iset}}(c)\\
& \quad \wedge \exists\, v \in V_{\mathit{local}} \bullet (\mathit{exp}_{\mathit{bound}} \in \mathit{syn}_{\mathit{ass,bool}}(\mathit{var}_{\mathit{CTX}}(c) \cup \{v\}, \mathit{sig}_{\mathit{CTX}}(c))\\
& \quad \wedge \mathit{id} = \mathit{id}_{V_{\mathit{local}}}(v))\}
\end{aligned}$$

Integer assignments assign a writable integer variable from an integer expression, similarly to boolean assignments. Additionally, a *non-deterministic assignment* from a finite set of integers can be given, that is constrained by a boolean expression. As an example, $\mathit{id}_{\mathit{int}} :\in \{i \in \{3..8\} \mid \mathit{id}_{\mathit{bool}}[i] \vee \mathit{id}_{\mathit{bool}}[i+1]\}$ first determines the subset $s \subseteq \{3,4,5,6,7,8\}$ for which the given boolean expression holds, and then chooses one of the elements for assignment to $\mathit{id}_{\mathit{int}}$.

$$\begin{aligned}
&\mathit{syn}_{\mathit{ass,int}} : \mathit{CTX} \to \mathcal{P}(\operatorname{seq}\Sigma_{\mathit{Exp}})\\
c \quad \mapsto \quad & \{\mathit{exp}_l \frown \langle{:=}\rangle \frown \mathit{exp}_r \mid \mathit{exp}_l \in \mathit{syn}_{\mathit{id}}(c)\\
& \quad \wedge \mathit{type}_{\mathit{IdTree}}(\mathit{it}(c)(\mathit{exp}_l)) = \mathit{int} \wedge \mathit{acc}_{\mathit{IdTree}}(\mathit{it}(c)(\mathit{exp}_l)) = \mathit{rw}\\
& \quad \wedge \mathit{exp}_r \in \mathit{syn}_{\mathit{int}}(c)\}\\
\cup \quad & \{\langle \forall, \mathit{id}, \in, \{\rangle \frown \mathit{exp}_{\mathit{iset}} \frown \langle \}, :=, (\rangle \frown \mathit{exp}_{\mathit{bound}} \frown \langle)\rangle \mid\\
& \quad \mathit{id} \in \operatorname{ran} \mathit{id}_{V_{\mathit{local}}} \wedge \langle \mathit{id}\rangle \notin \mathit{syn}_{\mathit{id}}(c) \wedge \mathit{exp}_{\mathit{iset}} \in \mathit{syn}_{\mathit{iset}}(c)\\
& \quad \wedge \exists\, v \in V_{\mathit{local}} \bullet (\mathit{exp}_{\mathit{bound}} \in \mathit{syn}_{\mathit{ass,int}}(\mathit{var}_{\mathit{CTX}}(c) \cup \{v\}, \mathit{sig}_{\mathit{CTX}}(c))\\
& \quad \wedge \mathit{id} = \mathit{id}_{V_{\mathit{local}}}(v))\}\\
\cup \quad & \{\mathit{exp}_{\mathit{var}} \frown \langle :\in, \{, \mathit{id}, \in, \{\rangle \frown \mathit{exp}_{\mathit{iset}} \frown \langle \}, |\rangle \frown \mathit{exp}_{\mathit{bound}} \frown \langle \}\rangle \mid\\
& \quad \mathit{exp}_{\mathit{var}} \in \mathit{syn}_{\mathit{id}}(c) \wedge \mathit{type}_{\mathit{IdTree}}(\mathit{it}(c)(\mathit{exp}_{\mathit{var}})) = \mathit{int}\\
& \quad \wedge \mathit{acc}_{\mathit{IdTree}}(\mathit{it}(c)(\mathit{exp}_{\mathit{var}})) = \mathit{rw} \wedge \mathit{id} \in \operatorname{ran} \mathit{id}_{V_{\mathit{local}}}\\
& \quad \wedge \langle \mathit{id}\rangle \notin \mathit{syn}_{\mathit{id}}(c) \wedge \mathit{exp}_{\mathit{iset}} \in \mathit{syn}_{\mathit{iset}}(c)\\
& \quad \wedge \exists\, v \in V_{\mathit{local}} \bullet (\mathit{exp}_{\mathit{bound}} \in \mathit{syn}_{\mathit{bool}}(\mathit{var}_{\mathit{CTX}}(c) \cup \{v\}, \mathit{sig}_{\mathit{CTX}}(c))\\
& \quad \wedge \mathit{id} = \mathit{id}_{V_{\mathit{local}}}(v))\}
\end{aligned}$$

Real assignments assign a writable real variable from a real-valued or integer expression. Similarly to boolean assignments, a quantified assignment group can be specified:

$$\begin{array}{lll} & syn_{ass,real} : CTX \to \mathcal{P}(\mathrm{seq}\,\Sigma_{Exp}) & \\ c & \mapsto & \{exp_l \frown \langle := \rangle \frown exp_r \mid exp_l \in syn_{id}(c) \wedge \\ & & \quad type_{IdTree}(it(c)(exp_l)) = real \\ & & \quad \wedge acc_{IdTree}(it(c)(exp_l)) = rw \wedge exp_r \in syn_{int}(c) \cup syn_{real}(c)\} \\ & \cup & \{\langle \forall, id, \in, \{ \rangle \frown exp_{iset} \frown \langle \}, :=, (\rangle \frown exp_{bound} \frown \langle) \rangle \mid \\ & & \quad id \in \mathrm{ran}\, id_{V_{local}} \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\ & & \quad \wedge \exists\, v \in V_{local} \bullet (exp_{bound} \in syn_{ass,real}(var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)) \\ & & \quad \wedge id = id_{V_{local}}(v))\} \end{array}$$

Analog real assignments assign a writable analog real variable from a real-valued or integer expression. They are defined in the same way as boolean and real assignments are.

$$\begin{array}{lll} & syn_{ass,anaReal} : CTX \to \mathcal{P}(\mathrm{seq}\,\Sigma_{Exp}) & \\ c & \mapsto & \{exp_l \frown \langle := \rangle \frown exp_r \mid exp_l \in syn_{id}(c) \wedge \\ & & \quad type_{IdTree}(it(c)(exp_l)) = anaReal \\ & & \quad \wedge acc_{IdTree}(it(c)(exp_l)) = rw \wedge exp_r \in syn_{int}(c) \cup syn_{real}(c)\} \\ & \cup & \{\langle \forall, id, \in, \{ \rangle \frown exp_{iset} \frown \langle \}, :=, (\rangle \frown exp_{bound} \frown \langle) \rangle \mid \\ & & \quad id \in \mathrm{ran}\, id_{V_{local}} \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\ & & \quad \wedge \exists\, v \in V_{local} \bullet (exp_{bound} \in \\ & & \qquad syn_{ass,anaReal}(var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)) \\ & & \quad \wedge id = id_{V_{local}}(v))\} \end{array}$$

Enumeration-typed assignments assign a writable variable of enumeration type from an expression of the same type. Again, assignment groups are possible.

$$\begin{array}{lll} & syn_{ass,enum} : CTX \times DT_{enum} \to \mathcal{P}(\mathrm{seq}\,\Sigma_{Exp}) & \\ (c,t) & \mapsto & \{exp_l \frown \langle := \rangle \frown exp_r \mid exp_l \in syn_{id}(c) \wedge \\ & & \quad type_{IdTree}(it(c)(exp_l)) = t \wedge acc_{IdTree}(it(c)(exp_l)) = rw \\ & & \quad \wedge exp_r \in syn_{enum}(c,t)\} \\ & \cup & \{\langle \forall, id, \in, \{ \rangle \frown exp_{iset} \frown \langle \}, :=, (\rangle \frown exp_{bound} \frown \langle) \rangle \mid \\ & & \quad id \in \mathrm{ran}\, id_{V_{local}} \wedge \langle id \rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c) \\ & & \quad \wedge \exists\, v \in V_{local} \bullet (exp_{bound} \in \\ & & \qquad syn_{ass,enum}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)), t) \\ & & \quad \wedge id = id_{V_{local}}(v))\} \end{array}$$

Structured data type assignments assign a writable variable of structured type from an expression of the same type. Assignment groups are available, too.

$$\begin{array}{lll} & syn_{ass,struc} : CTX \times DT_{struc} \to \mathcal{P}(\mathrm{seq}\,\Sigma_{Exp}) & \\ (c,t) & \mapsto & \{exp_l \frown \langle := \rangle \frown exp_r \mid exp_l \in syn_{id}(c) \wedge \\ & & \quad type_{IdTree}(it(c)(exp_l)) = t \wedge acc_{IdTree}(it(c)(exp_l)) = rw \end{array}$$

$$\begin{aligned}
&\quad \wedge exp_r \in syn_{struc}(c,t)\}\\
\cup &\quad \{\langle \forall, id, \in, \{\rangle ^\frown exp_{iset} ^\frown \langle \}, :=, (\rangle ^\frown exp_{bound} ^\frown \langle)\rangle \mid \\
&\quad id \in \operatorname{ran} id_{V_{local}} \wedge \langle id\rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c)\\
&\quad \wedge \exists\, v \in V_{local} \bullet (exp_{bound} \in \\
&\qquad syn_{ass,struc}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)), t)\\
&\quad \wedge id = id_{V_{local}}(v))\}
\end{aligned}$$

Differential Assignments. Differential assignments assign a writable analog real variable. In contrast to analog real assignments, there must be at least one subexpression which contains derivatives of analog real variables: (1) The right-hand side can contain derivatives of analog real variables $v_1, \ldots, v_n$, therefore the calculated value is calculated from their evolutions, e.g. $v := v_1' + v_2' + v_3$. (2) The left-hand side can be the derivative of a writable analog real variable v, therefore the right-hand side determines the relative change of v. Examples: $v' := 2 \cdot v_1$, $v' := 2 \cdot v_1'$ As with all other assignments, quantification by assignment groups is possible.

$$\begin{aligned}
&syn_{ass,diff} : CTX \to \mathcal{P}(\operatorname{seq} \Sigma_{Exp})\\
c \quad \mapsto &\quad \{exp_l ^\frown \langle := \rangle ^\frown exp_r \mid exp_l \in syn_{id}(c)\\
&\quad \wedge type_{IdTree}(it(c)(exp_l)) = anaReal\\
&\quad \wedge acc_{IdTree}(it(c)(exp_l)) = rw \wedge exp_r \in syn_{diff}(c)\}\\
\cup &\quad \{exp_l ^\frown \langle ', := \rangle ^\frown exp_r \mid exp_l \in syn_{id}(c) \wedge\\
&\quad type_{IdTree}(it(c)(exp_l)) = anaReal \wedge acc_{IdTree}(it(c)(exp_l)) = rw\\
&\quad \wedge exp_r \in syn_{diff}(c) \cup syn_{real}(c) \cup syn_{int}(c)\}\\
\cup &\quad \{\langle \forall, id, \in, \{\rangle ^\frown exp_{iset} ^\frown \langle \}, :=, (\rangle ^\frown exp_{bound} ^\frown \langle)\rangle \mid\\
&\quad id \in \operatorname{ran} id_{V_{local}} \wedge \langle id\rangle \notin syn_{id}(c) \wedge exp_{iset} \in syn_{iset}(c)\\
&\quad \wedge \exists\, v \in V_{local} \bullet (exp_{bound} \in syn_{ass,diff}(var_{CTX}(c) \cup \{v\}, sig_{CTX}(c))\\
&\quad \wedge id = id_{V_{local}}(v))\}
\end{aligned}$$

Integer Sets. An integer set expression determines a finite set of integers. It consists of a set of integer ranges, e.g. 1..7, 27..39 identifies the set $s_{\mathbb{Z}} = \{1..7\} \cup \{27..39\}$.

$$\begin{aligned}
&syn_{iset} : CTX \to \mathcal{P}(\operatorname{seq} \Sigma_{Exp})\\
c \quad \mapsto &\quad \{exp_l ^\frown \langle .. \rangle ^\frown exp_r \mid exp_l, exp_r \in syn_{int}(c)\}\\
\cup &\quad \{exp_l ^\frown \langle .. \rangle ^\frown exp_r ^\frown \langle , \rangle ^\frown exp_{iset} \mid\\
&\quad exp_l, exp_r \in syn_{int}(c) \wedge exp_{iset} \in syn_{iset}(c)\}
\end{aligned}$$

Index Assignments. An index assignment expression identifies a writable variable of type *int* that is supposed to be assigned with an index value (that is always of integer type). It is only applicable as index specification for received signals, see section 5.2.1. Example: $:= id_{int,rw}$

$$\begin{aligned}
&syn_{idxass} : CTX \to \mathcal{P}(\operatorname{seq} \Sigma_{Exp})\\
c \quad \mapsto &\quad \{\langle := \rangle ^\frown exp \mid exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = int\\
&\quad \wedge acc_{IdTree}(it(c)(exp)) = rw\}
\end{aligned}$$

Signal Raise Statements. Signal raise statements denote a signal to be raised, optionally equipped with a parameter list, defined by constant expressions.

(1) A signal without parameters is raised by use of a simple identifier that corresponds to a sendable signal, e.g.: $id_{sig,send}$
(2) Alternatively, the empty parameter list can be explicitly given: $id_{sig,send}()$
(3) Arbitrary constant expressions can be used for the actual signal parameter. Examples are: $id_{sig,send}(17)$, $id_{sig,send}(v_1 - v_2)$.
(4) Several parameters are given in the usual way: $id_{sig,send}(p_1, p_2, p_3)$

$$\begin{array}{lll}
\multicolumn{3}{l}{syn_{sigraise} : CTX \rightarrow \mathcal{P}(\operatorname{seq} \Sigma_{Exp})} \\
c & \mapsto & \{exp \in syn_{id}(c) \mid type_{IdTree}(it(c)(exp)) = sendSig\} \\
& \cup & \{exp ^\frown \langle (,) \rangle \mid exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = sendSig\} \\
& \cup & \{exp ^\frown \langle (\rangle ^\frown pexp ^\frown \langle) \rangle \mid exp \in syn_{id}(c) \\
& & \wedge type_{IdTree}(it(c)(exp)) = sendSig \wedge pexp \in syn_{const}(c)\} \\
& \cup & \{exp ^\frown \langle (\rangle ^\frown pexp_1 ^\frown \langle , \rangle ^\frown \ldots ^\frown \langle , \rangle ^\frown pexp_n ^\frown \langle) \rangle \mid \\
& & exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = sendSig \\
& & \wedge \forall\, i \in \{1..n\} \bullet pexp_i \in syn_{const}(c)\}
\end{array}$$

Trigger Expressions. Trigger statements define a signal to be received, optionally equipped with a parameter list, which consists of writable variables for parameter reception.

(1) A trigger without parameters is specified by use of a simple identifier which denotes a receivable signal, e.g.: $id_{sig,recv}$
(2) Alternatively, the empty parameter list can be explicitly given: $id_{sig,recv}()$
(3) The formal signal parameter is given by an identifier of a writable variable, e.g.: $id_{sig,recv}(v_{rw})$.
(4) Several parameters are given in the usual way: $id_{sig,recv}(v_{1,rw}, v_{2,rw}, v_{3,rw})$

$$\begin{array}{lll}
\multicolumn{3}{l}{syn_{trigger} : CTX \rightarrow \mathcal{P}(\operatorname{seq} \Sigma_{Exp})} \\
c & \mapsto & \{exp \in syn_{id}(c) \mid type_{IdTree}(it(c)(exp)) = recvSig\} \\
& \cup & \{exp ^\frown \langle (,) \rangle \mid exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = recvSig\} \\
& \cup & \{exp ^\frown \langle (\rangle ^\frown pexp ^\frown \langle) \rangle \mid exp \in syn_{id}(c) \\
& & \wedge type_{IdTree}(it(c)(exp)) = recvSig \wedge pexp \in syn_{id}(c) \\
& & \wedge type_{IdTree}(it(c)(pexp)) \in DT \wedge acc_{IdTree}(it(c)(pexp)) = rw\} \\
& \cup & \{exp ^\frown \langle (\rangle ^\frown pexp_1 ^\frown \langle , \rangle ^\frown \ldots ^\frown \langle , \rangle ^\frown pexp_n ^\frown \langle) \rangle \mid \\
& & exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = recvSig \\
& & \wedge \forall\, i \in \{1..n\} \bullet (pexp_i \in syn_{id}(c) \wedge type_{IdTree}(it(c)(pexp_i)) \in DT \\
& & \wedge acc_{IdTree}(it(c)(pexp_i)) = rw)\}
\end{array}$$

5.3.2 Intermediate Semantics of HybEL Expressions

Expressions $exp \in Exp_{HybEL}$ are mapped to trees of *hybel items*, which are composed of a *role* and a *value*:

$$HybelItem = HybelItemRole \times HybelItemVal$$

The role of the hybel item is roughly one of operation, assignment, literal, enumeration data type, variable, signal raise statement, trigger, read-only parameter expression, or write-only parameter expression. It also defines the *type* of the item.

$$\begin{aligned}
&\mathit{HybelItemOpRole} = \{(t, op) \mid t \in \{bool, int, real\}\} \cup \{(anaReal, \mathit{diffOp})\}\\
&\quad \cup\{(bool, q) \mid q \in \{\forall, \exists\}\} \cup \{(intSet, s) \mid s \in \{intSpecs, intRange\}\}\\
&\mathit{HybelItemVarRole} = \{(t, var) \mid t \in DT\} \cup \{(anaReal, derivVar)\}\\
&\mathit{HybelItemLitRole} = \{(t, lit) \mid t \in DT_{enum} \cup \{bool, int, real\} \cup \{sdt_{anon}\}\}\\
&\mathit{HybelItemAssRole} = \{(t, ass) \mid t \in DT\} \cup \{(t, assGroup) \mid t \in DT\}\cup\\
&\quad \{(anaReal, \mathit{diffAss}), (int, intNondetAss), (anaReal, \mathit{diffAssGroup}),\\
&\quad (int, indexAss)\}\\
&\mathit{HybelItemSigRole} = \{(sigtype, d) \mid d \in \{sendSig, recvSig\}\}\\
&\mathit{HybelItemRole} = \mathit{HybelItemOpRole} \cup \mathit{HybelItemVarRole}\\
&\quad \cup \mathit{HybelItemLitRole} \cup \mathit{HybelItemAssRole} \cup \mathit{HybelItemSigRole}
\end{aligned}$$

Depending on the role of the hybel item, an *item value* can be attached. The possible values are operation identifiers, boolean values, integer and real numbers, enumeration literals, as well as trees of *identifier items*. Identifier items represent variables or signals, respectively, which are given by the identifiers from the expression, as discussed in section 5.2.2.

The special value λ denotes the absence of an item value.

$$\begin{aligned}
&\mathit{HybelItemVal} = \{\cdot, /, +, -, \neg, \wedge, \vee, ==, \neq, \leq, <, \geq, >, \hat{}\,\}\\
&\quad \cup \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup L\\
&\quad \cup \mathrm{tree_o}\ \mathit{IdItem}\\
&\quad \cup \{\lambda\}
\end{aligned}$$

The mapping from expressions to hybel item trees depends on the provided context, similarly to identifier item trees (see section 5.2.2). For each possible context, a mapping from expressions to hybel item trees exists:

$$\begin{aligned}
&HT = \bigcup_{c \in CTX} (syn(c) \to \mathrm{tree_o}\ \mathit{HybelItem})\\
&ht : CTX \to HT
\end{aligned}$$

Literals of primitive data types are mapped to the respective value:

$$\begin{aligned}
&ht(c)(\langle lit_{bool}\rangle) = (((bool, lit), lit_{bool}), \langle\rangle)\\
&\quad ;\ lit_{bool} \in \{true, false\}\\
&ht(c)(\langle d_1, \ldots, d_n\rangle) = (((int, lit), d_1 \ldots d_n), \langle\rangle)\\
&\quad ;\ \forall\, i \in \{1..n\} \bullet d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\\
&ht(c)(\langle -, d_1, \ldots, d_n\rangle) = (((int, lit), -d_1 \ldots d_n), \langle\rangle)\\
&\quad ;\ \forall\, i \in \{1..n\} \bullet d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\\
&ht(c)(\langle d_1, \ldots, d_{k-1}, ., d_k, \ldots, d_n\rangle) = (((real, lit), d_1 \ldots d_{k-1}.d_k \ldots d_n), \langle\rangle)\\
&\quad ;\ \forall\, i \in \{1..n\} \bullet d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\\
&ht(c)(\langle -, d_1, \ldots, d_{k-1}, ., d_k, \ldots, d_n\rangle) = (((real, lit), -d_1 \ldots d_{k-1}.d_k \ldots d_n), \langle\rangle)\\
&\quad ;\ \forall\, i \in \{1..n\} \bullet d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
\end{aligned}$$

Literals of enumeration data types are contained themselves in the hybel item:

$$\begin{aligned}&ht(c)(\langle lit_{enum}\rangle) = (((dt_L(lit_{enum}), lit), lit_{enum}), \langle\rangle)\\ &\quad ;\ lit_{enum} \in L\end{aligned}$$

Structured data type literals are recursively mapped to a tree representing the contained expressions:

$$\begin{aligned}&ht(c)(\langle \{,\}\rangle) = (((sdt_{anon}, lit), \lambda), \langle\rangle)\\ &ht(c)(\langle \{\rangle \frown exp \frown \langle\}\rangle) = (((sdt_{anon}, lit), \lambda), \langle ht(c)(exp)\rangle)\\ &\quad ;\ exp \in syn_{const}(c)\\ &ht(c)(\langle \{\rangle \frown exp_1 \frown \langle ,\rangle \frown \ldots \frown \langle ,\rangle \frown exp_n \frown \langle\}\rangle) =\\ &\quad (((sdt_{anon}, lit), \lambda), \langle ht(c)(exp_1), \ldots, ht(c)(exp_n)\rangle)\\ &\quad ;\ \forall\, i \in \{1..n\} \bullet exp_i \in syn_{const}(c)\end{aligned}$$

Variable identifiers are mapped to the identifier item tree that represents the respective variable:

$$\begin{aligned}&ht(c)(exp) = (((type_{IdTree}(it(c)(exp)), var), it(c)(exp)), \langle\rangle)\\ &\quad ;\ exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) \in DT\end{aligned}$$

Signal identifiers are mapped to the identifier item tree that represents the respective signal, potentially including parameters:

$$\begin{aligned}&ht(c)(exp) = (((sigtype, type_{IdTree}(it(c)(exp))), it(c)(exp)), \langle\rangle)\\ &\quad ;\ exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) \in \{recvSig, sendSig\}\\ &ht(c)(exp \frown \langle (\rangle \frown pexp \frown \langle)\rangle)\\ &\quad = (((sigtype, type_{IdTree}(it(c)(exp))), it(c)(exp)), \langle ht(c)(pexp)\rangle)\\ &\quad ;\ exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) \in \{recvSig, sendSig\}\\ &\qquad \wedge pexp \in syn_{const}(c) \cup syn_{id}(c)\\ &ht(c)(exp \frown \langle (\rangle \frown pexp_1 \frown \langle ,\rangle \frown \ldots \frown \langle ,\rangle \frown pexp_n \frown \langle)\rangle) =\\ &\quad (((sigtype, type_{IdTree}(it(c)(exp))), it(c)(exp)),\\ &\qquad \langle ht(c)(pexp_1), \ldots, ht(c)(pexp_n)\rangle)\\ &\quad ;\ exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) \in \{recvSig, sendSig\}\\ &\qquad \wedge \forall\, i \in \{1..n\} \bullet pexp_i \in syn_{const}(c) \cup syn_{id}(c)\end{aligned}$$

Derivatives are mapped to the identifier item tree that represents the derived variable, which is always of type *anaReal*.

$$\begin{aligned}&ht(c)(exp \frown \langle '\rangle) = (((anaReal, derivVar), it(c)(exp)), \langle\rangle)\\ &\quad ;\ exp \in syn_{id}(c) \wedge type_{IdTree}(it(c)(exp)) = anaReal\end{aligned}$$

Simple operations with boolean result are recursively mapped to a tree that represents the operation, containing a subtree for each operand.[5]

$$\begin{aligned}&ht(c)(\langle \neg\rangle \frown exp) = (((bool, op), \neg), \langle ht(c)(exp)\rangle)\\ &ht(c)(exp_1 \frown \langle \Diamond\rangle \frown exp_2) = (((bool, op), \Diamond), \langle ht(c)(exp_1), ht(c)(exp_2)\rangle)\\ &\quad ;\ \Diamond \in \{==, \wedge, \vee, \neq, <, \leq, >, \geq\}\end{aligned}$$

[5]The usual operator precedence is guaranteed by the parsing algorithm. It is omitted in this presentation.

Simple operations with numeric result are recursively mapped to a tree that represents the operation, containing a subtree for each operand. The role is determined by the roles of the subtrees.[5]

$$\begin{aligned}
&ht(c)(exp_l \frown \langle\Diamond\rangle \frown exp_r) = \\
&\qquad (((int, op), \Diamond), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle) \\
&\qquad ;\ exp_l \frown \langle\Diamond\rangle \frown exp_r \in syn_{int}(c) \\
&\qquad\ \ \wedge \Diamond \in \{+, -, \cdot, /, \hat{}\ \} \\
&ht(c)(exp_l \frown \langle\Diamond\rangle \frown exp_r) = \\
&\qquad (((real, op), \Diamond), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle) \\
&\qquad ;\ exp_l \frown \langle\Diamond\rangle \frown exp_r \in syn_{real}(c) \\
&\qquad\ \ \wedge \Diamond \in \{+, -, \cdot, /, \hat{}\ \} \\
&ht(c)(exp_l \frown \langle\Diamond\rangle \frown exp_r) = \\
&\qquad (((anaReal, diffOp), \Diamond), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle) \\
&\qquad ;\ exp_l \frown \langle\Diamond\rangle \frown exp_r \in syn_{diff}(c) \\
&\qquad\ \ \wedge \Diamond \in \{+, -, \cdot, /, \hat{}\ \}
\end{aligned}$$

Quantified expressions are mapped to an item that holds an identifier tree which represents the bound variable that is introduced by the quantification. There are two children: (1) An expression defining a finite set of integers – these are the values that the bound variable adopts. (2) An expression that shall be evaluated for each value of the bound variable.

$$\begin{aligned}
&ht(c)(\langle \forall, id, \in, \{\rangle \frown exp_{iset} \frown \langle\}, \bullet, (\rangle \frown exp_{bound} \frown \langle)\rangle) = \\
&\qquad (((bool, \forall), it(c)(\langle id\rangle)), \\
&\qquad\ \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle) \\
&\qquad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id \\
&ht(c)(\langle \exists, id, \in, \{\rangle \frown exp_{iset} \frown \langle\}, \bullet, (\rangle \frown exp_{bound} \frown \langle)\rangle) = \\
&\qquad (((bool, \exists), it(c)(\langle id\rangle)), \\
&\qquad\ \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle) \\
&\qquad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id
\end{aligned}$$

An integer set specification is represented by an item that contains a list of integer range expressions:

$$\begin{aligned}
&ht(c)(exp) = (((intSet, intSpecs), \lambda), \langle sexp\rangle) \\
&\qquad ;\ \exists\, exp_l, exp_r \in syn_{int}(c) \bullet \\
&\qquad\ \ (exp = exp_l \frown \langle..\rangle \frown exp_r \\
&\qquad\ \ \wedge sexp = (((intSet, intRange), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle)) \\
&ht(c)(exp_1 \frown \langle,\rangle \frown \ldots \frown \langle,\rangle \frown exp_n) = \\
&\qquad (((intSet, intSpecs), \lambda), \langle sexp_1, \ldots, sexp_n\rangle) \\
&\qquad ;\ \forall\, i \in \{1..n\} \bullet \exists\, exp_l, exp_r \in syn_{int}(c) \bullet \\
&\qquad\ \ (exp_i = exp_l \frown \langle..\rangle \frown exp_r \\
&\qquad\ \ \wedge sexp_i = (((intSet, intRange), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle))
\end{aligned}$$

Simple assignments contain two subexpressions: (1) the left-hand side, consisting of a variable identifier, and (2) the right-hand side, representing a constant expression of appropriate type.

$$\begin{aligned}
&ht(c)(exp_l \frown \langle := \rangle \frown exp_r) = (((bool, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle)\\
&\qquad ;\ exp_l \frown \langle := \rangle \frown exp_r \in syn_{ass,bool}(c)\\
&ht(c)(exp_l \frown \langle := \rangle \frown exp_r) = (((int, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle)\\
&\qquad ;\ exp_l \frown \langle := \rangle \frown exp_r \in syn_{ass,int}(c)\\
&ht(c)(exp_l \frown \langle := \rangle \frown exp_r) = (((real, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle)\\
&\qquad ;\ exp_l \frown \langle := \rangle \frown exp_r \in syn_{ass,real}(c)\\
&ht(c)(exp_l \frown \langle := \rangle \frown exp_r) = (((anaReal, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle)\\
&\qquad ;\ exp_l \frown \langle := \rangle \frown exp_r \in syn_{ass,anaReal}(c)\\
&ht(c)(exp_l \frown \langle := \rangle \frown exp_r) = (((t, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle)\\
&\qquad ;\ exp_l \frown \langle := \rangle \frown exp_r \in syn_{ass,enum}(c, t)\\
&ht(c)(exp_l \frown \langle := \rangle \frown exp_r) = (((t, ass), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle)\\
&\qquad ;\ exp_l \frown \langle := \rangle \frown exp_r \in syn_{ass,struc}(c, t)
\end{aligned}$$

Differential assignments are either assignments to the derivative of a variable, or assignments which are calculated from derivatives of variables:

$$\begin{aligned}
&ht(c)(exp_l \frown \langle := \rangle \frown exp_r) =\\
&\qquad (((anaReal, diffAss), \lambda), \langle ht(c)(exp_l), ht(c)(exp_r)\rangle)\\
&\qquad ;\ exp_l \frown \langle := \rangle \frown exp_r \in syn_{ass,diff}(c)
\end{aligned}$$

A non-deterministic integer assignment is composed of four parts: (1) the variable that is assigned, (2) a bound variable, (3) an integer set specification, and (4) a bound expression:

$$\begin{aligned}
&ht(c)(exp_{var} \frown \langle :\in, \{, id, \in, \{\rangle \frown exp_{iset} \frown \langle \}, |\rangle \frown exp_{bound} \frown \langle \}\rangle) =\\
&\qquad (((int, intNondetAss), it(c)(\langle id\rangle)),\\
&\qquad\ \langle ht(c)(exp_{var}), ht(c)(exp_{iset}),\\
&\qquad\ ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle)\\
&\qquad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id
\end{aligned}$$

A group of assignments consists of a quantified assignment with corresponding type:

$$\begin{aligned}
&ht(c)(\langle \forall, id, \in, \{\rangle \frown exp_{iset} \frown \langle \}, :=, (\rangle \frown exp_{bound} \frown \langle)\rangle) =\\
&\qquad (((bool, assGroup), it(c)(\langle id\rangle)),\\
&\qquad\ \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle)\\
&\qquad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id\\
&\qquad\ \wedge exp_{bound} \in syn_{ass,bool}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))\\
&ht(c)(\langle \forall, id, \in, \{\rangle \frown exp_{iset} \frown \langle \}, :=, (\rangle \frown exp_{bound} \frown \langle)\rangle) =\\
&\qquad (((int, assGroup), it(c)(\langle id\rangle)),\\
&\qquad\ \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle)\\
&\qquad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id
\end{aligned}$$

$$
\begin{aligned}
&\quad\quad \wedge exp_{bound} \in syn_{ass,int}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))\\
&ht(c)(\langle \forall, id, \in, \{\rangle \frown exp_{iset} \frown \langle\}, :=, (\rangle \frown exp_{bound} \frown \langle)\rangle) =\\
&\quad (((real, assGroup), it(c)(\langle id \rangle)),\\
&\quad \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle)\\
&\quad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id\\
&\quad\quad \wedge exp_{bound} \in syn_{ass,real}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))\\
&ht(c)(\langle \forall, id, \in, \{\rangle \frown exp_{iset} \frown \langle\}, :=, (\rangle \frown exp_{bound} \frown \langle)\rangle) =\\
&\quad (((anaReal, assGroup), it(c)(\langle id \rangle)),\\
&\quad \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle)\\
&\quad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id\\
&\quad\quad \wedge exp_{bound} \in syn_{ass,anaReal}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))\\
&ht(c)(\langle \forall, id, \in, \{\rangle \frown exp_{iset} \frown \langle\}, :=, (\rangle \frown exp_{bound} \frown \langle)\rangle) =\\
&\quad (((t, assGroup), it(c)(\langle id \rangle)),\\
&\quad \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle)\\
&\quad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id \wedge t \in DT_{enum}\\
&\quad\quad \wedge exp_{bound} \in syn_{ass,enum}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)), t)\\
&ht(c)(\langle \forall, id, \in, \{\rangle \frown exp_{iset} \frown \langle\}, :=, (\rangle \frown exp_{bound} \frown \langle)\rangle) =\\
&\quad (((t, assGroup), it(c)(\langle id \rangle)),\\
&\quad \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle)\\
&\quad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id \wedge t \in DT_{struc}\\
&\quad\quad \wedge exp_{bound} \in syn_{ass,struc}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)), t)
\end{aligned}
$$

A group of differential assignments consists of a quantified differential assignment:

$$
\begin{aligned}
&ht(c)(\langle \forall, id, \in, \{\rangle \frown exp_{iset} \frown \langle\}, :=, (\rangle \frown exp_{bound} \frown \langle)\rangle) =\\
&\quad (((anaReal, diffAssGroup), it(c)(\langle id \rangle)),\\
&\quad \langle ht(c)(exp_{iset}), ht((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))(exp_{bound})\rangle)\\
&\quad ;\ v \in V_{local} \wedge id_{V_{local}}(v) = id \wedge\\
&\quad\quad \wedge exp_{bound} \in syn_{ass,diff}((var_{CTX}(c) \cup \{v\}, sig_{CTX}(c)))
\end{aligned}
$$

An index assignment contains a variable of integer type, such that an index value can be assigned to it:

$$ht(c)(\langle := \rangle \frown exp_{intvar}) = (((int, indexAss), \lambda), \langle ht(c)(exp_{intvar})\rangle)$$

5.4 Skeleton Evaluation Semantics of HybEL Expressions

In this section, a skeleton semantics for the evaluation of HybEL expressions is given. Its main purpose for this chapter is to define the value for multiplicity expressions of sub-properties of structured data types, as it is used in section 5.3.1. The definition of the full evaluation semantics is provided in section 6.2.

The *skeleton evaluation semantics* of hybel expressions is given by

$$\begin{aligned}
&VAL_{eval} = \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup L \cup \text{seq } VAL_{eval} \cup \mathcal{P}(\mathbb{Z})\\
&MAP_{VAL_{eval}} = \bigcup_{c \in CTX} (syn(c) \rightarrow VAL_{eval} \cup \{\lambda\})\\
&eval_{\varnothing} : CTX \rightarrow MAP_{VAL_{eval}}\\
&\qquad eval_{\varnothing}(c)(exp) = eval_{ht,\varnothing}(ht(c)(exp))
\end{aligned}$$

Each hybel item tree is mapped to either a value, or to the special value λ that denotes the absence of a value:[6]

$$eval_{ht,\varnothing} : \text{tree}_{\text{o}}\, HybelItem \rightarrow VAL_{eval} \cup \{\lambda\}$$

Literal values *are* the values:

$$\begin{aligned}
&(((t, lit), l), sub) \mapsto l;\ t \in DT\\
&(((sdt_{anon}, lit), v), \langle t_1, \ldots, t_n \rangle) \mapsto \langle eval_{ht,\varnothing}(t_1), \ldots, eval_{ht,\varnothing}(t_n) \rangle
\end{aligned}$$

Operations are evaluated:

$$\begin{aligned}
&(((t, op), \Diamond), \langle t_1, t_2 \rangle) \mapsto eval_{ht,\varnothing}(t_1) \Diamond eval_{ht,\varnothing}(t_2)\\
&\qquad ;\ t_1, t_2 \in \{int, real, anaReal\} \wedge \Diamond \in \{+, -, \cdot, /, \hat{}, <, \leq, \geq, >\}\\
&\qquad \ \vee t_1 = t_2 \wedge \Diamond \in \{==, \neq\}\\
&\qquad \ \vee t_1, t_2 \in \{bool\} \wedge \Diamond \in \{\wedge, \vee\}\\
&(((t, op), \neg), \langle t_1 \rangle) \mapsto \neg eval_{ht,\varnothing}(t_1)
\end{aligned}$$

No other expressions are successfully evaluated:

$$t \mapsto \lambda \text{ ; else}$$

[6]Note that values $v \in \mathcal{P}(\mathbb{Z})$ do not occur for $eval_{ht,\varnothing}$, but will for the full evaluation semantics in section 6.2.

Chapter 6

Executable HybridUML Semantics: Transformation Definition

This chapter defines the transformation Φ_{HUML} of HybridUML specifications into HL3 models. Since HL3 models have a formal operational semantics (defined in chapter 3), particular HybridUML specifications *spec* obtain their semantics directly by Φ_{HUML}.

The result of Φ_{HUML} consists of two parts: (1) From the HybridUML specification *spec*, the constant part $c_{spec} \in CONST$ of the state space S of the HL3 model is defined. This includes the definition of abstract machines, transitions, and flows from the HybridUML agents and modes with their contained expressions and (HybridUML) transitions. (2) Independently from the particular specification *spec*, HybridUML-specific definitions for operations of abstract subjects (and their internal state) are given, i.e. the operations that define the behavior of *Abstract Machines* and the *Selector*.

By the transformation Φ_{HUML}, a *simulation of the complete HybridUML specification* is defined. We do not consider any architectural specification (see section **??**). Therefore, (1) all HybridUML agents from the specification are mapped to abstract machines. There are *no interface modules*, because the simulation does not interact with an external environment. An architectural specification would be used to define parts of the specification that are defined externally, such that the respective agent instances would be transformed into interface modules, rather than into abstract machines.

Further, (2) the selector defined in this chapter is the *HybridUML Simulation Selector*. If in contrast, specific executions of the HybridUML specification had to be chosen, for example for a test setting (with external components) that utilizes elaborate test data generation algorithms, a different selector would be defined.

This chapter is structured as follows: Sections 6.1, 6.2, and 6.3 define some prerequisites for the definitions of the HL3 model, as well as for the abstract subject's operations. These are (1) the notion of an *intermediate representation* of HybridUML specifications, (2) the definition of the *evaluation semantics* of HybEL expressions in the context of the intermediate HybridUML representation, (3) and the definition of several mappings on expressions and expression nodes for code generation.

In section 6.4, the constant state space c_{spec} of the HL3 model is defined according to HybridUML specifications *spec*, which are given syntactically as discussed in chapter 4. The creation of programs $p \in \mathit{Program}$, which define a significant part of the HL3 model's behavior, is presented separately in section 6.5. Examples of the resulting HL3 model are given by means of references to appendix **??**, which contains the C++ variant of the HL3 program of the Radio-Based Train Control case study, as it is generated by the implementation of the transformation Φ_{HUML}.

Finally, the HybridUML-specific definition of the operations and the internal state of abstract subjects is discussed in section 6.6.

6.1 Intermediate Specification Representation

In this section, the HybridUML specification is transformed into an intermediate representation that consists of:

Tree of Agent Instance Nodes From the agent and agent instance specifications, a tree of agent instance nodes is created. Agent instance nodes are the resulting objects that are derived from recursive application of agent and agent instance specifications.

Set of Basic Agent Instance Nodes The leafs of the tree of agent instance nodes are the active objects that encapsulate the sequential behavioral components of the system.

Sets of Property and Signal Nodes For each agent instance node, a set of property nodes and a set of signal nodes is derived from its agent's properties. These represent instances of properties and signals wrt. the agent instance node.

Sets of Connected Property Nodes and Connected Signal Nodes The maximal sets of connected property nodes and connected signal nodes, as defined by connectors between properties and signals from the HybridUML specification, represent the shared variables of the system.

Tree of Mode Instance Nodes Each basic agent instance node has an attached behavioral specification which is defined by a tree of mode instance nodes. From the top-level mode instance of the associated agent, this tree is derived.

Sets of Control Point Instance Nodes and Sets of Transition Nodes The mode instance nodes are connected by transition nodes via control point instance nodes. The control point instance nodes are instantiated from control point instances and control points of the respective mode instances and modes, the transition nodes represent the transitions from the HybridUML specification which connect them.

Sets of Expression Nodes Attached to mode instance nodes and transition nodes, there are expression nodes that represent the expressions of the original specification: triggers, guards, and actions for transitions, and flow constraints as well as invariant constraints for modes.

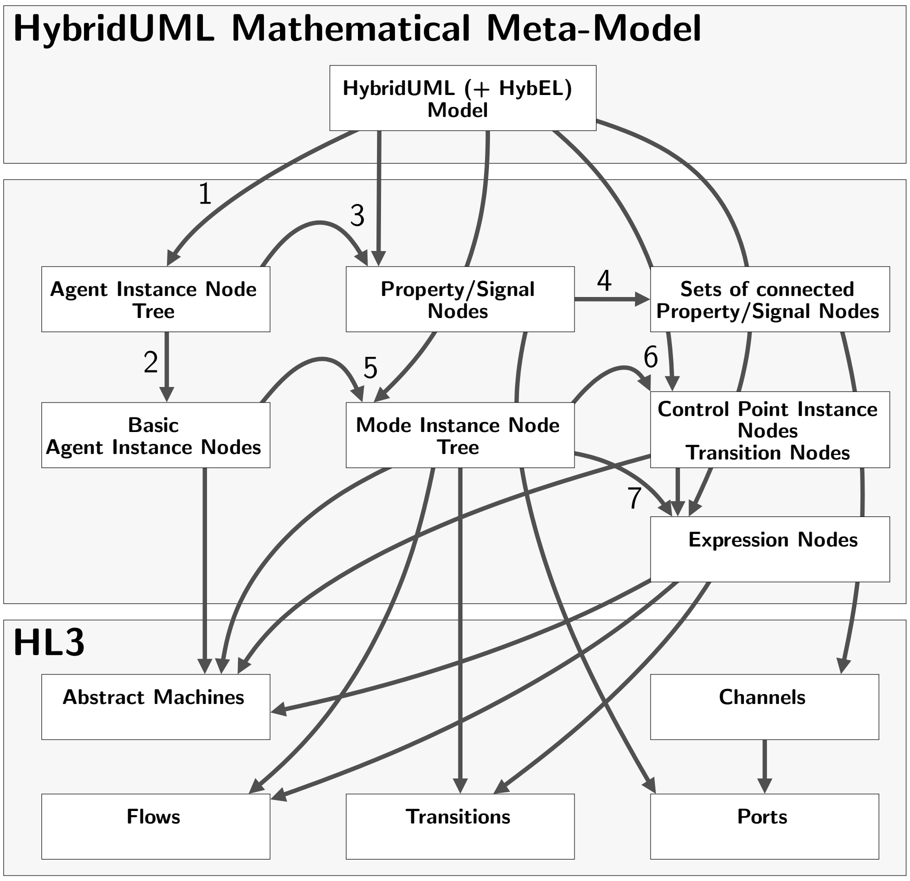

HybridUML Mathematical Meta-Model
HybridUML (+ HybEL) Model
1
2
3
4
5
6
7
Agent Instance Node Tree
Property/Signal Nodes
Sets of connected Property/Signal Nodes
Basic Agent Instance Nodes
Mode Instance Node Tree
Control Point Instance Nodes Transition Nodes
Expression Nodes
HL3
Abstract Machines
Channels
Flows
Transitions
Ports

Figure 6.1: Simplified transformation illustration. From the HybridUML specification, a HL3 model is created, via the Intermediate Representation.

The remainder of this section is divided into structural aspects (tree of agent instance nodes, property and signal nodes, and the connected sets thereof) and behavioral aspects (tree of mode instance nodes with control point instance nodes, transition nodes, and expression nodes).

6.1.1 Structure

Agent Instance Nodes. In contrast to agent instances $ai \in AI$ which define sets of instances per containing agent, an agent instance node represents exactly one single instance of an agent within the system. The possible agent instance nodes are defined as:

$$AIN = \operatorname{seq} AIN \times AI \times \mathbb{N}_0$$

Each of these represents an agent instance, given by its second component

$$ai_{AIN} = \pi_2\, AIN$$

In order to identify the agent instance node *ain* uniquely, the context of the represented agent instance is contained as a path $\langle a_1, \ldots, a_n \rangle \in \operatorname{seq} AIN$ of agent instance nodes leading from its direct ancestor a_1 to a top-level agent instance node a_n. The top-level node itself contains the empty context $\langle\rangle \in \operatorname{seq} AIN$. The context is given as the first component

$$path_{AIN} = \pi_1\, AIN$$

Within the context of its parent agent, the represented agent instance has a multiplicity m. This shall define the number of agent instance nodes that represent the agent instance. Each of these agent instance nodes is supposed to represent one index of $[0, m-1]$:

$$index_{AIN} = \pi_3\, AIN$$

For a particular HybridUML specification, there is a subset $AIN_{spec} \subseteq AIN$ of agent instance nodes that correspond to the specification. These agent instance nodes contain property nodes that shall represent a single index of a property of the corresponding agent, that means, properties of the agent with multiplicity m shall be represented by m independent property nodes:

$$vn_{AIN_{spec}} : AIN_{spec} \to \mathcal{P}(VN_{spec})$$

The same holds for signal nodes:

$$sn_{AIN_{spec}} : AIN_{spec} \to \mathcal{P}(SN_{spec})$$

The definitions are given below, with the definitions of property and signal nodes.

Tree of Agent Instance Nodes. The mapping $tree_{AIN}$ defines how an agent instance node is (recursively) mapped to a tree[1] of agent instances nodes $Tree_{AIN}$, such that the resulting tree represents the complete structural specification that is contained in the corresponding agent instance:

$$\begin{aligned}
&Tree_{AIN} = \operatorname{tree} AIN\\
&tree_{AIN} : AIN \to Tree_{AIN}\\
&(p, ai, n) \mapsto\\
&\quad ((p, ai, n), \{tree_{AIN}((p_s, ai_s, n_s)) \mid ai_s \in part_A(agent_{AI}(ai))\\
&\quad \wedge 0 \le n_s < eval(mult_{AI}(ai), head(p)) \wedge p_s = \langle (p, ai, n) \rangle \frown p\})\\
&\quad ; p \ne \langle\rangle\\
&(\langle\rangle, ai, n) \mapsto\\
&\quad ((p, ai, n), \{tree_{AIN}((p_s, ai_s, n_s)) \mid ai_s \in part_A(agent_{AI}(ai))\\
&\quad \wedge 0 \le n_s < eval_{\varnothing}(mult_{AI}(ai)) \wedge p_s = \langle (p, ai, n) \rangle \frown p\})
\end{aligned}$$

The definition of the evaluation function *eval* of HybEL expressions in the context of agent instance nodes is postponed to section 6.2.

[1]See chapter **??** for the definition of trees.

In order to obtain the tree of agent instance nodes for the complete specification, the mapping $tree_{AIN}$ is used with the single top-level agent instance that represents the system:

$$\begin{aligned} tree_{AI} &: AI \to Tree_{AIN} \\ &ai \mapsto tree_{AIN}((\langle\rangle, ai, 0)) \end{aligned}$$

Thus, $tree_{AI}(ai_{tl})$ is the complete agent instance node tree for the specification. The set of all contained trees is:

$$Tree_{AIN_{spec}} = \{\mathrm{subtrees_{AIN}}(tree_{AI}(ai_{tl}))\}$$

The set of agent instance nodes that are constructed for the specification then is:

$$AIN_{spec} = \{ain \in AIN \mid \exists (ain_1, \{t_0, \ldots, t_k\}) \in Tree_{AIN_{spec}} \bullet ain_1 = ain\}$$

The leafs of the tree of agent instance nodes are the basic agent instance nodes, they define the sequential behavioral components of the resulting system:

$$AIN_{basic} = \{ain \in AIN_{spec} \mid tree_{AIN}(ain) = (ain, \varnothing)\}$$

Property Nodes. A property node represents an index of a property within the context of an agent instance node. The set of all possible property nodes is

$$VN = AIN \times V \times (P_V \cup \{\lambda\}) \times (PI_V \cup \{\lambda\}) \times \mathbb{N}_0$$

The first component is embedding agent instance node:

$$ain_{VN} = \pi_1\ VN$$

The represented variable is contained, too:

$$var_{VN} = \pi_2\ VN$$

If a corresponding variable port instance exists, it shall be associated with the node, otherwise the special value λ denotes the absence of a port instance:

$$portIns_{VN} = \pi_4\ VN$$

If a corresponding variable port exists, it shall be associated with the node, similarly to port instances:

$$port_{VN} = \pi_3\ VN$$

Finally, the node's index wrt. the property's multiplicity is contained:

$$index_{VN} = \pi_5\ VN$$

The subset $VN_{spec} \subseteq VN$ defines the property nodes which are part of the representation of a concrete specification.

Consecutive Index of Property Nodes. In order to realize *point-to-point* connections between port instances of multiplicity m_p, contained by agent instances with multiplicity m_a, every property node is supposed to have a *consecutive index* within the context of the corresponding agent.

For the calculation of the consecutive index, a mapping is provided that gives all property nodes for a single one, such that all property nodes coincide for the port instance they represent:

$$\begin{aligned}
&vn_{VN,PI_V} : VN \rightarrow \mathcal{P}(VN)\\
&(ain, v, p, pi, n) \mapsto \{(ain, v, p, pi, m) \in VN\}\,;\ pi \neq \lambda\\
&(ain, v, p, \lambda, n) \mapsto \varnothing
\end{aligned}$$

Further, a sequence of property nodes, ordered by the properties' indices they represent, is defined:

$$\begin{aligned}
&vnseq : \mathcal{P}(VN) \rightarrow \operatorname{seq} VN\\
&\qquad \text{such that}\\
&\forall\, vn \in \mathcal{P}(VN) \bullet |vnseq(vn)| = |vn| \wedge \operatorname{ran} vnseq(vn) = vn\\
&\qquad \text{and}\\
&\forall\, i_1 \mapsto vn_1, i_2 \mapsto vn_2 \in vnseq(vn) \bullet i_1 \leq i_2 \Rightarrow index_{VN}(vn_1) \leq index_{VN}(vn_2)
\end{aligned}$$

The *local consecutive index* of a property node is then the (unique) position of the property node within the sequence of property nodes that represent its port index:

$$\begin{aligned}
&cindex_{VN,loc} : VN \rightarrow \mathbb{N}_0\\
&vn \mapsto i\,;\ i \mapsto vn \in vnseq(vn_{VN,PI_V}(vn))
\end{aligned}$$

Finally, the consecutive index then is calculated from the agent instance's index and the property node's local consecutive index:

$$\begin{aligned}
&cindex_{VN} : VN \rightarrow \mathbb{N}_0\\
&vn \mapsto index_{AIN}(ain_{VN}(vn)) \cdot |vn_{VN,PI_V}(vn)| + cindex_{VN,loc}(vn)
\end{aligned}$$

Signal Nodes. A signal node represents an index of a signal within the context of an agent instance node, exactly in the same fashion as variable nodes do for properties. Therefore, the set of all possible signal nodes is

$$SN = AIN \times S \times (P_S \cup \{\lambda\}) \times (PI_S \cup \{\lambda\}) \times \mathbb{N}_0$$

The components agent instance node, signal, variable port instance, variable port, and index are accessible by the following projections:

$$\begin{aligned}
&ain_{SN} = \pi_1\, SN\\
&sig_{SN} = \pi_2\, SN\\
&portIns_{SN} = \pi_4\, SN\\
&port_{SN} = \pi_3\, SN\\
&index_{SN} = \pi_5\, SN
\end{aligned}$$

The subset $SN_{spec} \subseteq SN$ then defines the signal nodes that are part of the representation of a concrete specification.

Consecutive Index of Signal Nodes. Point-to-point connections are supported, too, by the definition of the consecutive index for signal nodes. This is done in the same fashion as for property nodes.

A mapping is provided that gives all signal nodes that coincide wrt. the port instance with a given one:

$$\begin{aligned}
&sn_{SN,PI_S} : SN \rightarrow \mathcal{P}(SN)\\
&(ain, s, p, pi, n) \mapsto \{(ain, s, p, pi, m) \in SN\}\,;\ pi \neq \lambda\\
&(ain, s, p, \lambda, n) \mapsto \varnothing
\end{aligned}$$

A sequence of signal nodes is defined:

$$\begin{aligned}
&snseq : \mathcal{P}(SN) \rightarrow \operatorname{seq} SN\\
&\qquad\text{such that}\\
&\forall\, sn \in \mathcal{P}(SN) \bullet |snseq(sn)| = |sn| \wedge \operatorname{ran} snseq(sn) = sn\\
&\qquad\text{and}\\
&\forall\, i_1 \mapsto sn_1, i_2 \mapsto sn_2 \in snseq(sn) \bullet i_1 \leq i_2 \Rightarrow index_{SN}(sn_1) \leq index_{SN}(sn_2)
\end{aligned}$$

The *local consecutive index* of a signal node is:

$$\begin{aligned}
&cindex_{SN,loc} : SN \rightarrow \mathbb{N}_0\\
&sn \mapsto i\,;\ i \mapsto sn \in snseq(sn_{SN,PI_S}(sn))
\end{aligned}$$

The consecutive index then is calculated from the agent instance's index and the signal node's local consecutive index:

$$\begin{aligned}
&cindex_{SN} : SN \rightarrow \mathbb{N}_0\\
&sn \mapsto index_{AIN}(ain_{SN}(sn)) \cdot |sn_{SN,PI_S}(sn)| + cindex_{SN,loc}(sn)
\end{aligned}$$

Mapping to Property Nodes. Based on the tree of agent instance nodes, the mapping vn_{AIN} collects the property nodes that represent the properties of the agents within each context of their agent instance nodes:

$$\begin{aligned}
&vn_{AIN} : Tree_{AIN} \rightarrow \mathcal{P}(VN)\\
&((p, ai, n), \{t_0, \ldots, t_k\}) \mapsto\\
&\qquad \{((p, ai, n), v, pt, pti, m) \mid pti \in portIns_{AI,Var}(ai)\\
&\qquad \wedge m \in eval(indices_{PI_V}(pti), (p, ai, n))\\
&\qquad \wedge pt = port_{PI_V}(pti)\\
&\qquad \wedge v = var_{P_V}(pt) \wedge 0 \leq m < eval(mult_V(v), (p, ai, n))\}\\
&\quad \cup \quad \{((p, ai, n), v, pt, \lambda, m) \mid \nexists\, pti \in portIns_{AI,Var}(ai) \bullet\\
&\qquad\ (m \in eval(indices_{PI_V}(pti), (p, ai, n))\\
&\qquad\ \wedge pt = port_{PI_V}(pti))\\
&\qquad \wedge pt \in port_{A,Var}(agent_{AI}(ai))\\
&\qquad \wedge v = var_{P_V}(pt) \wedge 0 \leq m < eval(mult_V(v), (p, ai, n))\}\\
&\quad \cup \quad \{((p, ai, n), v, \lambda, \lambda, m) \mid \nexists\, pt \in port_{A,Var}(agent_{AI}(ai)) \bullet v = var_{P_V}(pt)\\
&\qquad \wedge v \in var_A(agent_{AI}(ai))\\
&\qquad \wedge 0 \leq m < eval(mult_V(v), (p, ai, n))\}\\
&\quad \cup \quad \bigcup_{i=0}^{k} vn_{AIN}(t_i)
\end{aligned}$$

The property nodes which are part of the representation of the specification are the ones provided by vn_{AIN}, restricted to the agent instance nodes from the specification:

$$VN_{spec} = \{vn \in VN \mid \exists\, t \in Tree_{AIN_{spec}} \bullet vn \in vn_{AIN}(t)\}$$

The mapping $vn_{AIN_{spec}}$ from agent instance nodes to sets of property nodes is directly given by vn_{AIN} as well:

$$\begin{aligned}
&vn_{AIN_{spec}} : AIN_{spec} \rightarrow \mathcal{P}(VN_{spec})\\
&ain \mapsto vn_{AIN}(tree_{AIN}(ain))
\end{aligned}$$

For property nodes that represent parameters, the dedicated value defined by σ_V is evaluated, such that the value can be used while constructing the agent instance node tree. The special "ID" parameters are set to the agent instance node's index:

$$\begin{aligned}
&\sigma_{VN_{spec}} : VN_{spec} \rightarrow VAL_{eval} \cup \{\lambda\}\\
&(ain, v, pt, pti, m) \mapsto eval(exp_{\sigma_V}(val), ain)\\
&\qquad ; val \in \sigma_V \wedge var_{\sigma_V}(val) = v\\
&\qquad \wedge \exists\, a \in A \bullet v \in param_A(a) \wedge v \neq v_{id,agent_{AI}(ai)}\\
&((p, ai, n), v, pt, pti, m) \mapsto n;\ v = v_{id,agent_{AI}(ai)}\\
&(ain, v, pt, pti, m) \mapsto \lambda\,; \text{else}
\end{aligned}$$

Mapping to Signal Nodes. Based on the tree of agent instance nodes, the mapping sn_{AIN} collects the signal nodes that represent the properties of the agents within each context of their agent instance nodes:

$$\begin{aligned}
&sn_{AIN} : Tree_{AIN} \rightarrow \mathcal{P}(SN)\\
&((p, ai, n), \{t_0, \ldots, t_k\}) \mapsto\\
&\qquad \{((p, ai, n), s, pt, pti, m) \mid pti \in portIns_{AI,Sig}(ai)\\
&\qquad \wedge m \in eval(indices_{PI_S}(pti), (p, ai, n))\\
&\qquad \wedge pt = port_{PI_S}(pti)\\
&\qquad \wedge s = sig_{P_S}(pt) \wedge 0 \leq m < eval(mult_S(s), (p, ai, n))\}\\
&\quad \cup \quad \{((p, ai, n), s, pt, \lambda, m) \mid \nexists\, pti \in portIns_{AI,Sig}(ai) \bullet\\
&\qquad (m \in eval(indices_{PI_S}(pti), (p, ai, n))\\
&\qquad \wedge pt = port_{PI_S}(pti))\\
&\qquad \wedge pt \in port_{A,Sig}(agent_{AI}(ai))\\
&\qquad \wedge s = sig_{P_S}(pt) \wedge 0 \leq m < eval(mult_S(s), (p, ai, n))\}\\
&\quad \cup \quad \{((p, ai, n), s, \lambda, \lambda, m) \mid \nexists\, pt \in port_{A,Sig}(agent_{AI}(ai)) \bullet s = sig_{P_S}(pt)\\
&\qquad \wedge s \in sig_A(agent_{AI}(ai))\\
&\qquad \wedge 0 \leq m < eval(mult_S(s), (p, ai, n))\}\\
&\quad \cup \quad \bigcup_{i=0}^{k} sn_{AIN}(t_i)
\end{aligned}$$

The signal nodes which are part of the representation of the specification are the ones provided by sn_{AIN}, restricted to the agent instance nodes from the specification:

$$SN_{spec} = \{sn \in SN \mid \exists\, t \in Tree_{AIN_{spec}} \bullet sn \in sn_{AIN}(t)\}$$

The mapping $sn_{AIN_{spec}}$ from agent instance nodes to sets of signal nodes is directly given by sn_{AIN} as well:

$$\begin{aligned}
&sn_{AIN_{spec}} : AIN_{spec} \to \mathcal{P}(SN_{spec})\\
&ain \mapsto sn_{AIN}(tree_{AIN}(ain))
\end{aligned}$$

Connected Property Nodes. Sets of connected property nodes define shared variables of the system. In the first step, directly connected property nodes are calculated – each set of nodes is connected by a single connector:

$$\begin{aligned}
&VN_{conn,local} =\\
&\qquad \{VN_{sub} \in \mathcal{P}(VN_{spec}) \mid \exists\, c \in C_V \bullet\\
&\qquad \forall (ain_1, v_1, p_1, pi_1, n_1), (ain_2, v_2, p_2, pi_2, n_2) \in VN_{sub} \bullet\\
&\qquad (pi_1 \in portIns_{C_V}(c) \wedge pi_2 \in portIns_{C_V}(c)\\
&\qquad \wedge(kind_{C_V}(c) = ptp \Rightarrow\\
&\qquad\quad cindex_{VN}((ain_1, v_1, p_1, pi_1, n_1)) =\\
&\qquad\qquad cindex_{VN}((ain_2, v_2, p_2, pi_2, n_2))))\}\\
&\quad \cup \quad \{\{vn\} \mid vn \in VN_{spec}\}
\end{aligned}$$

$VN_{conn,local}$ contains sets of property nodes from the specification that are connected locally, i.e. they represent properties of agent instances which are connected through port instances within a common parent agent. Note that this is guaranteed because connectors are restricted to connect port instances only locally.

The sets of property nodes are not maximal, particularly there is a singleton set for each property node from the specification. The different connector kinds are respected, in that only nodes with coinciding consecutive index are connected for point-to-point connectors, in contrast to multicast connectors.

$$\begin{aligned}
&VN_{conn,cont} =\\
&\qquad \{VN_{sub} \in \mathcal{P}(VN_{spec}) \mid \exists\, c \in C_V \bullet\\
&\qquad \forall (ain_1, v_1, p_1, pi_1, n_1), (ain_2, v_2, p_2, pi_2, n_2) \in VN_{sub} \bullet\\
&\qquad (p_1 \in port_{C_V}(c) \wedge pi_2 \in portIns_{C_V}(c)\\
&\qquad \wedge(kind_{C_V}(c) = ptp \Rightarrow\\
&\qquad\quad cindex_{VN}((ain_1, v_1, p_1, pi_1, n_1)) =\\
&\qquad\qquad cindex_{VN}((ain_2, v_2, p_2, pi_2, n_2))))\}
\end{aligned}$$

$VN_{conn,cont}$ contains sets of property nodes that are connected across hierarchy levels. A connector can be attached to a port of its containing agent and thus connect the corresponding property node with property nodes of contained agent instances. Each set contains exactly two property nodes, one per hierarchy level.

The second step consists of joining the sets of property nodes such that every set of such sets is united which share a common property node. This constitutes *transitive connections* via several connectors.

$$
\begin{aligned}
&VN_{conn} = \\
&\quad \{ VN_{sub} \in \mathcal{P}(VN_{spec}) \mid \exists\, n \in \mathbb{N}_0 \bullet \\
&\quad\ (VN_{sub} = \bigcup_{i=0}^{n} VN_i \\
&\quad\ \wedge \forall\, i \in \{0..n\} \bullet VN_i \in (VN_{conn,local} \cup VN_{conn,cont}) \\
&\quad\ \wedge \bigcap_{i=0}^{n} VN_i \neq \varnothing)\}
\end{aligned}
$$

Finally, from the sets of connected property nodes, all maximal sets are taken:

$$VN_{conn,max} = \{ VN_{sub} \in VN_{conn} \mid \nexists\, VN_2 \in VN_{conn} \bullet VN_{sub} \subset VN_2\}$$

Connected Signal Nodes. Sets of connected signal nodes define shared variables of the system, in the same way that connected property nodes do. First, directly connected signal nodes are calculated:

$$
\begin{aligned}
&SN_{conn,local} = \\
&\qquad \{SN_{sub} \in \mathcal{P}(SN_{spec}) \mid \exists\, c \in C_S \bullet \\
&\qquad \forall (ain_1, s_1, p_1, pi_1, n_1), (ain_2, s_2, p_2, pi_2, n_2) \in SN_{sub} \bullet \\
&\qquad (pi_1 \in portIns_{C_S}(c) \wedge pi_2 \in portIns_{C_S}(c) \\
&\qquad \wedge (kind_{C_S}(c) = ptp \Rightarrow \\
&\qquad\quad cindex_{SN}((ain_1, s_1, p_1, pi_1, n_1)) = \\
&\qquad\qquad cindex_{SN}((ain_2, s_2, p_2, pi_2, n_2))))\} \\
&\quad \cup \quad \{\{sn\} \mid sn \in SN_{spec}\}
\end{aligned}
$$

$SN_{conn,local}$ contains sets of signal nodes from the specification that are connected locally.

$$
\begin{aligned}
&SN_{conn,cont} = \\
&\qquad \{SN_{sub} \in \mathcal{P}(SN_{spec}) \mid \exists\, c \in C_S \bullet \\
&\qquad \forall (ain_1, s_1, p_1, pi_1, n_1), (ain_2, s_2, p_2, pi_2, n_2) \in SN_{sub} \bullet \\
&\qquad (p_1 \in port_{C_S}(c) \wedge pi_2 \in portIns_{C_S}(c) \\
&\qquad \wedge (kind_{C_S}(c) = ptp \Rightarrow \\
&\qquad\quad cindex_{SN}((ain_1, s_1, p_1, pi_1, n_1)) = \\
&\qquad\qquad cindex_{SN}((ain_2, s_2, p_2, pi_2, n_2))))\}
\end{aligned}
$$

$VN_{conn,cont}$ contains sets of signal nodes that are connected across hierarchy levels; each set contains exactly two signal nodes, one per hierarchy level.

The second step consists of joining the sets of signal nodes such that every set of such sets is united which share a common signal node.

$$
\begin{aligned}
&SN_{conn} = \\
&\qquad \{SN_{sub} \in \mathcal{P}(SN_{spec}) \mid \exists\, n \in \mathbb{N}_0 \bullet
\end{aligned}
$$

$$\begin{aligned}&(SN_{sub} = \bigcup_{i=0}^{n} SN_i\\
&\wedge \forall\, i \in \{0..n\} \bullet SN_i \in (SN_{conn,local} \cup SN_{conn,cont})\\
&\wedge \bigcap_{i=0}^{n} SN_i \neq \varnothing)\}\end{aligned}$$

Finally, from the sets of connected signal nodes, all maximal sets are taken:

$$SN_{conn,max} = \{SN_{sub} \in SN_{conn} \mid \nexists\, SN_2 \in SN_{conn} \bullet SN_{sub} \subset SN_2\}$$

6.1.2 Behavior

Mode Instance Nodes. Each mode instance from the specification is represented by a mode instance node within a basic agent instance node. In order to identify the node, the path of ancestor mode instance nodes is contained:

$$MIN = \mathrm{seq}\, MIN \times MI \times AIN_{basic}$$

The components are provided by the projections

$$\begin{aligned}&mi_{MIN} = \pi_2\, MIN\\
&path_{MIN} = \pi_1\, MIN\\
&ain_{MIN} = \pi_3\, MIN\end{aligned}$$

Similar to agent instance nodes, the hierarchy of mode instance nodes is represented by a tree structure:

$$\begin{aligned}&Tree_{MIN} = \mathrm{tree}\, MIN\\
&tree_{MIN} : MIN \to Tree_{MIN}\\
&(p, mi, ain) \mapsto\\
&\quad ((p, mi, ain), \{tree_{MIN}((p_s, mi_s, ain)) \mid mi_s \in submode_M(mode_{MI}(mi))\\
&\quad \wedge p_s = \langle(p, mi, ain)\rangle \frown p\})\end{aligned}$$

The behavior of a basic agent instance node is defined by a dedicated mode instance node that represents the mode instance of the corresponding agent:

$$\begin{aligned}&mtree_{AIN} : AIN_{basic} \to Tree_{MIN}\\
&ain \mapsto tree_{MIN}((\langle\rangle, mi, ain))\\
&\quad ;\, mi \in behavior_A(agent_{AI}(ai_{AIN}(ain)))\end{aligned}$$

This mapping is well-defined, since (1) mode instance mi always exists and (2) there is at most one. The corresponding agent for basic agent instance node ain has no contained agent instances, therefore (1) holds and there can be at most one mode instance for any agent, thus (2) follows.

The set of mode instance nodes that represent mode instances for the HybridUML specification and the corresponding trees are those that are attached to the basic agent instance nodes:

$$\begin{aligned}&Tree_{MIN,spec} =\\
&\quad \{t \in Tree_{MIN} \mid \exists\, ain \in AIN_{basic} \bullet t \in \mathrm{subtrees}_{\mathrm{MIN}}(mtree_{AIN}(ain))\}\\
&MIN_{spec} =\\
&\quad \{min \in MIN \mid \exists(min_1, \{t_0, \ldots, t_k\}) \in Tree_{MIN,spec} \bullet min_1 = min\}\end{aligned}$$

Control Point Instance Nodes. There are control point instance nodes that represent the control point instances of the mode instances:

$$CPIN = MIN \times CPI$$

with projections

$$min_{CPIN} = \pi_1\, CPIN$$
$$cpi_{CPIN} = \pi_2\, CPIN$$

The control point instance nodes are attached to the mode instance nodes:

$$\begin{aligned} & cpin_{MIN} : Tree_{MIN} \to \mathcal{P}(CPIN) \\ & \quad ((p, mi, am), \{t_0, \ldots, t_k\}) \mapsto \\ & \quad \{((p, mi, am), cpi) \mid cpi \in cpi_{MI}(mi)\} \cup \bigcup_{i=0}^{k} cpin_{MIN}(t_i) \end{aligned}$$

The control point instance nodes from the specification are those that are attached to mode instance nodes from the specification:

$$CPIN_{spec} = \{cpin \in CPIN \mid \exists\, t \in Tree_{MIN,spec} \bullet cpin \in cpin_{MIN}(t)\}$$

Transition Nodes. Transition nodes are the representations of transitions in the context of mode instance nodes:

$$TN = MIN \times T$$

The components are given by

$$min_{TN} = \pi_1\, TN$$
$$trans_{TN} = \pi_2\, TN$$

The transition nodes are attached to the mode instance nodes:

$$\begin{aligned} & tn_{MIN} : Tree_{MIN} \to \mathcal{P}(TN) \\ & \quad ((p, mi, am), \{t_0, \ldots, t_k\}) \mapsto \\ & \quad \{((pi, mi, am), trans) \mid trans \in trans_M(mode_{MI}(mi))\} \\ & \quad \cup \bigcup_{i=0}^{k} tn_{MIN}(t_i) \end{aligned}$$

The transition nodes from the specification are those that are contained in the mode instance nodes from the specification:

$$TN_{spec} = \{tn \in TN \mid \exists\, t \in Tree_{MIN,spec} \bullet tn \in tn_{MIN}(t)\}$$

Each transition node is connected with the control point instance node that represents its source:

$$\begin{aligned} & src_{TN_{spec}} : TN_{spec} \to CPIN_{spec} \\ & \quad (min, trans) \mapsto (min, cpi);\ src_T(trans) = cp_{CPI}(cpi) \\ & \quad (min, trans) \mapsto (min_s, cpi) \\ & \qquad ;\ src_T(trans) = cpi\ \wedge \\ & \qquad \exists\, t, t_0, \ldots, t_k \in Tree_{MIN,spec}, st \in \mathcal{P}(Tree_{MIN,spec}), i \in 0..k \bullet \\ & \qquad t = (min, \{t_0, \ldots, t_k\}) \wedge t_i = (min_s, st) \end{aligned}$$

As in the definition of transitions themselves, it is distinguished between transitions that originate from a control point or from a control point instance. In the first case, the control point belongs to the transition's parent mode itself (represented by *min*), whereas in the latter case the control point instance is attached to one of the submodes (represented by min_s).

The transition nodes are connected with the control point instance nodes that represent their target:

$$\begin{aligned}
&tar_{TN_{spec}} : TN_{spec} \rightarrow CPIN_{spec}\\
&\quad (min, trans) \mapsto (min, cpi);\ tar_T(trans) = cp_{CPI}(cpi)\\
&\quad (min, trans) \mapsto (min_s, cpi)\\
&\quad\ ;\ tar_T(trans) = cpi \wedge\\
&\quad\ \exists\, t, t_0, \ldots, t_k \in Tree_{MIN,spec}, st \in \mathcal{P}(Tree_{MIN,spec}), i \in 0..k \bullet\\
&\quad\ t = (min, \{t_0, \ldots, t_k\}) \wedge t_i = (min_s, st)
\end{aligned}$$

Expression Nodes. Expressions are attached to transitions, to modes, or to agents. They represent one of (1) trigger, (2) guard, (3) action, (4) flow, (5) invariant constraint, and (6) init state constraint:

$$\begin{aligned}
&ExpN = TrgExpN \cup GrdExpN \cup ActExpN \cup FlowExpN \cup InvExpN\\
&\quad \cup IscExpN\\
&TrgExpN = TN \times Exp \times \{trg\}\\
&GrdExpN = TN \times Exp \times \{grd\}\\
&ActExpN = TN \times Exp \times \{act\}\\
&FlowExpN = MIN \times Exp \times \{flow\}\\
&InvExpN = MIN \times Exp \times \{inv\}\\
&IscExpN = AIN \times Exp \times \{isc\}
\end{aligned}$$

They are partitioned into transition expression nodes, mode expression nodes, and agent expression nodes:

$$\begin{aligned}
&TExpN = TrgExpN \cup GrdExpN \times ActExpN\\
&MExpN = FlowExpN \cup InvExpN\\
&AExpN = IscExpN
\end{aligned}$$

Transition expression nodes consist of a transition node, an expression, and an expression kind (or role):

$$\begin{aligned}
&tn_{TExpN} = \pi_1\ TExpN\\
&exp_{TExpN} = \pi_2\ TExpN\\
&kind_{TExpN} = \pi_3\ TExpN
\end{aligned}$$

Mode expression nodes consist of a mode instance node, an expression, and an expression kind (or role):

$$\begin{aligned}
&min_{MExpN} = \pi_1\ MExpN\\
&exp_{MExpN} = \pi_2\ MExpN\\
&kind_{MExpN} = \pi_3\ MExpN
\end{aligned}$$

Agent expression nodes consist of an agent instance node, an expression, and an expression kind (or role):

$$\begin{aligned}
ain_{AExpN} &= \pi_1\, AExpN \\
exp_{AExpN} &= \pi_2\, AExpN \\
kind_{AExpN} &= \pi_3\, AExpN
\end{aligned}$$

Every expression node is – directly or indirectly – contained in an agent instance node. For convenience, a respective mapping is defined:

$$\begin{aligned}
& ain_{ExpN} : ExpN \rightarrow AIN \\
& \qquad expn \mapsto \begin{cases} ain_{MIN}(min_{MExpN}(expn));\ expn \in MExpN \\ ain_{MIN}(min_{TN}(tn_{TExpN}(expn)));\ expn \in TExpN \\ ain_{AExpN}(expn);\ expn \in AExpN \end{cases}
\end{aligned}$$

Further, the expression that is represented by an arbitrary expression node is given by

$$\begin{aligned}
& exp_{ExpN} : ExpN \mapsto Exp \\
& \qquad expn \mapsto \begin{cases} exp_{MExpN}(expn);\ expn \in MExpN \\ exp_{TExpN}(expn);\ expn \in TExpN \\ exp_{AExpN}(expn);\ expn \in AExpN \end{cases}
\end{aligned}$$

There are three kinds of expressions which are attached to transitions: trigger expressions, guard expressions, and action expressions. There is up to one trigger expression per transition, which is represented by an expression node and attached to the corresponding transition node:

$$\begin{aligned}
& trg_{TN_{spec}} : TN_{spec} \rightarrow \mathcal{P}(TrgExpN) \\
& \quad (min, trans) \mapsto \{((min, trans), exp, trg) \mid exp \in sig_T(trans)\}
\end{aligned}$$

For each transition, up to one boolean guard expression exists. If there is none, this is interpreted as the expression exp_{true} that always evaluates to `true`. A corresponding expression node is created:

$$\begin{aligned}
& grd_{TN_{spec}} : TN_{spec} \rightarrow \mathcal{P}(GrdExpN) \\
& \quad (min, trans) \mapsto \{((min, trans), exp, grd) \mid exp \in grd_T(trans)\} \\
& \quad\ ;\ grd_T(trans) \neq \varnothing \\
& \quad (min, trans) \mapsto \{((min, trans), exp_{true}, grd)\} \\
& \quad\ ;\ grd_T(trans) = \varnothing
\end{aligned}$$

The sequence of actions of a transition defines a sequence of action nodes within the context of the transition node:

$$\begin{aligned}
& act_{TN_{spec}} : TN_{spec} \rightarrow \operatorname{seq} ActExpN \\
& \quad (min, trans) \mapsto san \\
& \quad\ ;\ san = \langle an_1, \ldots, an_k \rangle \wedge k = |act_T(trans)| \wedge \\
& \quad\ \ \forall\, i \in \{1..k\} \bullet an_i = ((min, trans), act_T(trans)(i), act)
\end{aligned}$$

Expressions which are attached to modes are distinguished as flow expressions and invariant expressions. Note that there is no technical difference between the specification's flow and algebraic constraints; they are subsumed as flow expressions here:

$$\begin{aligned} & \mathit{flow}_{\mathit{MIN}_{spec}} : \mathit{MIN}_{spec} \to \mathcal{P}(\mathit{FlowExpN}) \\ & \quad (sm, mi, am) \mapsto \{((sm, mi, am), exp, \mathit{flow}) \mid \\ & \qquad exp \in \mathit{flow}_M(\mathit{mode}_{MI}(mi)) \vee exp \in \mathit{alge}_M(\mathit{mode}_{MI}(mi))\} \end{aligned}$$

Invariant expressions are represented by corresponding nodes:

$$\begin{aligned} & \mathit{inv}_{\mathit{MIN}_{spec}} : \mathit{MIN}_{spec} \to \mathcal{P}(\mathit{InvExpN}) \\ & \quad (sm, mi, am) \mapsto \{((sm, mi, am), exp, inv) \mid exp \in \mathit{inv}_M(\mathit{mode}_{MI}(mi))\} \end{aligned}$$

Init state constraint expression nodes are constructed from the available agent instance nodes and the init state constraint expressions of the corresponding agent instances and agents:

$$\begin{aligned} & \mathit{isc}_{\mathit{AIN}_{spec}} : \mathit{AIN}_{spec} \to \mathcal{P}(\mathit{IscExpN}) \\ & \quad (sa, ai, n) \mapsto \{((sa, ai, n), exp, isc) \mid \\ & \qquad exp \in \mathit{initState}_{AI}(ai) \cup \mathit{initState}_A(\mathit{agent}_{AI}(ai))\} \end{aligned}$$

All init state constraint expression nodes are collected recursively for agent instance nodes, such that each agent instance node is mapped to its own init states and the init states of its ancestor nodes:

$$\begin{aligned} & \mathit{allisc}_{\mathit{AIN}_{spec}} : \mathit{AIN}_{spec} \to \mathcal{P}(\mathit{IscExpN}) \\ & \quad (\langle p \rangle \frown sq, ai, n) \mapsto \mathit{isc}_{\mathit{AIN}_{spec}}(\langle p \rangle \frown sq, ai, n) \cup \mathit{allisc}_{\mathit{AIN}_{spec}}(p) \\ & \quad (\langle\rangle, ai, n) \mapsto \mathit{isc}_{\mathit{AIN}_{spec}}(\langle\rangle, ai, n) \end{aligned}$$

The sets of available expression nodes for a given HybridUML specification are:

$$\begin{aligned} & \mathit{ExpN}_{spec} = \mathit{TrgExpN}_{spec} \cup \mathit{GrdExpN}_{spec} \cup \mathit{ActExpN}_{spec} \cup \mathit{FlowExpN}_{spec} \\ & \quad \cup \mathit{InvExpN}_{spec} \\ & \mathit{TrgExpN}_{spec} = \{txn \in \mathit{TrgExpN} \mid \exists\, tn \in \mathit{TN}_{spec} \bullet txn \in \mathit{trg}_{\mathit{TN}_{spec}}(tn)\} \\ & \mathit{GrdExpN}_{spec} = \{gxn \in \mathit{GrdExpN} \mid \exists\, tn \in \mathit{TN}_{spec} \bullet gxn \in \mathit{grd}_{\mathit{TN}_{spec}}(tn)\} \\ & \mathit{ActExpN}_{spec} = \{axn \in \mathit{ActExpN} \mid \exists\, tn \in \mathit{TN}_{spec} \bullet axn \in \mathrm{ran}(\mathit{act}_{\mathit{TN}_{spec}}(tn))\} \\ & \mathit{FlowExpN}_{spec} = \\ & \quad \{fxn \in \mathit{FlowExpN} \mid \exists\, min \in \mathit{MIN}_{spec} \bullet fxn \in \mathit{flow}_{\mathit{MIN}_{spec}}(min)\} \\ & \mathit{InvExpN}_{spec} = \{ixn \in \mathit{InvExpN} \mid \exists\, min \in \mathit{MIN}_{spec} \bullet ixn \in \mathit{inv}_{\mathit{MIN}_{spec}}(min)\} \\ & \mathit{IscExpN}_{spec} = \{ixn \in \mathit{IscExpN} \mid \exists\, ain \in \mathit{AIN}_{spec} \bullet ixn \in \mathit{isc}_{\mathit{AIN}_{spec}}(ain)\} \\ & \mathit{TExpN}_{spec} = \mathit{TrgExpN}_{spec} \cup \mathit{GrdExpN}_{spec} \cup \mathit{ActExpN}_{spec} \\ & \mathit{MExpN}_{spec} = \mathit{FlowExpN}_{spec} \cup \mathit{InvExpN}_{spec} \\ & \mathit{AExpN}_{spec} = \mathit{IscExpN}_{spec} \end{aligned}$$

6.2 Evaluation Semantics of HybEL Expressions

In this section, the semantics for the evaluation of HybEL expressions is appended. For its definition, the notion of agent instance nodes (from section 6.1) is needed. It is needed to define values for (1) multiplicity expressions of agent instances, properties, and signals, as well as for (2) index specifications of properties and signals.

The semantics given here extends the skeleton semantics of section 5.4, in that it defines values for properties that are equipped with a value specification. Further, the evaluation of sets of integers is added, as it is needed for the index specifications of properties and signals.

Expression Context of Agents. For the evaluation of expressions, the variable and signal context is required. This is defined by the agent which contains the expression, such that exactly the agent's variables and signals are available:

$$ctx_A : A \to CTX$$
$$a \mapsto (var_A(a), \{(s, ac) \mid s \in sig_A(a) \wedge \exists\, p \in port_{A,Sig}(a) \bullet acc_{P_S}(p) = ac\})$$

Expression Context of Agent Instance Nodes. For convenience, the context given by agent instance nodes is defined in a straight-forward way, such that it is given by the agent instance node's agent:

$$ctx_{AIN} : AIN \to CTX$$
$$ain \mapsto ctx_A(agent_{AI}(ai_{AIN}(ain)))$$

Semantics of Expressions. The *evaluation semantics* of hybel expressions is then given by

$$eval : Exp \times AIN \to VAL_{eval} \cup \{\lambda\}$$
$$eval(exp, ain) = eval_{ht}(ht(ctx_{AIN}(ain))(exp), ain)$$

Each hybel item tree along with an agent instance node is mapped to either a value, or to the special value λ that denotes the absence of a value:

$$eval_{ht} : \mathrm{tree_o}\, HybelItem \times AIN \to VAL_{eval} \cup \{\lambda\}$$

Literal values *are* the values (as for $eval_{ht,\varnothing}$):

$$((((t, lit), l), sub), ain) \mapsto l;\ t \in DT$$
$$((((sdt_{anon}, lit), v), \langle t_1, \ldots, t_n\rangle), ain)$$
$$\mapsto \langle eval_{ht}(t_1, ain), \ldots, eval_{ht}(t_n, ain)\rangle$$

Operations are evaluated (as for $eval_{ht,\varnothing}$):

$$((((t, op), \Diamond), \langle t_1, t_2\rangle), ain) \mapsto eval_{ht}(t_1, ain) \Diamond eval_{ht}(t_2, ain)$$
$$;\ t_1, t_2 \in \{int, real, anaReal\} \wedge \Diamond \in \{+, -, \cdot, /, \hat{}, <, \leq, \geq, >\}$$
$$\vee t_1 = t_2 \wedge \Diamond \in \{==, \neq\}$$
$$\vee t_1, t_2 \in \{bool\} \wedge \Diamond \in \{\wedge, \vee\}$$
$$((((t, op), \neg), \langle t_1\rangle), ain) \mapsto \neg eval_{ht}(t_1, ain)$$

Integer set specifications define sets of integers:

$$((((intSet, intSpecs), val), \langle$$
$$(((intSet, intRange), val_1), \langle s_{low,1}, s_{up,1}\rangle), \ldots,$$
$$(((intSet, intRange), val_n), \langle s_{low,n}, s_{up,n}\rangle)\rangle), ain)$$
$$\mapsto \{eval_{ht}(s_{low,1}, ain), \ldots, eval_{ht}(s_{up,1}, ain)\} \cup \ldots \cup$$
$$\{eval_{ht}(s_{low,n}, ain), \ldots, eval_{ht}(s_{up,n}, ain)\}$$

Variables are mapped to assigned values, if the agent instance node has a corresponding value specification. If an index sub-expression is available, it is evaluated itself, otherwise index 0 is applied:

$$\begin{aligned}
&((((t, \mathit{var}), ((t, (v, \mathit{acc})), \langle((\mathit{indexExp}, \mathit{htree}), \mathit{sub}_{id})\rangle)), \mathit{sub}), \mathit{ain}) \\
&\qquad \mapsto \sigma_{VN_{spec}}(\mathit{ain}, v, p, \mathit{pi}, \mathit{eval}_{ht}(\mathit{htree}, \mathit{ain})) \\
&\qquad \text{with unique } (\mathit{ain}, v, p, \mathit{pi}, \mathit{eval}_{ht}(\mathit{htree}, \mathit{ain})) \in \mathit{vn}_{AIN}(\mathit{tree}_{AIN}(\mathit{ain})) \\
&((((t, \mathit{var}), ((t, (v, \mathit{acc})), \langle\rangle)), \mathit{sub}), \mathit{ain}) \mapsto \sigma_{VN_{spec}}(\mathit{ain}, v, p, \mathit{pi}, 0) \\
&\qquad \text{with unique } (\mathit{ain}, v, p, \mathit{pi}, 0) \in \mathit{vn}_{AIN}(\mathit{tree}_{AIN}(\mathit{ain}))
\end{aligned}$$

The remaining expressions have no evaluation result:

$$(t, \mathit{ain}) \mapsto \lambda \,;\ \text{else}$$

6.3 Prerequisites for Code Creation

This section provides some prerequisites that are used for code creation (discussed in section 6.5), for the definition of local HL3 variables, which is done in section 6.4, as well as for the HybridUML-specific definition of operations of abstract subjects in section 6.6.

First, the determination of different sets of variables and signals from expressions is defined, followed by the determination of variable and signal nodes from expression nodes. Finally, the hybel item tree representation of expressions from expression nodes is given.

6.3.1 Variables and Signals of Expressions

As a prerequisite for the mapping of expression nodes to programs, the sets of HybridUML variables and signals, which are accessed by particular expressions, are determined.

Read Variables of Hybel Trees. The variables which are read by an expression are given by a mapping of hybel item trees to sets of HybridUML variables:

$$\mathit{var}_{ht,read} : \mathrm{tree}_{\mathrm{o}}\ \mathit{HybelItem} \rightarrow \mathcal{P}(V)$$

For a hybel item tree that represents a variable, the variable itself is contained. Additionally, variables of subexpressions (for structured data types) as well as variables contained in the identifier item tree (as part of the index expression) are collected. The corresponding definition of $\mathit{var}_{it,read}$ is given below.

$$\begin{aligned}
&(((t, r), ((t_{id}, (v, \mathit{acc})), \mathit{sub}_{id})), \langle s_1, \dots s_n\rangle) \mapsto \\
&\qquad \{v\} \cup \bigcup_{i=1}^{n} \mathit{var}_{ht,read}(s_i) \cup \mathit{var}_{it,read}(((t_{id}, (v, \mathit{acc})), \mathit{sub}_{id})) \\
&\text{with } r \in \{\mathit{var}, \mathit{derivVar}\}
\end{aligned}$$

Expressions that send signals may contain read variables in subexpressions, from which values of signal parameters are created. Further, a signal's index expression can contain read variables, too:

$$(((\mathit{sigtype}, \mathit{sendSig}), \mathit{idtree}), \langle s_1, \ldots s_n\rangle) \mapsto \\ \bigcup_{i=1}^{n} \mathit{var}_{ht,read}(s_i) \cup \mathit{var}_{it,read}(\mathit{idtree})$$

For trigger expressions, i.e. expressions that read signals, only variables of the index expression are collected (via the identifier item tree). Variables of subexpressions are *written* and thus left out.

$$(((\mathit{sigtype}, \mathit{recvSig}), \mathit{idtreee}), \mathit{sub}) \mapsto \mathit{var}_{it,read}(\mathit{idtree})$$

Most assignment expressions contain a left-hand side that consists of a variable that is written only, and a right-hand side that is read only. Therefore, read variables are collected from the hybel item tree's subexpressions, excluding the first one (which represents the left-hand side). Additionally, the identifier item tree of the first subexpression is searched for contained index expressions, which again may contain read variables:

$$(((t, r), \mathit{val}), \langle(((t_1, \mathit{var}), \mathit{idtree}_1), \mathit{sub}_1), s_2, \ldots, s_n\rangle) \mapsto \\ \bigcup_{i=2}^{n} \mathit{var}_{ht,read}(s_i) \cup \mathit{var}_{it,read}(\mathit{idtree}_1) \\ \text{with } r \in \{\mathit{ass}, \mathit{intNondetAss}, \mathit{diffAss}\}$$

Note that assignments *to* derivatives are excluded above, since the derivative's variable is not only written, but also read. Therefore, for those assignments, the first subexpression is included in the generic rule given below.

Index assignment expressions are treated separately, too, because they explicitly do not contain any read variables:

$$(((\mathit{int}, \mathit{indexAss}), \mathit{val}), \mathit{sub}) \mapsto \varnothing$$

All remaining hybel item trees are mapped according to the following generic rule, that does not collect variables from the expression itself, but from all contained subexpressions:

$$(\mathit{hi}, \langle s_1, \ldots, s_n\rangle) \mapsto \bigcup_{i=1}^{n} \mathit{var}_{ht,read}(s_i)\,; \ \text{ else}$$

Read Variables of Identifier Trees. The variables which are read by an identifier expression are given by the mapping of identifier item trees to sets of HybridUML variables:

$$\mathit{var}_{it,read} : \mathrm{tree}_{\mathrm{o}}\, \mathit{IdItem} \to \mathcal{P}(V)$$

Since the identifier itself is handled by the mapping $\mathit{var}_{ht,read}$, this mapping only collects contained variables of index expressions:

$$((\mathit{indexExp}, \mathit{htree}), \mathit{sub}) \mapsto \mathit{var}_{ht,read}(\mathit{htree})$$

For subexpressions, representing sub-variables of a structured data type, also only index expressions are searched; therefore the sub-variables themselves are omitted. This is desired, because variables of structured data type will always be read or written entirely within the generated code.

$$((t, \mathit{val}), \langle s_1, \ldots, s_n \rangle) \mapsto \bigcup_{i=1}^{n} \mathit{var}_{\mathit{it},\mathit{read}}(s_i)\,; \text{ else}$$

Derivative Read Access of Hybel Trees. The variables for which the derivative value is read are collected for each hybel item tree, by the function

$$\mathit{var}_{\mathit{ht},\mathit{readDeriv}} : \mathrm{tree_o}\ \mathit{HybelItem} \to \mathcal{P}(V)$$

Trees that represent derivatives map to their variables:

$$(((\mathit{anaReal}, \mathit{derivVar}), ((\mathit{anaReal}, (v, \mathit{acc})), \mathit{sub}_{\mathit{id}})), \langle s_1, \ldots s_n \rangle) \mapsto \{v\}$$

From the above variables, the ones that are only written shall be excluded; therefore from assignment expressions, the left-hand side is omitted:

$$(((\mathit{anaReal}, \mathit{diffAss}), \mathit{val}), \langle s_1, \ldots, s_n \rangle) \mapsto \bigcup_{i=2}^{n} \mathit{var}_{\mathit{ht},\mathit{readDeriv}}(s_i)$$

All remaining trees are only descended, such that their sub-expressions are searched for respective variables:

$$(\mathit{item}, \langle s_1, \ldots, s_n \rangle) \mapsto \bigcup_{i=1}^{n} \mathit{var}_{\mathit{ht},\mathit{readDeriv}}(s_i)\,; \text{ else}$$

Previous Value Access of Hybel Trees. In contrast to “conventional” read access to variables, some expressions require the read access to the previous value of a variable. Those variables are given by the mapping

$$\mathit{var}_{\mathit{ht},\mathit{readPrev}} : \mathrm{tree_o}\ \mathit{HybelItem} \to \mathcal{P}(V)$$

Variables for which the derivative value is read within the expression are included, in the same manner as for $\mathit{var}_{\mathit{ht},\mathit{readDeriv}}$:

$$(((\mathit{anaReal}, \mathit{derivVar}), ((\mathit{anaReal}, (v, \mathit{acc})), \mathit{sub}_{\mathit{id}})), \langle s_1, \ldots s_n \rangle) \mapsto \{v\}$$

The ones that are only written shall be excluded, too:

$$(((\mathit{anaReal}, \mathit{diffAss}), \mathit{val}), \langle s_1, \ldots, s_n \rangle) \mapsto \bigcup_{i=2}^{n} \mathit{var}_{\mathit{ht},\mathit{readPrev}}(s_i)$$

Additionally, real-valued variables which take part in an equality comparison are included. This is necessary, because of the discretization of continuous steps, an approximation has to be done for those comparisons. See section 6.5 for details.

$$\begin{aligned}&(((\mathit{bool}, \mathit{op}), \mathit{val}), \langle ((\mathit{role}_1, \mathit{val}_1), \mathit{sub}_1), ((\mathit{role}_2, \mathit{val}_2), \mathit{sub}_2) \rangle) \mapsto \\ &\qquad \mathit{var}_{\mathit{ht},\mathit{read}}(((\mathit{role}_1, \mathit{val}_1), \mathit{sub}_1)) \cup \mathit{var}_{\mathit{ht},\mathit{read}}(((\mathit{role}_2, \mathit{val}_2), \mathit{sub}_2)) \\ &\text{with } \{\mathit{role}_1, \mathit{role}_2\} \cap \{(\mathit{real}, \mathit{op}), (\mathit{anaReal}, \mathit{diffOp}), (\mathit{real}, \mathit{lit}), \\ &\qquad (\mathit{real}, \mathit{var}), (\mathit{anaReal}, \mathit{var}), (\mathit{anaReal}, \mathit{derivVar})\} \\ &\qquad \neq \varnothing\end{aligned}$$

All remaining trees are only descended, such that their sub-expressions are searched for respective variables:

$$(\mathit{item}, \langle s_1, \ldots, s_n \rangle) \mapsto \bigcup_{i=1}^{n} \mathit{var}_{\mathit{ht},\mathit{readPrev}}(s_i)\,; \text{ else}$$

Read Signals of Hybel Trees. Read signals, i.e. triggers are only given directly by a trigger expression. Since there are no nested triggers, the complete mapping is

$$\begin{aligned}
&\mathit{sig}_{ht,read} : \mathrm{tree_o}\ \mathit{HybelItem} \to \mathcal{P}(S)\\
&\qquad(((\mathit{sigtype}, \mathit{recvSig}), ((t_{id}, (s, \mathit{acc})), \mathit{sub}_{id})), \mathit{sub}) \mapsto \{s\}\\
&\qquad \mathit{hi} \mapsto \varnothing\,;\ \text{ else}
\end{aligned}$$

Written Variables of Hybel Trees. The variables which are written by an expression are given by a corresponding mapping of hybel item trees to sets of HybridUML variables:

$$\mathit{var}_{ht,write} : \mathrm{tree_o}\ \mathit{HybelItem} \to \mathcal{P}(V)$$

For any kind of assignment statement, the first subexpression represents a variable that is written:

$$\begin{aligned}
&(((t, r), \mathit{val}), \langle(((t_1, r_1), ((t_{1,id}, (v, \mathit{acc})), \mathit{sub}_{1,id})), \mathit{sub}_1), s_2, \ldots, s_n\rangle) \mapsto\\
&\qquad \{v\} \cup \bigcup_{i=2}^{n} \mathit{var}_{ht,write}(s_i)\\
&\text{with } r \in \{\mathit{ass}, \mathit{diffAss}, \mathit{intNondetAss}, \mathit{indexAss}\}\\
&\qquad \wedge\ r_1 \in \{\mathit{var}, \mathit{derivVar}\}
\end{aligned}$$

Trigger expressions can contain parameter subexpressions, that consist of variables that are assigned with the actual parameter values. Additionally, the signal may have an attached index assignment expression, included in the identifier item tree:

$$\begin{aligned}
&(((\mathit{sigtype}, \mathit{recvSig}), \mathit{idtree}), \langle\\
&\qquad(((t_1, r_1), ((t_{1,id}, (v_1, \mathit{acc})), \mathit{sub}_{1,id})), \mathit{sub}_1), \ldots,\\
&\qquad(((t_n, r_n), ((t_{n,id}, (v_n, \mathit{acc})), \mathit{sub}_{n,id})), \mathit{sub}_n)\rangle)\\
&\mapsto \mathit{var}_{it,write}(\mathit{idtree}) \cup \{v_1, \ldots, v_n\}
\end{aligned}$$

For all other trees, the variables of their sub-expressions are collected:

$$(\mathit{item}, \langle s_1, \ldots, s_n\rangle) \mapsto \bigcup_{i=1}^{n} \mathit{var}_{ht,write}(s_i)\,;\ \text{ else}$$

Written Variables of Identifier Trees. The variables which are written by an identifier expression are given by

$$\mathit{var}_{it,write} : \mathrm{tree_o}\ \mathit{IdItem} \to \mathcal{P}(V)$$

Since the identifier itself is handled by the mapping $\mathit{var}_{ht,write}$, this mapping only collects contained variables of index expressions:

$$((\mathit{indexExp}, \mathit{htree}), \mathit{sub}) \mapsto \mathit{var}_{ht,write}(\mathit{htree})$$

Like for $\mathit{var}_{it,read}$, sub-variables of a structured data type are omitted. Therefore the recursive descending into the identifier tree only affects index expressions:

$$((t, \mathit{val}), \langle s_1, \ldots, s_n\rangle) \mapsto \bigcup_{i=1}^{n} \mathit{var}_{it,write}(s_i)\,;\ \text{ else}$$

Derivative Write Access of Hybel Trees. The variables for which the derivative value is written are collected for each hybel item tree:

$$var_{ht,writeDeriv} : \text{tree}_\text{o}\ HybelItem \to \mathcal{P}(V)$$

The variable of the left-hand side of an assignment expression is contained, if it is a derivative:

$$\begin{aligned}&(((anaReal, diffAss), val),\\ &\quad\langle((((anaReal, derivVar), ((anaReal, (v, acc)), sub_{id})), sub), s_2, \ldots, s_n\rangle)\\ &\mapsto \{v\}\end{aligned}$$

For other trees, the sub-expressions are searched for respective variables:

$$(item, \langle s_1, \ldots, s_n\rangle) \mapsto \bigcup_{i=1}^{n} var_{ht,writeDeriv}(s_i)\ ;\ \text{else}$$

Written Signals of Hybel Trees. Written signals are given by the mapping

$$sig_{ht,write} : \text{tree}_\text{o}\ HybelItem \to \mathcal{P}(S)$$

A signal raise statement contains a written signal:

$$(((sigtype, sendSig), ((t_{id}, (s, acc)), sub_{id})), sub) \mapsto \{s\}$$

Some other kinds of expressions may host a signal raise statement, therefore all remaining expressions are descended:

$$(item, \langle s_1, \ldots, s_n\rangle) \mapsto \bigcup_{i=1}^{n} sig_{ht,write}(s_i)\ ;\ \text{else}$$

6.3.2 Variable and Signal Nodes of Expression Nodes

For HybridUML variables and signals, there are mappings that provide the variable or signal nodes which represent the given variable or signal within the context of an expression node. Each variable or signal node represents a dedicated index of the respective variable or signal, therefore the number of nodes and the multiplicity of the variable or signal are equal.

$$\begin{aligned}&vn_{expn} : V \times ExpN \to \mathcal{P}(VN)\\ &\quad(v, expn) \mapsto \{vn \in vn_{AIN}(tree_{AIN}(ain_{ExpN}(expn))) \mid var_{VN}(vn) = v\}\\ &sn_{expn} : S \times ExpN \to \mathcal{P}(SN)\\ &\quad(s, expn) \mapsto \{sn \in sn_{AIN}(tree_{AIN}(ain_{ExpN}(expn))) \mid sig_{SN}(sn) = s\}\end{aligned}$$

6.3.3 Hybel Item Trees of Expression Nodes

For convenience, a mapping from expression nodes to the hybel item tree that represents the expression of the node is defined. For this, the mapping from agent instance nodes to expression contexts of section 6.2 is used:

$$\begin{aligned}&ht_{expn} : ExpN_{spec} \to \text{tree}_\text{o}\ HybelItem\\ &expn \mapsto ht(ctx_{AIN}(ain_{ExpN}(expn)))(exp_{ExpN}(expn))\end{aligned}$$

6.4 HL3 Model Definition

Based on the intermediate representation of HybridUML specifications given in section 6.1, we instantiate the HL3 model that represents a particular HybridUML specification in this section. Therefore, we define the corresponding constant part $c_{spec} \in CONST$ of the state space of the operational HL3 semantics, as it is predetermined and discussed in section 3.2.

Below, we follow the structure of section 3.2, such that all components of $CONST$ are defined here in the same order as they appear in section 3.2. As an example for the resulting HL3 model, a C++ variant of $c_{tc} \in CONST$ of the case study "Radio-Based Train Control" is presented in appendix **??**, created by the implementation of transformation Φ_{HUML}.

The definition of code creation from HybEL expressions, which is particularly needed for the definition of programs for program subjects, will be given in section 6.5.

Finally, the definition of HybridUML-specific behavior for abstract subjects follows in section 6.6.

6.4.1 Entities

scheduler. This is the pre-defined HL3 scheduler, as for every HL3 model:

$$sched(c_{spec}) = scheduler$$

SELECTOR. The applied selector is the pre-defined HybridUML selector:

$$sel(c_{spec}) = selector_{HUML}$$

AM. The abstract machines of the system represent the sequential behavioral components, which in turn are defined by basic agent instance nodes $ain \in AIN_{basic}$. Therefore, for each ain, an abstract machine is created, i.e. there is a bijection

$$am_{HL3,AIN_{basic}} : AIN_{basic} \rightarrowtail\!\!\!\!\!\twoheadrightarrow Am$$

that defines the abstract machines of the HL3 model:

$$am(c_{spec}) = Am$$

The inverse function that maps abstract machines back to agent instance nodes is given as $ain_{Am} = am^{-1}_{HL3,AIN_{basic}}$.

IFM. For a HybridUML simulation, there are no interface modules:

$$ifm(c_{spec}) = \varnothing$$

FLOW. The flows of the HL3 model are given by the flow expression nodes. Each $fxn \in FlowExpN_{spec}$ defines a HL3 flow, i.e. there is a bijection (and the corresponding inverse)

$$flow_{HL3,FlowExpN_{spec}} : FlowExpN_{spec} \rightarrowtail\!\!\!\!\!\twoheadrightarrow Flow$$

$$fxn_{Flow} = flow^{-1}_{HL3,FlowExpN_{spec}}$$

This defines the flows of the HL3 model:

$$flow(c_{spec}) = Flow$$

TRANS. The transitions of the HL3 model are defined by the transition nodes, in that for each transition node a unique transition is generated:

$$trans_{HL3,TN_{spec}} : TN_{spec} \rightarrowtail\!\!\!\!\twoheadrightarrow Trans$$
$$tn_{trans} = trans^{-1}_{HL3,TN_{spec}}$$

Those are the transitions of the model:

$$trans(c_{spec}) = Trans$$

VAR. The local variables $var(c_{spec})$ are defined for the expression nodes $ExpN_{spec}$, such that they can be used by their generated programs. There are two kinds of variables – (1) *simple variables* and (2) *array variables.* All variables that represent HybridUML variables or signals are array variables, such that according to the multiplicity defined by the HybridUML variable or signal, the local HL3 variable contains a corresponding number of *subscripted variables.* In contrast, most of the additionally introduced HL3 variables are simple variables.

In the following, several mappings are given that define the available local HL3 variables. For every variable, its type is defined, too. The union of all types *Val* contains all HybridUML-specific data values:

$$VAL_{eval} \subseteq Val$$

HybridUML Properties. Every HybridUML property $v \in V \cup V_{local}$ (including bound variables of quantified expressions) that is read or written by the expression of an expression node is represented by a local variable. For the calculation of flow integration steps, a previous value may be needed:

$$lvar_{V,V_{local}} : ExpN \times (V \cup V_{local}) \times \{read, write\} \times \{cur, prev\} \nrightarrow Var$$

Since $lvar_{V,V_{local}}$ is a partial function, local variables are not generated for every tuple of the domain; precisely, local variables are created for each variable of expression nodes from the given HybridUML specification, (1) for which the current value is read, (2) for which the previous value is read, or (3) for which a (current) value is written:

$$\begin{aligned}
&\text{dom}\, lvar_{V,V_{local}} = \\
&\{(expn, v, read, cur) \mid expn \in ExpN_{spec} \wedge v \in var_{ht,read}(ht_{expn}(expn))\} \\
&\cup \{(expn, v, read, prev) \\
&\qquad \mid expn \in ExpN_{spec} \wedge v \in var_{ht,readPrev}(ht_{expn}(expn))\} \\
&\cup \{(expn, v, write, cur) \mid expn \in ExpN_{spec} \wedge v \in var_{ht,write}(ht_{expn}(expn))\}
\end{aligned}$$

The mapping $lvar_{V,V_{local}}$ is not injective, because for some expression nodes, the local variables that represent the same HybridUML variable coincide: As defined later, the sequence of actions of a transition node, along with its trigger, define a single shared program. Within, any sequence of read and write accesses to the same HybridUML variable v_{huml} may occur via local variables $v_1, \dots, v_n$ for expression nodes $expn_1, \dots, expn_n$. In order to ensure that every write access is effective for subsequent read accesses, exactly those variables are identical:

$$\begin{aligned}
&\forall (expn_1, v_1, acc_1, tm_1), (expn_2, v_2, acc_2, tm_2) \in \operatorname{dom} lvar_{V, V_{local}} \bullet \\
&\quad \{expn_1, expn_2\} \subseteq TExpN \land tn_{TExpN}(expn_1) = tn_{TExpN}(expn_2) \\
&\quad \land v_1 = v_2 \land tm_1 = tm_2 = cur \\
&\quad \Leftrightarrow \\
&\quad lvar_{V, V_{local}}(expn_1, v_1, acc_1, tm_1) = lvar_{V, V_{local}}(expn_2, v_2, acc_2, tm_2)
\end{aligned}$$

Note that also for a single expression node, this means that read and write access are realized with the same local variable.

The types of the created variables are defined by the types of the corresponding HybridUML variable:

$$\begin{aligned}
&\forall (expn, v, acc, tm) \in \operatorname{dom} lvar_{V, V_{local}} \bullet \\
&\quad type_{Var}(lvar_{V, V_{local}}(expn, v, acc, tm)) = \begin{cases} type_{HL3, V}(v) & ; \ v \in V \\ \mathbb{N}_0 \to \mathbb{Z} & ; \ v \in V_{local} \end{cases}
\end{aligned}$$

The types are defined for HybridUML variables and HybridUML data types. Since HybridUML variables have multiplicities, the valuation of a corresponding HL3 variable is given as a function of natural numbers to the type of the subscripted variables. The array variable's type is then the set of these functions. Formally, there is no upper bound for array indices of this array representation, but for an implementation, an upper bound defined by the multiplicity would be used explicitly.

$$\begin{aligned}
&type_{HL3, V} : V \to \mathcal{P}(Val) \\
&\quad v \mapsto (\mathbb{N}_0 \to type_{HL3, DT}(type_V(v))) \\
&type_{HL3, DT} : DT \to \mathcal{P}(Val) \\
&bool \mapsto \mathbb{B} \\
&int \mapsto \mathbb{Z} \\
&real \mapsto \mathbb{R} \\
&anaReal \mapsto \mathbb{R} \\
&et \mapsto lit_{DT}(et) \, ; \ et \in DT_{enum} \\
&sdt \mapsto \{\langle type_{HL3, DT}(varseq_{DT}(sdt)(1)), \ldots, \\
&\quad type_{HL3, DT}(varseq_{DT}(sdt)(n))\rangle \mid |varseq_{DT}(sdt)| = n\} \\
&\qquad ; \ sdt \in DT_{struc}
\end{aligned}$$

Note that bound variables from HybEL expressions are also of array type, such that the handling of them and HybridUML variables is the same. However, these are effectively simple variables, because there is only one subscripted variable inside.

HybridUML Signals. Every HybridUML signal $s \in S$ that is received or sent by the expression of a node is represented by a variable:

$$lvar_S : ExpN \times S \times \{read, write\} \rightarrowtail\!\!\!\!\rightarrow Var$$

The variables are of boolean type, such that the value *true* denotes that the signal is currently active, within the scope of the corresponding program. Programs can either read this value, such that for trigger expressions, the enabledness of the corresponding transition is affected, or write this value. This will be done for signal raise statements, in order to activate a signal for the complete system, or for trigger statements, in order to consume the signal locally.

$$\forall (expn, s, acc) \in \operatorname{dom} lvar_S \bullet type_{Var}(lvar_S(expn, s, acc)) = \mathbb{B}$$

Local variables are created for the combinations of signals and expression nodes that actually use them:

$$\begin{aligned}&\operatorname{dom} lvar_S = \\ &\{(expn, s, read) \mid expn \in ExpN_{spec} \wedge s \in sig_{ht,read}(ht_{expn}(expn))\} \\ &\cup \{(expn, s, write) \mid expn \in ExpN_{spec} \wedge v \in sig_{ht,write}(ht_{expn}(expn))\}\end{aligned}$$

Parameters of HybridUML Signals. For every HybridUML signal, there are up to two local variables for each parameter of the signal within the corresponding expression nodes, one variable for the reception and one for the sending of the parameter:

$$lvar_{SigParam} : ExpN \times S \times \{read, write\} \times \mathbb{N}_0 \rightarrowtail\!\!\!\!\!\!\!\rightarrow Var$$

For the combinations of signals and their parameter indices and expression nodes that actually use the signals, local variables are created:

$$\begin{aligned}&\operatorname{dom} lvar_{SigParam} = \\ &\{(expn, s, read, n) \mid expn \in ExpN_{spec} \wedge s \in sig_{ht,read}(ht_{expn}(expn)) \\ &\qquad \wedge 0 \leq n < |sn_{expn}(s, expn)|\} \\ &\cup \{(expn, s, write) \mid expn \in ExpN_{spec} \wedge v \in sig_{ht,write}(ht_{expn}(expn)) \\ &\qquad \wedge 0 \leq n < |sn_{expn}(s, expn)|\}\end{aligned}$$

The parameter's type is defined by its the HybridUML data type, as defined for HybridUML variables above:

$$\begin{aligned}&\forall (expn, s, acc, n) \in \operatorname{dom} lvar_{SigParam} \bullet type_{Var}(lvar_{SigParam}(expn, s, acc, n)) \\ &\qquad = type_{HL3,DT}(paramTypes_S(s)(n+1))\end{aligned}$$

Index Counters. For the reception of signals, a local *trigger index* variable is created that stores the signal index wrt. its multiplicity for which the signal's occurrence is received:

$$lvar_{trgIdx} : ExpN \times S \rightarrowtail\!\!\!\!\!\!\!\rightarrow Var$$

This is done for each trigger within a corresponding expression node:

$$\begin{aligned}&\operatorname{dom} lvar_{trgIdx} = \\ &\{(expn, s) \mid expn \in ExpN_{spec} \wedge s \in sig_{ht,read}(ht_{expn}(expn))\}\end{aligned}$$

A local *signal index* exists for the sending of signals:

$$lvar_{sigIdx} : ExpN \times S \rightarrowtail\!\!\!\!\!\!\!\rightarrow Var$$

A corresponding variable is generated for sent signals within a corresponding expression node:

$$\begin{aligned}&\operatorname{dom} \mathit{lvar}_{\mathit{sigIdx}} = \\ &\{(\mathit{expn}, s) \mid \mathit{expn} \in \mathit{ExpN}_{\mathit{spec}} \wedge s \in \mathit{sig}_{\mathit{ht},\mathit{write}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}))\}\end{aligned}$$

Index variables are of natural number type:

$$\begin{aligned}&\forall (\mathit{expn}, s) \in \operatorname{dom} \mathit{lvar}_{\mathit{trgIdx}} \bullet \mathit{type}_{\mathit{Var}}(\mathit{lvar}_{\mathit{trgIdx}}(\mathit{expn}, s)) = \mathbb{N}_0 \\ &\forall (\mathit{expn}, s) \in \operatorname{dom} \mathit{lvar}_{\mathit{sigIdx}} \bullet \mathit{type}_{\mathit{Var}}(\mathit{lvar}_{\mathit{sigIdx}}(\mathit{expn}, s)) = \mathbb{N}_0\end{aligned}$$

Derivative Helper Variables. From HybridUML variables that are used to access derivative values, local variables to store the publication time ticks of the current value and of the previous value are generated, as well as local variables to store the time difference.

$$\begin{aligned}&\mathit{lvar}_{V,\Delta t} : \mathit{ExpN} \times V \rightarrowtail\!\!\!\!\!\leftrightarrow \mathit{Var} \\ &\mathit{lvar}_{V,\mathit{curTck}} : \mathit{ExpN} \times V \rightarrowtail\!\!\!\!\!\leftrightarrow \mathit{Var} \\ &\mathit{lvar}_{V,\mathit{prevTck}} : \mathit{ExpN} \times V \rightarrowtail\!\!\!\!\!\leftrightarrow \mathit{Var}\end{aligned}$$

For combinations of read derivatives within expression nodes, variables for the previous time stamp and the difference exist:

$$\begin{aligned}&\operatorname{dom} \mathit{lvar}_{V,\mathit{prevTck}} = \operatorname{dom} \mathit{lvar}_{V,\Delta t} = \\ &\{(\mathit{expn}, v) \mid \mathit{expn} \in \mathit{ExpN}_{\mathit{spec}} \wedge v \in \mathit{var}_{\mathit{ht},\mathit{readDeriv}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}))\}\end{aligned}$$

The current time tick is also used for written derivatives:

$$\begin{aligned}&\operatorname{dom} \mathit{lvar}_{V,\mathit{curTck}} = \\ &\{(\mathit{expn}, v) \mid \mathit{expn} \in \mathit{ExpN}_{\mathit{spec}} \wedge v \in \mathit{var}_{\mathit{ht},\mathit{readDeriv}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn})) \\ &\qquad \cup \mathit{var}_{\mathit{ht},\mathit{writeDeriv}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}))\}\end{aligned}$$

The time tick variables hold *ModelTime* values, the time difference is a natural number:

$$\begin{aligned}&\forall (\mathit{expn}, v) \in \operatorname{dom} \mathit{lvar}_{V,\mathit{curTck}} \bullet \\ &\qquad \mathit{type}_{\mathit{Var}}(\mathit{lvar}_{V,\mathit{curTck}}(\mathit{expn}, v)) = \mathit{ModelTime} \\ &\forall (\mathit{expn}, v) \in \operatorname{dom} \mathit{lvar}_{V,\mathit{prevTck}} \bullet \\ &\qquad \mathit{type}_{\mathit{Var}}(\mathit{lvar}_{V,\mathit{prevTck}}(\mathit{expn}, v)) = \mathit{ModelTime} \\ &\forall (\mathit{expn}, v) \in \operatorname{dom} \mathit{lvar}_{V,\Delta t} \bullet \\ &\qquad \mathit{type}_{\mathit{Var}}(\mathit{lvar}_{V,\Delta t}(\mathit{expn}, v)) = \mathbb{N}_0\end{aligned}$$

Quantified Expression Results. The result of quantified boolean expressions is stored in an exclusive local variable:

$$\mathit{lvar}_{\mathit{quantRes}} : \mathit{ExpN} \times \mathrm{tree}_{\mathrm{o}}\, \mathit{HybelItem} \rightarrowtail\!\!\!\!\!\leftrightarrow \mathit{Var}$$

Therefore, a variable exists for each occurrence of a quantified expression in an expression node:

$$\begin{aligned}&\operatorname{dom} \mathit{lvar}_{\mathit{quantRes}} = \\ &\{(\mathit{expn}, (((\mathit{bool}, q), \mathit{val}), \mathit{sub})) \mid \mathit{expn} \in \mathit{ExpN}_{\mathit{spec}} \wedge \\ &\qquad (((\mathit{bool}, q), \mathit{val}), \mathit{sub}) \in \mathrm{subtrees}_{\mathrm{o},\mathrm{HybelItem}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn})) \wedge q \in \{\forall, \exists\}\}\end{aligned}$$

The result variable is of boolean type:

$$\forall (expn, htree) \in \mathrm{dom}\, lvar_{quantRes} \bullet type_{Var}(lvar_{quantRes}(expn, htree)) = \mathbb{B}$$

Visibility Set Helper Variables. A visibility set $v \in VisibilitySet$ is provided for the publication of written values. It gets the respective visibility set parameter for HL3 flows and HL3 transitions by default, but can be re-calculated for specific purposes:

$$lvar_{visSet} : ExpN \mathrel{+\!\!\!\rightarrow\!\!\!\rightarrow} Var$$

A corresponding variable exists for flow expression nodes and for action and trigger expression nodes. The variables for actions and triggers coincide for common transitions:

$$\begin{aligned}
&\mathrm{dom}\, lvar_{visSet} = TrgExpN_{spec} \cup ActExpN_{spec} \cup FlowExpN_{spec}\\
&\forall\, expn_1, expn_2 \in \mathrm{dom}\, lvar_{visSet} \bullet\\
&\quad lvar_{visSet}(expn_1) = lvar_{visSet}(expn_2) \Leftrightarrow\\
&\quad \{expn_1, expn_2\} \subseteq TrgExpN_{spec} \cup ActExpN_{spec}\\
&\qquad \wedge tn_{TExpN}(expn_1) = tn_{TExpN}(expn_2)\\
&\quad \vee\, expn_1 = expn_2
\end{aligned}$$

For the calculation of visibility sets, several local helper variables are needed. They are used in the context of the assignment of a derivative variable, and for resetting a received signal on its consumption. (1) A visibility variable stores a single visibility $vis \in Visibility$. It is only used for expression nodes that either assign a derivative or reset a signal:

$$\begin{aligned}
&lvar_{vis} : ExpN \rightarrowtail\!\!\!\!\rightarrow\!\!\!\!\!+ Var\\
&\mathrm{dom}\, lvar_{vis} = \mathrm{dom}\, lvar_{visSet} \cap (TrgExpN_{spec} \cup\\
&\quad \{expn \in ExpN_{spec} \mid \exists\, v \in V \bullet v \in var_{ht,writeDeriv}(ht_{expn}(expn))\})
\end{aligned}$$

(2) The publication time tick of the visibility is stored in a local variable:

$$\begin{aligned}
&lvar_{newTck} : ExpN \rightarrowtail\!\!\!\!\rightarrow\!\!\!\!\!+ Var\\
&\mathrm{dom}\, lvar_{newTck} = \mathrm{dom}\, lvar_{vis}
\end{aligned}$$

(3) For flow integration operations, the entries of the pre-defined visibility set are iterated, by means of a visibility index:

$$\begin{aligned}
&lvar_{visIdx} : ExpN \rightarrowtail\!\!\!\!\rightarrow\!\!\!\!\!+ Var\\
&\mathrm{dom}\, lvar_{visIdx} = \mathrm{dom}\, lvar_{visSet} \cap\\
&\quad \{expn \in ExpN_{spec} \mid \exists\, v \in V \bullet v \in var_{ht,writeDeriv}(ht_{expn}(expn))\}
\end{aligned}$$

(4) In order to read the pre-defined visibility set for iteration, a local visibility set variable is provided:

$$\begin{aligned}
&lvar_{visOrig} : ExpN \rightarrowtail\!\!\!\!\rightarrow\!\!\!\!\!+ Var\\
&\mathrm{dom}\, lvar_{visOrig} = \mathrm{dom}\, lvar_{visIdx}
\end{aligned}$$

The types of variables are obvious:

$$\begin{aligned}
&\forall\, expn \in \operatorname{dom} lvar_{visSet} \bullet type_{Var}(lvar_{visSet}(expn)) = VisibilitySet\\
&\forall\, expn \in \operatorname{dom} lvar_{vis} \bullet type_{Var}(lvar_{vis}(expn)) = Visibility\\
&\forall\, expn \in \operatorname{dom} lvar_{visIdx} \bullet type_{Var}(lvar_{visIdx}(expn)) = \mathbb{N}_0\\
&\forall\, expn \in \operatorname{dom} lvar_{newTck} \bullet type_{Var}(lvar_{newTck}(expn)) = ModelTime\\
&\forall\, expn \in \operatorname{dom} lvar_{visOrig} \bullet type_{Var}(lvar_{visOrig}(expn)) = VisibilitySet
\end{aligned}$$

Non-Deterministic Assignment Helper Variables. For each non-deterministic assignment of integers, there is a local variable that counts the number of possible values during evaluation, before one is chosen:

$$lvar_{hitCount} : ExpN \times \mathrm{tree_o}\, HybelItem \rightarrowtail\!\!\!\!\!\rightarrow Var$$

Each possible value is stored itself in a corresponding array variable:

$$lvar_{hits} : ExpN \times \mathrm{tree_o}\, HybelItem \rightarrowtail\!\!\!\!\!\rightarrow Var$$

Similarly to quantified expressions, a variable exists for each occurrence of a non-deterministic assignment in an expression node:

$$\begin{aligned}
&\operatorname{dom} lvar_{hitCount} = \operatorname{dom} lvar_{hits} =\\
&\{(expn, (((int, intNondetAss), val), sub)) \mid expn \in ExpN_{spec} \land\\
&\qquad (((int, intNondetAss), val), sub) \in \mathrm{subtrees_{o,HybelItem}}(ht_{expn}(expn))\}
\end{aligned}$$

The hit counter holds a natural number, the hit array stores natural-numbered indices:

$$\begin{aligned}
&\forall (expn, htree) \in \operatorname{dom} lvar_{hitCount} \bullet type_{Var}(lvar_{hitCount}(expn, htree) = \mathbb{N}_0\\
&\forall (expn, htree) \in \operatorname{dom} lvar_{hits} \bullet type_{Var}(lvar_{hits}(expn, htree) = (\mathbb{N}_0 \rightarrow \mathbb{N}_0)
\end{aligned}$$

Boolean Result of Programs. Programs that represent the calculation of boolean expressions of the HybridUML model have a dedicated *return variable.* The boolean expressions from the model are trigger expressions, guard expressions, and invariant constraints:

$$\begin{aligned}
BExpN_{spec} &= TrgExpN_{spec} \cup GrdExpN_{spec} \cup InvExpN_{spec} \cup IscExpN_{spec}\\
&\subseteq TExpN \cup MExpN \cup AExpN
\end{aligned}$$

The return variables are identified by the injection

$$retvar_{BExpN} : BExpN_{spec} \rightarrowtail Var$$

They are of boolean type:

$$\forall\, expn \in \operatorname{dom} retvar_{BExpN} \bullet type_{Var}(retvar_{BExpN}(expn)) = \mathbb{B}$$

Local Variables of the HL3 Model. The ranges of all the above mappings are disjoint, such that there are unique variables for all described purposes. Then, their union define the set of local variables:

$$\begin{aligned}
&var(c_{spec}) = \\
&\quad \operatorname{ran} lvar_{V,V_{local}} \cup \operatorname{ran} lvar_S \cup \operatorname{ran} lvar_{SigParam} \\
&\quad \cup \operatorname{ran} lvar_{trgIdx} \cup \operatorname{ran} lvar_{sigIdx} \cup \operatorname{ran} lvar_{V,\Delta t} \\
&\quad \cup \operatorname{ran} lvar_{V,curTck} \cup \operatorname{ran} lvar_{V,prevTck} \\
&\quad \cup \operatorname{ran} lvar_{quantRes} \cup \operatorname{ran} lvar_{vis} \cup \operatorname{ran} lvar_{visSet} \\
&\quad \cup \operatorname{ran} lvar_{visIdx} \cup \operatorname{ran} lvar_{newTck} \cup \operatorname{ran} lvar_{visOrig} \\
&\quad \cup \operatorname{ran} lvar_{hitCount} \cup \operatorname{ran} lvar_{hits} \\
&\quad \cup \operatorname{ran} retvar_{BExpN}
\end{aligned}$$

CHAN. The channels of the HL3 model are defined by the set of connected property nodes or signal nodes, respectively. Each of such a set identifies a unique variable or signal channel. Additionally, for the parameters of signals there are separate channels that carry the parameter values, such that three disjoint sets of channels CHN_{Var}, CHN_{Sig}, and $CHN_{SigParam}$ with $CHN_{Var} \cap CHN_{Sig} \cap CHN_{SigParam} = \varnothing$ result:

$$\begin{aligned}
&chan_{VN_{conn,max}} : VN_{conn,max} \rightarrowtail\!\!\!\!\twoheadrightarrow CHN_{Var} \\
&chan_{SN_{conn,max}} : SN_{conn,max} \rightarrowtail\!\!\!\!\twoheadrightarrow CHN_{Sig} \\
&vn_{conn,max,CHN_{Var}} = chan^{-1}_{VN_{conn,max}} \\
&sn_{conn,max,CHN_{Sig}} = chan^{-1}_{SN_{conn,max}}
\end{aligned}$$

The sets of parameter channels are defined by the sets of connected signal nodes. The number of parameter channels is determined by the number of parameters of the respective signal.

$$\begin{aligned}
&chan_{SN_{conn,max},param} : SN_{conn,max} \times \mathbb{N}_0 \nrightarrow\!\!\!\!\twoheadrightarrow CHN_{SigParam} \\
&\operatorname{dom} chan_{SN_{conn,max},param} = \{(snset, n) \mid \exists\, sn \in snset \bullet \\
&\quad 0 \leq |paramTypes_S(sig_{SN}(sn))|\}
\end{aligned}$$

The parameter channels, like variable and signal channels, are unique, i.e. $chan_{SN_{conn,max},param}$ is also injective.

Then, the set of channels is the union of variable and signal channels and the signal parameter channels:

$$chan(c_{spec}) = CHN_{Var} \cup CHN_{Sig} \cup CHN_{SigParam}$$

PORT. The model's variable ports $PORT_{Var}$ and signal ports $PORT_{Sig}$ are created such that there is one port for each property node or signal node that is attached to a basic agent instance node:

$$\begin{aligned}
&VN_{basic} = \{vn \in VN_{spec} \mid \exists\, ain \in AIN_{basic} \bullet vn \in vn_{AIN}(tree_{AIN}(ain))\} \\
&SN_{basic} = \{sn \in SN_{spec} \mid \exists\, ain \in AIN_{basic} \bullet sn \in sn_{AIN}(tree_{AIN}(ain))\} \\
&port_{VN_{basic}} : VN_{basic} \rightarrowtail\!\!\!\!\twoheadrightarrow PORT_{Var} \\
&port_{SN_{basic}} : SN_{basic} \rightarrowtail\!\!\!\!\twoheadrightarrow PORT_{Sig}
\end{aligned}$$

$$vn_{PORT_{Var}} = port^{-1}_{VN_{basic}}$$
$$sn_{PORT_{Sig}} = port^{-1}_{SN_{basic}}$$

The ports representing signal parameters $PORT_{SigParam}$ are defined by the corresponding signal node, along with the parameter's index in the signal's parameter list:

$$port_{SN_{basic},param} : SN_{basic} \times \mathbb{N}_0 \twoheadrightarrow\!\!\!\!\!\!\!+\ PORT_{SigParam}$$
$$\operatorname{dom} port_{SN_{basic},param} = \{(sn, n) \mid 0 \leq |paramTypes_S(sig_{SN}(sn))|\}$$

The parameter port mapping $port_{SN_{basic},param}$ is injective, like the ones for variable and signal ports.

The set of all ports is defined as the union of the disjoint sets of variable ports, signal ports, and signal parameter ports, i.e.

$$port(c_{spec}) = Port = PORT_{Var} \cup PORT_{Sig} \cup PORT_{SigParam}$$

with $PORT_{Var} \cap PORT_{Sig} \cap PORT_{SigParam} = \varnothing$.

LWP. The set of light weight processes depends on the number of available processors within the hardware system. It cannot be defined by the transformation Φ automatically, but has to be configured manually, such that

$$|lwp(c_{spec})| = n$$

for n available processors.

6.4.2 Dependencies

Am$_{\textbf{Trans}}$. The mapping from transitions to abstract machines is defined by the assignment of transition nodes to agent instance nodes:

$$am_{huml,trans} : Trans \rightarrow Am$$
$$t \mapsto am_{HL3,AIN_{basic}}(ain_{MIN}(min_{TN}(tn_{trans}(t))))$$

Therefore, $am_{trans}(c_{spec}) = am_{huml,trans}$ holds.

Am$_{\textbf{Flow}}$. The assignment of flows to abstract machines for the HL3 model is given by the flow expression nodes that define the flows:

$$am_{huml,flow} : Flow \rightarrow Am$$
$$f \mapsto am_{HL3,AIN_{basic}}(ain_{MIN}(min_{MExpN}(fxn_{Flow}(f))))$$

Hence, $am_{flow}(c_{spec}) = am_{huml,flow}$ holds.

Subject$_{\textbf{Var}}$. The mapping of variables the their responsible subject is essentially given by the involved expression node:

$$subject_{ExpN_{spec}} : ExpN_{spec} \rightarrow Flow \cup Trans \cup Am$$
$$expn \mapsto \begin{cases} flow_{HL3,FlowExpN_{spec}}(expn) & ;\ expn \in FlowExpN_{spec} \\ trans_{HL3,TN_{spec}}(tn_{TExpN}(expn)) & ;\ expn \in ActExpN_{spec} \\ am_{HL3,AIN_{basic}}(ain_{ExpN}(expn)) & ;\ \text{else} \end{cases}$$

Then, the local variables are mapped to subjects, corresponding to the expression nodes that are mapped the local variables themselves:

$$
\begin{aligned}
&\mathit{subject}_{lvar} : \mathit{Var} \to \mathit{Flow} \cup \mathit{Trans} \cup \mathit{Ifm} \cup \mathit{Am} \\
&\mathit{lv} \mapsto \mathit{subject}_{ExpN_{spec}}(\mathit{expn}) \\
&\text{with} \\
&\quad \exists(v, \mathit{acc}, \mathit{tm}) \in V \times \{\mathit{read}, \mathit{write}\} \times \{\mathit{cur}, \mathit{prev}\} \bullet \\
&\qquad (\mathit{expn}, v, \mathit{acc}, \mathit{tm}) \mapsto \mathit{lv} \in \mathit{lvar}_{V, V_{local}} \\
&\quad \vee \exists(s, \mathit{acc}) \in S \times \{\mathit{read}, \mathit{write}\} \bullet (\mathit{expn}, s, \mathit{acc}) \mapsto \mathit{lv} \in \mathit{lvar}_S \\
&\quad \vee \exists(s, \mathit{acc}, n) \in S \times \{\mathit{read}, \mathit{write}\} \times \mathbb{N}_0 \bullet \\
&\qquad (\mathit{expn}, s, \mathit{acc}, n) \mapsto \mathit{lv} \in \mathit{lvar}_{SigParam} \\
&\quad \vee \exists\, s \in S \bullet (\mathit{expn}, s) \mapsto \mathit{lv} \in \mathit{lvar}_{trgIdx} \\
&\quad \vee \exists\, s \in S \bullet (\mathit{expn}, s) \mapsto \mathit{lv} \in \mathit{lvar}_{sigIdx} \\
&\quad \vee \exists\, v \in V \bullet (\mathit{expn}, v) \mapsto \mathit{lv} \in \mathit{lvar}_{V,\Delta t} \\
&\quad \vee \exists\, v \in V \bullet (\mathit{expn}, v) \mapsto \mathit{lv} \in \mathit{lvar}_{V,curTck} \\
&\quad \vee \exists\, v \in V \bullet (\mathit{expn}, v) \mapsto \mathit{lv} \in \mathit{lvar}_{V,prevTck} \\
&\quad \vee \exists\, t \in \mathrm{tree}_{\mathrm{o}}\, \mathit{HybelItem} \bullet (\mathit{expn}, t) \mapsto \mathit{lv} \in \mathit{lvar}_{quantRes} \\
&\quad \vee \mathit{expn} \mapsto \mathit{lv} \in \mathit{lvar}_{vis} \\
&\quad \vee \mathit{expn} \mapsto \mathit{lv} \in \mathit{lvar}_{visSet} \\
&\quad \vee \mathit{expn} \mapsto \mathit{lv} \in \mathit{lvar}_{visIdx} \\
&\quad \vee \mathit{expn} \mapsto \mathit{lv} \in \mathit{lvar}_{newTck} \\
&\quad \vee \mathit{expn} \mapsto \mathit{lv} \in \mathit{lvar}_{visOrig} \\
&\quad \vee \exists\, t \in \mathrm{tree}_{\mathrm{o}}\, \mathit{HybelItem} \bullet (\mathit{expn}, t) \mapsto \mathit{lv} \in \mathit{lvar}_{hitCount} \\
&\quad \vee \exists\, t \in \mathrm{tree}_{\mathrm{o}}\, \mathit{HybelItem} \bullet (\mathit{expn}, t) \mapsto \mathit{lv} \in \mathit{lvar}_{hits} \\
&\quad \vee \mathit{expn} \mapsto \mathit{lv} \in \mathit{retvar}_{BExpN}
\end{aligned}
$$

The above mapping defines the subject dependency for local variables for the particular HybridUML specification:

$$\mathit{subject}_{var}(c_{spec}) = \mathit{subject}_{lvar}$$

Chan$_{\mathbf{Port}}$. Each port is mapped to the channel for which it provides access. This is defined by the sets of connected property or signal nodes: (1) Each node is contained in at most one maximal set of nodes, since every two sets of property nodes are united for a common property node within VN_{conn}, and (2) each node is contained at least in one such set, because every port is collected within $VN_{conn,local}$.

$$
\begin{aligned}
&\mathit{chan}_{huml,port} : \mathit{Port} \to \mathit{Chan} \\
&p \mapsto c \\
&\qquad ; p \in PORT_{Var} \wedge c \in CHN_{Var} \wedge vn_{PORT_{Var}}(p) \in vn_{conn,max,CHN_{Var}}(c) \\
&p \mapsto c \\
&\qquad ; p \in PORT_{Sig} \wedge c \in CHN_{Sig} \wedge sn_{PORT_{Sig}}(p) \in sn_{conn,max,CHN_{Sig}}(c)
\end{aligned}
$$

Then, this *is* the mapping from ports to channels:

$$\mathit{chan}_{port}(c_{spec}) = \mathit{chan}_{huml,port}$$

InitVal$_{\textbf{Port}}$. For the definition of initial port values, HybridUML provides two complementary concepts, that were introduced in section 5.1 – (1) “Initial Property Values” and (2) “Init State Constraints”:

Init State Constraints. An init state constraint is a boolean expression that defines a constraint on an agent instance node that must be satisfied in the HL3 model's initial state. All properties of the agent are available for the expression, therefore arbitrary relations on them can be postulated. This provides a powerful means to define a set of valid initial HL3 channel states.

In general, such a set of constraints cannot always be solved easily, and the definition of a constraint solver (for an adequate subset of init state constraints) is out of scope of this thesis. However, we provide a mechanism to guarantee that HL3 models are only executed, when the initial channel state satisfies the init state constraints. This is done by the HybridUML selector, which is defined in section 6.6.2.

Initial Property Values. Initial property value specifications are provided as a less powerful, but straight-forward mechanism to define the initial state of HL3 channels. For each HybridUML property (of agents), an independent value specification can be given, such that mainly, a single initial channel state can be defined explicitly.

Since initial values are attached to properties, there can be more than one initial value:

$$
\begin{aligned}
&\sigma_{CHN_{Var},initChoice} : CHN_{Var} \to \mathcal{P}(VAL_{eval})\\
&\qquad c \mapsto \{v \in VAL_{eval} \mid \exists\, vn \in c \bullet v = \sigma_{VN_{spec}}(vn)\}
\end{aligned}
$$

One of the provided initial values is chosen non-deterministically, if there is any:

$$
\begin{aligned}
&\sigma_{CHN_{Var},init} : CHN_{Var} \to VAL_{eval} \cup \{\lambda\}\\
&\qquad c \mapsto v;\ v \in \sigma_{CHN_{Var},initChoice}(c)\\
&\qquad c \mapsto \lambda\,;\ \sigma_{CHN_{Var},initChoice}(c) = \varnothing
\end{aligned}
$$

For HybridUML, every variable port of a channel initially has access to the same value – the channel's initial value. If the channel does not provide an initial value, then a default value, corresponding to the port's type, is used. Note that this overrides the (possible) use of init state constraints discussed above. An appropriate constraint solver would be integrated here, such that channels without initial value would get their default values then from the solver.

$$
\begin{aligned}
&\sigma_{PORT_{Var},init} : PORT_{Var} \to VAL_{eval}\\
&\qquad p \mapsto \sigma_{CHN_{Var},init}(chan_{huml,port}(p))\\
&\qquad\quad ;\ \sigma_{CHN_{Var},init}(chan_{huml,port}(p)) \neq \lambda\\
&\qquad p \mapsto defval_{DT}(type_V(var_{VN}(vn_{PORT_{Var}}(p))))\\
&\qquad\quad ;\ \sigma_{CHN_{Var},init}(chan_{huml,port}(p)) = \lambda
\end{aligned}
$$

The default value is deterministically defined for primitive and structured data types, and non-deterministically chosen for enumeration types:

$$
\begin{aligned}
& \mathit{defval}_{DT} : DT \to \mathit{VAL}_{eval} \\
& \quad \mathit{bool} \mapsto \mathit{false} \\
& \quad pt \mapsto 0;\ pt \in \{\mathit{int}, \mathit{real}, \mathit{anaReal}\} \\
& \quad \mathit{sdt} \mapsto \langle \mathit{defval}_{DT}(\mathit{varseq}_{DT}(\mathit{sdt})(1)), \ldots, \\
& \quad\quad \mathit{defval}_{DT}(\mathit{varseq}_{DT}(\mathit{sdt})(|\mathit{varseq}_{DT}(\mathit{sdt})|))\rangle \\
& \quad\quad ;\ \mathit{kind}_{DT}(\mathit{sdt}) = \mathit{struc} \\
& \quad \mathit{et} \mapsto l;\ \mathit{kind}_{DT}(\mathit{et}) = DT_{enum} \wedge \mathit{dt}_L(l) = \mathit{et}
\end{aligned}
$$

Signal ports initially get the value *false*, i.e. initially there is no occurrence of any signal.

$$
\begin{aligned}
& \sigma_{PORT_{Sig},init} : PORT_{Sig} \to \mathit{VAL}_{eval} \\
& \quad p \mapsto \mathit{false}
\end{aligned}
$$

The initial port data of the HL3 model is then defined by the initial values above:

$$
\begin{aligned}
& \mathit{data}_{PORT_{Var},init} : PORT_{Var} \to \mathit{Data} \\
& \quad p \mapsto \mathit{data}_{Val}(\sigma_{PORT_{Var},init}(p)) \\
& \mathit{data}_{PORT_{Sig},init} : PORT_{Sig} \to \mathit{Data} \\
& \quad p \mapsto \mathit{data}_{Val}(\sigma_{PORT_{Sig},init}(p))
\end{aligned}
$$

The union of initial data for variable ports and signal ports gives the complete port initialization function:

$$\mathit{initval}_{port}(c_{spec}) = \mathit{data}_{PORT_{Var},init} \cup \mathit{data}_{PORT_{Sig},init}$$

Subject$_{\mathbf{Port}}$. Every port of the model is accessible for the abstract machine *am* from which it originates, i.e. that is created from the basic agent instance node that hosts the property node or signal node, from which the port is generated. Further, all flows and transitions that are associated with *am* have access, too:

$$
\begin{aligned}
& \mathit{subject}_{huml,port} : \mathit{Port} \to \mathcal{P}(\mathit{Subject}) \\
& p \mapsto \{am\} \cup \{t \in \mathit{Trans} \mid am_{huml,trans}(t) = am\} \\
& \quad \cup \{f \in \mathit{Flow} \mid am_{huml,flow}(f) = am\} \\
& ;\ p \in PORT_{Var} \wedge am = am_{HL3,AIN_{basic}}(\mathit{ain}_{VN}(\mathit{vn}_{PORT_{Var}}(p))) \\
& \quad \vee p \in PORT_{Sig} \wedge am = am_{HL3,AIN_{basic}}(\mathit{ain}_{SN}(\mathit{sn}_{PORT_{Sig}}(p)))
\end{aligned}
$$

The mapping from ports to subjects is then defined as

$$\mathit{subject}_{port}(c_{spec}) = \mathit{subject}_{huml,port}$$

SelPort. The HybridUML selector only accesses channels (via ports) in order to deactivate signals. Therefore, it has access to the signal ports:

$$\mathit{selport}(c_{spec}) = PORT_{Sig}$$

VisibilitySet$_{\mathbf{Flow}}$. The publication visibility for HybridUML flows is neither restricted nor delayed. That is, for every flow, all ports are contained here such that it may publish its results to whatever recipient it may address. Further, the delay time stamp is set to zero for all entries:

$$\begin{aligned}&\mathit{vis}_{\mathit{huml,flow}} : \mathit{Flow} \to \mathit{VisibilitySet}\\ &\quad f \mapsto \{0.0\} \times \mathit{Port}\end{aligned}$$

Hence, $\mathit{vis}_{\mathit{flow}}(c_{\mathit{spec}}) = \mathit{vis}_{\mathit{huml,flow}}$ holds.

VisibilitySet$_{\mathbf{Ifm}}$. The mapping of interface modules to visibility sets is empty, since there are no interface modules:

$$\mathit{vis}_{\mathit{ifm}}(c_{\mathit{spec}}) = \varnothing$$

VisibilitySet$_{\mathbf{Trans}}$. The publication visibility for HybridUML flows is neither restricted nor delayed, similarly to flows:

$$\begin{aligned}&\mathit{vis}_{\mathit{huml,trans}} : \mathit{Trans} \to \mathit{VisibilitySet}\\ &\quad t \mapsto \{0.0\} \times \mathit{Port}\end{aligned}$$

Thus, $\mathit{vis}_{\mathit{trans}}(c_{\mathit{spec}}) = \mathit{vis}_{\mathit{huml,trans}}$ holds.

Lwp$_{\mathbf{Subj_{abs}}}$. Since the set of light weight processes is configured manually, the assignment of abstract subjects to light weight processes $\mathit{lwp}_{\mathit{subj}}(c_{\mathit{spec}})$ has to be done manually, too.

Program$_{\mathbf{Subj_{prog}}}$. The programs that define the behavior of the program subjects of the HL3 model are created from the associated expressions. The program subjects are the flows $f \in \mathit{Flow}$ and the transitions $t \in \mathit{Trans}$. There are no interface modules. The definitions for program creation from expression nodes are given in section 6.5. Here, the assignment from subjects and their respective operations to the resulting programs is defined: (1) The flows' programs are created from the expressions of the corresponding flow expression nodes, (2) the transitions' programs are based on the action expressions of the corresponding transition nodes, and of the transition expression.

$$\begin{aligned}&\mathit{prg}_{\mathit{huml,subj}} : (\mathit{Flow} \times \mathit{Op}_{\mathit{Flow}}) \cup (\mathit{Trans} \times \mathit{Op}_{\mathit{Trans}}) \cup (\mathit{Ifm} \times \mathit{Op}_{\mathit{Ifm}})\\ &\quad \to \mathit{Program}\\ &(f, \mathit{integrate}) \mapsto \mathit{prg}_{\mathit{ExpN,ass}}(\langle \mathit{fxn}_{\mathit{Flow}}(f)\rangle)\\ &(t, \mathit{action}) \mapsto\\ &\quad \mathit{prg}_{\mathit{ExpN,ass}}(\mathrm{anyseq}_{\mathrm{ExpN}}(\mathit{trg}_{\mathit{TN_{spec}}}(\mathit{tn}_{\mathit{trans}}(t))) \frown \mathit{act}_{\mathit{TN_{spec}}}(\mathit{tn}_{\mathit{trans}}(t)))\end{aligned}$$

As a result, $\mathit{prg}_{\mathit{subj}}(c_{\mathit{spec}}) = \mathit{prg}_{\mathit{huml,subj}}$ holds.

6.4.3 Physical Constraints

As stated in section **??**, HybridUML has no specification facility for the definition of physical constraints. Therefore, the system period has to be configured manually, and the period factors for flows are set to a fixed default value.

SysPeriod. The internal scheduling period $\delta_{period}(c_{spec})$ has to be configured manually, and is constrained by two factors:

1. The performance of the available hardware must be sufficient, i.e. fast enough for the chosen period.

2. The chosen period must be appropriate for the simulation's requirements. Mostly, this means that it must be small enough.

Note that these constraints are conflicting in general. Therefore, it is not ensured that a valid period $\delta_{period}(c_{spec})$ can be found.

Period$_{\mathbf{Flow}}$. By default, every flow is scheduled with the system period $\delta_{period}(c_{spec})$, i.e. the factor is set to 1 for all flows:

$$\begin{aligned}\mathit{period}_{\mathit{huml,flow}} &: \mathit{Flow} \to \mathbb{N}\\ f &\mapsto 1\end{aligned}$$

This mapping can be adjusted manually. Anyway, $\mathit{period}_{\mathit{flow}}(c_{spec}) = \mathit{period}_{\mathit{huml,flow}}$ holds.

Period$_{\mathbf{Ifm,poll}}$. Since there are no interface modules, the mapping from interface modules to period factors for the operation *poll* is empty:

$$\mathit{period}_{\mathit{ifm,poll}}(c_{spec}) = \varnothing$$

Period$_{\mathbf{Ifm,tmit}}$. Similarly, the mapping from interface modules to period factors for the operation *transmit* is empty:

$$\mathit{period}_{\mathit{ifm,tmit}}(c_{spec}) = \varnothing$$

6.5 Code Creation for Expression Nodes

This section defines the creation of program code for sequences of expression nodes. The resulting code defines the execution semantics of a sequence of expressions, within their expression node context. In particular, the semantics of single expression nodes are therefore given.

Exemplarily, for selected generation rules, corresponding program code is pointed out in appendix **??**. There, from the implementation of the transformation Φ_{HUML}, C++ operations are given, which correspond to programs of the HL3 model of the case study "Radio-Based Train Control".

Note that the C++ code differs from the formally defined programs $p \in$ *Program* in details. For example, declaration of local variables is done within the code, not separately, and initialization is merged with declaration. Further, some expressions and statements are represented more conveniently, like `for`-statements, instead of `while`.

Program Strings. The transformation rules given in this section define code by giving *program strings*, that are sequences $s \in \mathit{String}$ of characters. For convenience, we use the conventional string notation "`abc`" from programming languages as a shorthand notation for the sequence $\langle a, b, c\rangle$.

For the representation of literals, local variables, and ports we assume an injective mapping

$$\mathit{string} : \mathit{VAL}_{eval} \cup \mathit{Var} \cup \mathit{Port} \rightarrowtail \mathit{String}$$

that defines a unique string representation for each literal, variable, and port.

In terms of the HL3 operational semantics, programs are represented as sequences $p \in \mathit{Program}$ of *program statements.* The mapping

$$\mathit{prg}_{\mathit{String}} : \mathit{String} \to \mathit{Program}$$

from program strings to sequences of statements is straight-forward, using the character "; " as statement delimiter, as well as the standard block notation for conditional and loop statements, ending with a closing brace "}". We omit the formal definition here.

Evaluation and Assignment Interpretation of Expressions. There are two different interpretations of expressions in the context of HybridUML specifications:

Evaluation Interpretation Some expressions have an evaluation interpretation, such that the corresponding program calculates a result value when it is executed. The result value of complete expressions that are contained in expression nodes is always of boolean type, whereas contained subexpressions of course can have other types, as defined in chapter 5.
(1) Trigger expressions, (2) guard expressions, and (3) invariant constraint expressions have evaluation interpretations that determine whether a trigger is active, or a guard or an invariant constraint is satisfied, respectively.

Assignment Interpretation The assignment interpretation of expressions defines programs that are executed like procedure calls and do not return a result value. Those programs will typically calculate new values for HybridUML variables or set/unset HybridUML signals and publish the new values on the respective channels.
The affected kinds of expressions are (1) action expressions, (2) flow (and algebraic) constraint expressions, and (3) trigger expressions. The effect of the assignment interpretation of triggers is that the corresponding signal is unset after it has been consumed.

The definition of the resulting programs given below is structured accordingly: On the basis of the hybel item trees and identifier item trees which represent (sub-)expressions, the mappings below are defined mutually recursively, such that particularly assignment code can contain evaluation code for the calculation of new values.

Evaluation Code of Expression Nodes. For a sequence of expression nodes, the code that implements the evaluation interpretation is given by the function

$$\mathit{prg}_{\mathit{ExpN},eval} : \operatorname{seq} \mathit{ExpN}_{spec} \to \mathit{Program}$$

For expression nodes with contained expression of boolean type, i.e. for

$$\begin{aligned}&\forall\, \mathit{expn} \in \{\mathit{expn}_1, \ldots, \mathit{expn}_n\} \bullet \exists(((t, r), \mathit{val}), \mathit{sub}) \in \mathrm{tree}_{\mathrm{o}}\, \mathit{HybelItem} \bullet\\ &\qquad \mathit{ht}_{\mathit{expn}}(\mathit{expn}) = (((t, r), \mathit{val}), \mathit{sub}) \wedge t = \mathit{bool} \wedge r \in \{\mathit{op}, \forall, \exists, \mathit{var}, \mathit{lit}\}\end{aligned}$$

the function $\mathit{prg}_{\mathit{ExpN},\mathit{eval}}$ is defined as

$$\begin{aligned}&\langle \mathit{expn}_1, \ldots, \mathit{expn}_n\rangle \mapsto \mathit{prg}_{\mathit{String}}(\\ &\qquad \mathit{initcode}_{V,\mathit{cur}}(\{\mathit{expn}_1, \ldots, \mathit{expn}_n\}) ^\frown\\ &\qquad \mathit{initcode}_{V,\mathit{prev}}(\{\mathit{expn}_1, \ldots, \mathit{expn}_n\}) ^\frown \mathit{initcode}_S(\{\mathit{expn}_1, \ldots, \mathit{expn}_n\}) ^\frown\\ &\qquad \mathit{pre}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}_1), \mathit{cur}, \mathit{expn}_1) ^\frown \ldots ^\frown \mathit{pre}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}_n), \mathit{cur}, \mathit{expn}_n)\\ &\qquad ^\frown \mathit{string}(\mathit{retvar}_{\mathit{BExpN}}(\mathit{expn}_1)) ^\frown \text{"}{:=}\text{"} ^\frown\\ &\qquad \mathit{main}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}_1), \mathit{cur}, \mathit{expn}_1)\\ &\qquad ^\frown \text{"}\wedge\text{"} ^\frown \ldots ^\frown \text{"}\wedge\text{"} ^\frown\\ &\qquad \mathit{main}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}_n), \mathit{cur}, \mathit{expn}_n)\\ &\qquad ^\frown \text{"};\text{"} ^\frown\\ &\qquad \mathit{post}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}_1), \mathit{cur}, \mathit{expn}_1) ^\frown \ldots ^\frown \mathit{post}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}_n), \mathit{cur}, \mathit{expn}_n))\end{aligned}$$

At the beginning, local variables are initialized which are accessed by the expressions of the expression nodes. For the evaluation code, (1) current values of HybridUML variables, (2) previous values of HybridUML variables, and (3) signal states, i.e. enabledness of signals are read from the respective channels and stored locally in dedicated HL3 variables. The actual evaluation of an expression consists of up to three parts:

Main Code The main evaluation code is an expression in terms of while-programs. Simple HybEL expressions are represented directly by such an expression; for example, the boolean literal *true* is represented as the program expression "true".
In contrast, more complex HybEL expressions are implemented by a sequential sub-program and cannot be given by a simple program expression. Examples are the quantifiers ∀, ∃ – they are mapped to while-loops which store the result in a local variable *result*. Here, "result" constitutes the main code of the quantified HybEL expression, whereas the sub-program that is run before is the *pre-code*.

Pre-Code The evaluation pre-code subsumes all sub-programs (concatenated sequentially) that are needed in order to prepare the expression of the main code.

Post-Code For expressions that require some functionality to be effective *after* the evaluation is done, the post-code can be created and filled with the respective program string. This will be used for the *assignment interpretation* of expressions, and exists here for the evaluation interpretation only for consistency.

Therefore, succeeding the initialization code, the rest of the program consists of the three code parts described above. First, pre-code is inserted per expression node. Then, the main code is inserted for all expression nodes, combined by logical conjunction; and the resulting value is assigned to the dedicated return variable of the first expression node. Finally, the post-code is appended for all expression nodes.

For expression nodes that contain an expression which is not of boolean type, the function $prg_{ExpN,eval}$ provides the empty program:

$$\langle expn_1, \ldots, expn_n \rangle \mapsto \langle\rangle$$

Assignment Code of Expression Nodes. The assignment code that is created for a sequence of expression nodes is defined as function

$$prg_{ExpN,ass} : \operatorname{seq} ExpN_{spec} \to Program$$

Expressions to be interpreted for assignment must be HybEL assignment expressions, therefore

$$\begin{aligned}&\forall\, expn \in \{expn_1, \ldots, expn_n\} \bullet \exists(((t,r), val), sub) \in \operatorname{tree}_{\mathrm{o}} HybelItem \bullet \\ &\quad ht_{expn}(expn) = (((t,r), val), sub) \wedge r \in \{ass, diffAss, intNondetAss, \\ &\qquad assGroup, diffAssGroup, sendSig\}\end{aligned}$$

is required. Then, the function $prg_{ExpN,ass}$ is defined as

$$\begin{aligned}&\langle expn_1, \ldots, expn_n \rangle \mapsto prg_{String}(\\ &\quad initcode_{visparam}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad initcode_{V,cur}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad initcode_{V,prev}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad initcode_{V,curTck}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad initcode_{V,prevTck}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad initcode_{S}(\{expn_1, \ldots, expn_n\}) \frown initcode_{SigParam}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad asspre(ht_{expn}(expn_1), expn_1) \frown \ldots \frown asspre(ht_{expn}(expn_n), expn_n) \frown \\ &\quad assmain(ht_{expn}(expn_1), expn_1) \frown \ldots \frown assmain(ht_{expn}(expn_n), expn_n) \frown \\ &\quad pubcode_{V}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad pubcode_{S,send}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad pubcode_{SigParam}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad pubcode_{S,recv}(\{expn_1, \ldots, expn_n\}) \frown \\ &\quad asspost(ht_{expn}(expn_1), expn_1) \frown \ldots \frown asspost(ht_{expn}(expn_n), expn_n))\end{aligned}$$

The creation of assignment code is somewhat similar to the creation of evaluation code. The generated code starts with the initialization of the visibility set variable from the visibility set parameter, followed by the initialization of local variables from the respective channels. In addition to the current and previous values of HybridUML variables and HybridUML signal states, the time stamps of the current and previous values of HybridUML variables, and the parameter values of HybridUML signals are read and stored locally.

It is followed by the pre-code, which is appended by the main code. The main *assignment* code of an expression node is a program statement, rather than a program expression; therefore, there is no conjunction and assignment, but just a sequence of statements.

Before the post-code is appended, the *publication code* is added: Since the aim of assignment code is to calculate *and publish* new values for channels, this code publishes the calculated values from local variables to the respective channels. Note that the post-code therefore is effective *after* the publication of results.

For expression nodes that contain an expression which is not an assignment, the function $prg_{ExpN,ass}$ provides the empty program:

$$\langle expn_1, \ldots, expn_n\rangle \mapsto \langle\rangle$$

6.5.1 Evaluation Interpretation of Hybel Item Trees

In the context of an expression node, a hybel item tree defines program strings that implement the evaluation of the encoded expression. This is defined by the mapping

$$code : \text{tree}_o\, HybelItem \times \{cur, prev\} \times ExpN \rightarrow String^3$$

The distinction between *cur* and *prev* is made for the contained variables, for which normally the currently available value is effective for calculations, but in specific contexts the previous one. The main code, pre-code and post-code parts are given by the mappings

$$\begin{aligned}
&main : \text{tree}_o\, HybelItem \times \{cur, prev\} \times ExpN \rightarrow String\\
&\quad tvn \mapsto \pi_1\, code(tvn)\\
&pre : \text{tree}_o\, HybelItem \times \{cur, prev\} \times ExpN \rightarrow String\\
&\quad tvn \mapsto \pi_2\, code(tvn)\\
&post : \text{tree}_o\, HybelItem \times \{cur, prev\} \times ExpN \rightarrow String\\
&\quad tvn \mapsto \pi_3\, code(tvn)
\end{aligned}$$

The definition of *code* is then given recursively, by the structure of the hybel item trees.

Literals. Literals of all types are mapped to their string representations:

$$\begin{aligned}
&((((t, lit), val), sub), tm, expn) \mapsto (string(val), "", "")\\
&\quad ;\, t \in \{bool, int, real\} \cup DT_{enum}\\
&((((sdt_{anon}, lit), val), \langle s_1, \ldots, s_n\rangle), tm, expn) \mapsto\\
&\quad ("\{" \frown main(s_1, tm, expn) \frown "," \frown \ldots \frown "," \frown main(s_n, tm, expn) \frown "\}",\\
&\quad pre(s_1, tm, expn) \frown \ldots \frown pre(s_n, tm, expn),\\
&\quad post(s_1, tm, expn) \frown \ldots \frown post(s_n, tm, expn))
\end{aligned}$$

Examples: Almost all examples of section **??** contain literals; e.g. **??**, **??**.

Variables and Signals. The string representation of variables, as well as of received signals (i.e. triggers) is solely given by the contained identifier item tree. For sent signals, there is no interpretation within read-only context.

$$\begin{aligned}
&((((t, var), idtree), sub), tm, expn) \mapsto code_{Id}(idtree, tm, expn)\\
&\quad ;\, t \in DT \cup \{recvSig\}
\end{aligned}$$

Examples: **??**, **??**, **??**, **??**, **??**, **??**.

Derivative Values of Variables. Reading of a derivative is implemented by calculating the last integration step, from the current and previous value and their time stamps of the corresponding analog real variable.

$((((\mathit{anaReal}, \mathit{derivVar}), \mathit{idtree}), \mathit{sub}), \mathit{tm}, \mathit{expn}) \mapsto$

$(\text{"((" } \frown \mathit{main}_{Id}(\mathit{idtree}, \mathit{cur}, \mathit{expn}) \frown \text{"}-\text{"} \frown \mathit{main}_{Id}(\mathit{idtree}, \mathit{prev}, \mathit{expn})$
$\frown \text{")/"} \frown \mathit{string}(\mathit{lvar}_{V,\Delta t}(\mathit{expn}, v)) \frown \text{"["} \frown \mathit{idx}_{main} \frown \text{"])"},$

$\mathit{idx}_{pre} \frown$
$\mathit{string}(\mathit{lvar}_{V,\Delta t}(\mathit{expn}, v)) \frown \text{"[0] :="} \frown$
$\text{"}\mathtt{left}\text{("} \frown \mathit{string}(\mathit{lvar}_{V,curTck}(\mathit{expn}, v)) \frown \text{"[0])"} \frown$
$\text{"}-\mathtt{left}\text{("} \frown \mathit{string}(\mathit{lvar}_{V,prevTck}(\mathit{expn}, v)) \frown \text{"[0]); "}$
$\frown \ldots \frown$ repeat for $1..n$
$\mathit{string}(\mathit{lvar}_{V,\Delta t}(\mathit{expn}, v)) \frown \text{"["} \frown \mathit{string}(n) \frown \text{"] :="} \frown$
$\text{"}\mathtt{left}\text{("} \frown \mathit{string}(\mathit{lvar}_{V,curTck}(\mathit{expn}, v)) \frown \text{"["} \frown \mathit{string}(n) \frown \text{"])"} \frown$
$\text{"}-\mathtt{left}\text{("} \frown \mathit{string}(\mathit{lvar}_{V,prevTck}(\mathit{expn}, v)) \frown \text{"["} \frown \mathit{string}(n) \frown \text{"]); "}$
$\frown$

$\text{"}\mathtt{if}\text{("} \frown \mathit{string}(\mathit{lvar}_{V,\Delta t}(\mathit{expn}, v)) \frown \text{"[0]} \leq \text{0)"} \frown$
$\text{"\{}\mathtt{return}\text{; \}"}$
$\frown \ldots \frown$ repeat for $1..n$
$\text{"}\mathtt{if}\text{("} \frown \mathit{string}(\mathit{lvar}_{V,\Delta t}(\mathit{expn}, v)) \frown \text{"["} \frown \mathit{string}(n) \frown \text{"]} \leq \text{0)"} \frown$
$\text{"\{}\mathtt{return}\text{; \}"},$

$\mathit{idx}_{post})$

with
$\mathit{idtree} = ((t_{id}, (v, \mathit{acc})), \mathit{sub}_{id}) \wedge n = |\mathit{vn}_{expn}(v, \mathit{expn})| - 1 \wedge$
$(\mathit{sub}_{id} = \langle((\mathit{indexExp}, \mathit{exptree}), \langle\rangle)\rangle$
$\quad \Rightarrow (\mathit{idx}_{main}, \mathit{idx}_{pre}, \mathit{idx}_{post}) = \mathit{code}(\mathit{exptree}, \mathit{tm}, \mathit{expn}))$
$\wedge (\mathit{sub}_{id} = \langle\rangle \Rightarrow (\mathit{idx}_{main}, \mathit{idx}_{pre}, \mathit{idx}_{post}) = (\text{"0"}, \text{""}, \text{""}))$

The pre-code prepares the time difference for each of the variable's indices; this is necessary, because the variable's index expression dynamically decides which index is accessed during execution. Further, a check is included to ensure that the time difference is positive, otherwise the calculation will be skipped. Therefore, a division by zero is avoided. The main code just calculates the integration step.

Unary Operations. The code for the negation operation, which is the single available unary operation, is based on the code for the contained expression. Additionally, the negation operator is applied:

$((((\mathit{bool}, \mathit{op}), \neg), \langle s\rangle), \mathit{tm}, \mathit{expn}) \mapsto$

$(\text{"}\neg\text{"} \frown \mathit{main}(s, \mathit{tm}, \mathit{expn}), \mathit{pre}(s, \mathit{tm}, \mathit{expn}), \mathit{post}(s, \mathit{tm}, \mathit{expn}))$

Examples: **??**, **??**, **??**

Binary Operations. Most binary operations directly rely on the contained expressions, i.e. the left-hand side and the right-hand side are transformed to programs separately first, and then combined with the corresponding operator within the program, in a similar fashion as for unary operations:

$$((((t, r), \Diamond), \langle(((t_1, r_1), val_1), sub_1), (((t_2, r_2), val_2), sub_2)\rangle), tm, expn) \mapsto$$

$$\begin{aligned}&\text{"("} \frown main((((t_1, r_1), val_1), sub_1)) \frown \text{"}\Diamond\text{"} \frown main((((t_2, r_2), val_2), sub_2)) \frown \text{")"},\\ &pre((((t_1, r_1), val_1), sub_1)) \frown pre((((t_2, r_2), val_2), sub_2)),\\ &post((((t_1, r_1), val_1), sub_1)) \frown post((((t_2, r_2), val_2), sub_2)))\end{aligned}$$

$$\text{with}\{t_1, t_2\} \cap \{real, anaReal\} = \varnothing \vee \Diamond \notin \{==, \neq\}$$

Examples: **??**, **??**, **??**, **??**, **??**, **??**.

Equality Comparison for Real Values. There are two binary operations that need special treatment – equality comparison and inequality comparison of real values is problematic. Since the execution of the HL3 model is discretized, it is not guaranteed that *zero crossings* of values x_1, x_2 are detected, i.e. that during the discretized run, $x_1 = x_2$ is observed, when their continuous evolutions would cross. Depending on the chosen system period $\delta_{period}(c_{spec})$ and the (implemented) precision of the representation of real values, it is rather unlikely that a naively-implemented (in-)equality comparison would succeed.

Therefore, as an approximation, we compare not only the current values, but also the previous ones, such that a zero crossing is detected, whenever between two succeeding discrete evaluation steps, *no second* zero crossing occurs. Therefore, we actually check the condition

$$x_{1,prev} \leq x_{2,cur} \wedge x_{1,cur} \geq x_{2,prev} \vee x_{1,prev} \geq x_{2,cur} \wedge x_{1,cur} \leq x_{2,prev}$$

Thus, hybel item trees representing (in-)equality comparisons are mapped to programs implementing this check; they are defined by

$$((((t, r), ==), \langle(((t_1, r_1), val_1), sub_1), (((t_2, r_2), val_2), sub_2)\rangle), tm, expn) \mapsto$$

$$\begin{aligned}&\text{"(((("} \frown\\ &main((((t_1, r_1), val_1), sub_1), prev, expn) \frown \text{"}\geq\text{"} \frown\\ &main((((t_2, r_2), val_2), sub_2), cur, expn) \frown \text{")} \wedge \text{("} \frown\\ &main((((t_1, r_1), val_1), sub_1), cur, expn) \frown \text{"}\leq\text{"} \frown\\ &main((((t_2, r_2), val_2), sub_2), prev, expn) \frown \text{"))} \vee \text{(("} \frown\\ &main((((t_1, r_1), val_1), sub_1), prev, expn) \frown \text{"}\leq\text{"} \frown\\ &main((((t_2, r_2), val_2), sub_2), cur, expn) \frown \text{")} \wedge \text{("} \frown\\ &main((((t_1, r_1), val_1), sub_1), cur, expn) \frown \text{"}\geq\text{"} \frown\\ &main((((t_2, r_2), val_2), sub_2), prev, expn) \frown\\ &\text{"))))"},\\ \\ &\text{""}, \text{""})\end{aligned}$$

with $\{t_1, t_2\} \cap \{\mathit{real}, \mathit{anaReal}\} \neq \varnothing$

Inequality comparison programs are based on equality comparison:

$((((t, r), \neq), \langle(((t_1, r_1), \mathit{val}_1), \mathit{sub}_1), (((t_2, r_2), \mathit{val}_2), \mathit{sub}_2)\rangle), \mathit{tm}, \mathit{expn}) \mapsto$
$(\text{"}\neg\text{"} \frown \mathit{main}((((t, r), ==), \langle(((t_1, r_1), \mathit{val}_1), \mathit{sub}_1), (((t_2, r_2), \mathit{val}_2), \mathit{sub}_2)\rangle),$
$\mathit{tm}, \mathit{expn}),$
$\mathit{pre}((((t, r), ==), \langle(((t_1, r_1), \mathit{val}_1), \mathit{sub}_1), (((t_2, r_2), \mathit{val}_2), \mathit{sub}_2)\rangle), \mathit{tm}, \mathit{expn}),$
$\mathit{post}((((t, r), ==), \langle(((t_1, r_1), \mathit{val}_1), \mathit{sub}_1), (((t_2, r_2), \mathit{val}_2), \mathit{sub}_2)\rangle), \mathit{tm}, \mathit{expn}))$

with $\{t_1, t_2\} \cap \{\mathit{real}, \mathit{anaReal}\} \neq \varnothing$

Quantified Boolean Expressions. Quantified boolean expressions are implemented by `while` loops over the specified integer ranges. For each integer range, a separate loop is created. Within the loop, the bound expression is evaluated, and the boolean result is adapted and stored in a dedicated variable. This is encoded in the pre-code part; the main code then only contains the result variable:

$(\mathit{quantTree}, \mathit{tm}, \mathit{expn}) \mapsto$

$(\mathit{string}(\mathit{lvar}_{\mathit{quantRes}}(\mathit{expn}, \mathit{quantTree})),$

$\mathit{asspre}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \mathit{pre}(s_{\mathit{low},1}, \mathit{tm}, \mathit{expn})$
$\frown \mathit{pre}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \mathit{pre}(s_{\mathit{up},1}, \mathit{tm}, \mathit{expn})$
$\frown \mathit{pre}(\mathit{boundExp}, \mathit{tm}, \mathit{expn}) \frown \mathit{asspre}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn})$
$\frown \ldots \frown$ repeat for $1..n$
$\mathit{asspre}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \mathit{pre}(s_{\mathit{low},n}, \mathit{tm}, \mathit{expn})$
$\frown \mathit{pre}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \mathit{pre}(s_{\mathit{up},n}, \mathit{tm}, \mathit{expn})$
$\frown \mathit{pre}(\mathit{boundExp}, \mathit{tm}, \mathit{expn}) \frown \mathit{asspre}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown$

$\mathit{string}(\mathit{lvar}_{\mathit{quantRes}}(\mathit{expn}, \mathit{quantTree})) \frown \text{":="} \frown \mathit{initval} \frown \text{"; "} \frown$

$\mathit{assmain}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \text{":="} \frown \mathit{main}(s_{\mathit{low},1}, \mathit{tm}, \mathit{expn}) \frown \text{"; "} \frown$
$\text{"}\mathtt{while}\text{(("} \frown \mathit{main}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \text{"}\leq\text{"} \frown \mathit{main}(s_{\mathit{up},1}, \mathit{tm}, \mathit{expn})$
$\frown \text{")}\wedge\text{"} \frown \mathit{sign} \frown \mathit{string}(\mathit{lvar}_{\mathit{quantRes}}(\mathit{expn}, \mathit{quantTree})) \frown \text{")\{"} \frown$
$\mathit{string}(\mathit{lvar}_{\mathit{quantRes}}(\mathit{expn}, \mathit{quantTree})) \frown \text{":="} \frown \mathit{main}(\mathit{boundExp}, \mathit{tm}, \mathit{expn})$
$\frown \text{"; "} \frown \text{"++"} \frown \mathit{assmain}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \text{"; "}$
$\frown \ldots \frown$ repeat for $1..n$
$\mathit{assmain}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \text{":="} \frown \mathit{main}(s_{\mathit{low},n}, \mathit{tm}, \mathit{expn}) \frown \text{"; "} \frown$
$\text{"}\mathtt{while}\text{(("} \frown \mathit{main}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \text{"}\leq\text{"} \frown \mathit{main}(s_{\mathit{up},n}, \mathit{tm}, \mathit{expn})$
$\frown \text{")}\wedge\text{"} \frown \mathit{sign} \frown \mathit{string}(\mathit{lvar}_{\mathit{quantRes}}(\mathit{expn}, \mathit{quantTree})) \frown \text{")\{"} \frown$
$\mathit{string}(\mathit{lvar}_{\mathit{quantRes}}(\mathit{expn}, \mathit{quantTree})) \frown \text{":="} \frown \mathit{main}(\mathit{boundExp}, \mathit{tm}, \mathit{expn})$
$\frown \text{"; "} \frown \text{"++"} \frown \mathit{assmain}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \text{"; "},$

$\mathit{asspost}_{\mathit{Id}}(\mathit{idtree}_{\mathit{bnd}}, \mathit{tm}, \mathit{expn}) \frown \mathit{post}(s_{\mathit{low},1}, \mathit{tm}, \mathit{expn})$

$$\frown post_{Id}(idtree_{bnd}, tm, expn) \frown post(s_{up,1}, tm, expn)$$
$$\frown post(boundExp, tm, expn) \frown asspost_{Id}(idtree_{bnd}, tm, expn)$$
$$\frown \ldots \frown \text{ repeat for } 1..n$$
$$asspost_{Id}(idtree_{bnd}, tm, expn) \frown post(s_{low,n}, tm, expn)$$
$$\frown post_{Id}(idtree_{bnd}, tm, expn) \frown post(s_{up,n}, tm, expn)$$
$$\frown post(boundExp, tm, expn) \frown asspost_{Id}(idtree_{bnd}, tm, expn))$$

with

$$quantTree = (((bool, q), idtree_{bnd}), \langle(((intSet, intSpecs), val), \langle(((intSet, intRange), val_1), \langle s_{low,1}, s_{up,1}\rangle), \ldots, (((intSet, intRange), val_n), \langle s_{low,n}, s_{up,n}\rangle)\rangle), boundExp\rangle)$$

$$\wedge\ q \in \{\forall, \exists\}$$
$$\wedge\ q = \forall \Rightarrow inital = \text{"true"} \wedge sign = \text{""}$$
$$\wedge\ q = \exists \Rightarrow inital = \text{"false"} \wedge sign = \text{"}\neg\text{"}$$

Examples: **??**, **??**, **??**, **??**, **??**, **??**.

Assignment Expressions. Assignment expressions that assign values to variables apply the *assignment interpretation* for the left-hand side, which represents the variable to be assigned. The right-hand side is an expression that is interpreted as a "conventional" expression, i.e. the read interpretation is used, recursively. This rule excludes differential assignments that assign a derivative:

$$((((t, r), val), \langle s_1, \ldots, s_n\rangle), tm, expn) \mapsto$$

$$(assmain(s_1, expn) \frown \text{":="} \frown main(s_2, tm, expn),$$
$$asspre(s_1, expn) \frown pre(s_2, tm, expn),$$
$$asspost(s_1, expn) \frown post(s_2, tm, expn))$$

with $s_1 = ((role_1, val_1), sub_1) \wedge role_1 \neq (anaReal, derivVar)$
$\wedge\ r \in \{ass, diffAss\}$

Examples: **??**, **??**, **??**, **??**.

Assignment of Derivative Values of Variables. The assignment of the derivative of an analog real variable v of the form $\dot{v} = val$ is interpreted such that for the next integration step, the calculated value from the right-hand side is the constant slope for this integration step. That is, for the approximation of the derivative $\dot{v}$

$$\begin{aligned}
\dot{v}(t) &= \lim_{\Delta t \to 0} \frac{v(t + \Delta t) - v(t)}{\Delta t} \\
&\approx \frac{v(t + \Delta t) - v(t)}{\Delta t} \\
&= \frac{v(t_{new}) - v(t)}{t_{new} - t}
\end{aligned}$$

a new value for v results:

$$\begin{aligned} v(t_{new}) &= val \cdot (t_{new} - t) + v(t) \\ &\approx \dot{v}(t) \cdot (t_{new} - t) + v(t) \end{aligned}$$

Since there are possibly different publication times t_{new}, given by the visibility set parameter of the flow that hosts the assignment, this calculation has to be done for each visibility entry within that set. Therefore, a `while` loop is generated that iterates the visibility entries and repeats the calculation of the integration step, correspondingly.

$$\begin{aligned}
&((((anaReal, diffAss), val), \\
&\quad \langle(((anaReal, derivVar), idtree_1), sub_1), s_2, \ldots, s_n\rangle), tm, expn) \mapsto \\[1ex]
&(assmain((((anaReal, derivVar), idtree_1), sub_1), expn) \frown \text{"}{:=}\ (\text{"} \\
&\frown main(s_2, tm, expn) \frown \text{"}) \cdot (\texttt{left}(\text{"} \frown string(lvar_{newTck}(expn)) \\
&\frown \text{"}) - \texttt{left}(\text{"} \frown string(lvar_{V,curTck}(expn, v)) \frown \text{"})+\text{"} \\
&\frown main((((anaReal, derivVar), idtree_1), sub_1), tm, expn), \\[1ex]
&pre((((anaReal, derivVar), idtree_1), sub_1), tm, expn) \\
&\frown asspre((((anaReal, derivVar), idtree_1), sub_1), expn) \\
&\frown pre(s_2, tm, expn) \frown \\[1ex]
&string(lvar_{visOrig}(expn)) \frown \text{"}{:=}\ \texttt{getVisParam}();\text{"} \frown \\
&string(lvar_{visIdx}(expn)) \frown \text{"}{:=}\ \texttt{size}(\text{"} \frown string(lvar_{visSet}(expn)) \frown \text{"});\text{"} \\
&\frown \text{"}\texttt{while}(\text{"} \frown string(lvar_{visIdx}(expn)) \frown \text{"}> 0)\{\text{"} \frown \\
&\text{"}\texttt{clear}(\text{"} \frown string(lvar_{visSet}(expn)) \frown \text{"});\text{"} \frown \\
&string(lvar_{vis}(expn)) \frown \text{"}{:=}\ \texttt{getEntry}(\text{"} \frown string(lvar_{visOrig}(expn)) \\
&\frown \text{","} \frown string(lvar_{visIdx}(expn)) \frown \text{"});\text{"} \frown \\
&\text{"}\texttt{addEntry}(\text{"} \frown string(lvar_{visSet}(expn)) \\
&\frown \text{","} \frown string(lvar_{vis}(expn)) \frown \text{"});\text{"} \frown \\
&string(lvar_{newTck}(expn)) \frown \text{"}{:=}\ \texttt{left}(\text{"} \frown string(lvar_{vis}(expn)) \frown \text{"});\text{"}, \\[1ex]
&post((((anaReal, derivVar), idtree_1), sub_1), tm, expn) \\
&\frown asspost((((anaReal, derivVar), idtree_1), sub_1), expn) \\
&\frown post(s_2, tm, expn) \\
&\frown \text{"}--\text{"} \frown string(lvar_{visIdx}(expn)) \frown \text{"};\}\text{"})
\end{aligned}$$

with $idtree_1 = ((t, (v, acc)), sub_{id})$

Note that the main code is the calculation of one single integration step, whereas the pre-code and post-code model the surrounding `while` loop: The pre-code prepares the iteration of the visibility entries given by the visibility set parameter and defines the condition of the loop. The post-code increments the counter for the visibility entries and closes the loop. Here it is assumed that between main code and post-code some send code is inserted that actually publishes the calculated value for the current iteration; this is defined above, by $prg_{ExpN,ass}$.

Examples: **??**.

Default Rule. For all other hybel item trees that are not mapped above, an empty program is created:

$$(tree, tm, expn) \mapsto (\text{""}, \text{""}, \text{""});\ \text{ else}$$

6.5.2 Evaluation Interpretation of Identifier Item Trees

For identifiers (including index expressions), the evaluation interpretation maps identifier item trees to program strings. Similarly to hybel item trees, this also depends on the expression node for which the program is created, and on the flag that either requests the current or the previous value for the identifier.

$$code_{Id} : \text{tree}_{\text{o}}\, IdItem \times \{cur, prev\} \times ExpN \to String^3$$

There are main code, pre-code, and post-code projections, given by the mappings

$$\begin{aligned}
&main_{Id} : \text{tree}_{\text{o}}\, IdItem \times \{cur, prev\} \times ExpN \to String\\
&\quad tvn \mapsto \pi_1\, code_{Id}(tvn)\\
&pre_{Id} : \text{tree}_{\text{o}}\, IdItem \times \{cur, prev\} \times ExpN \to String\\
&\quad tvn \mapsto \pi_2\, code_{Id}(tvn)\\
&post_{Id} : \text{tree}_{\text{o}}\, IdItem \times \{cur, prev\} \times ExpN \to String\\
&\quad tvn \mapsto \pi_3\, code_{Id}(tvn)
\end{aligned}$$

Variable Identifiers. A variable identifier with index expression is mapped to the local variable that represents *read* access to the HybridUML variable or signal. For every identifier, the local variable is of array type, and the array index results from the index expression:

$$\begin{aligned}
&(((t, (v, acc)), \langle((indexExp, exptree), \langle\rangle)\rangle), tm, expn) \mapsto\\
&(string(lvar_{V, V_{local}}(expn, v, read, tm))\\
&\frown \text{"["} \frown main_{Id}(((indexExp, exptree), \langle\rangle), tm, expn) \frown \text{"]"},\\
&pre_{Id}(((indexExp, exptree), \langle\rangle), tm, expn),\\
&post_{Id}(((indexExp, exptree), \langle\rangle), tm, expn))\\
&\text{with } t \in DT
\end{aligned}$$

Variable identifiers without index expression are treated similarly, but array index 0 is used. Therefore, variable expressions x and $x[0]$ have the same meaning.

$$\begin{aligned}
&(((t, (v, acc)), \langle\rangle), tm, expn) \mapsto\\
&(string(lvar_{V, V_{local}}(expn, v, read, tm)) \frown \text{"[0]"}, \text{""}, \text{""})\\
&\text{with } t \in DT
\end{aligned}$$

Examples: **??**, **??**, **??**, **??**, **??**, **??**.

Sub-Variables of Structured Data Types. Access to a sub-variable of a variable of structured data type is defined recursively:

$$\begin{aligned}
&(((t,(v,acc)),\langle s_1,s_2\rangle),tm,expn)\mapsto\\
&(string(lvar_{V,V_{local}}(expn,v,read,tm))\frown\text{"[}"\frown main_{Id}(s_1,tm,expn)\frown\text{"].}"\\
&\frown main_{Id}(s_2,tm,expn),\\
&pre_{Id}(s_1,tm,expn)\frown pre_{Id}(s_2,tm,expn),\\
&post_{Id}(s_1,tm,expn)\frown post_{Id}(s_2,tm,expn))\\
&\text{with } t\in DT_{struc}
\end{aligned}$$

If there is no index expression, index 0 is applied:

$$\begin{aligned}
&(((t,(v,acc)),\langle((t_s,val_s),sub_s)\rangle),tm,expn)\mapsto\\
&(string(lvar_{V,V_{local}}(expn,v,read,tm))\frown\text{"[0]."}\\
&\frown main_{Id}(((t_s,val_s),sub_s),tm,expn),\\
&pre_{Id}(((t_s,val_s),sub_s),tm,expn),\\
&post_{Id}(((t_s,val_s),sub_s),tm,expn))\\
&\text{with } t\in DT_{struc}\wedge t_s\neq indexExp
\end{aligned}$$

Examples: **??**, **??**, **??**, **??**, **??**, **??**.

Index Expressions of Triggers. Triggers can have index assignment subexpressions, which require special treatment. In contrast, triggers with conventional index expressions are mapped in the same fashion as variables are:

$$\begin{aligned}
&(((recvSig,(s,recv)),\langle((indexExp,(((int,r),val),sub)),\langle\rangle)\rangle),tm,expn)\mapsto\\
&(string(lvar_S(expn,s,read))\\
&\frown\text{"[}"\frown main_{Id}(((indexExp,(((int,r),val),sub)),\langle\rangle),tm,expn)\frown\text{"]"},\\
&pre_{Id}(((indexExp,(((int,r),val),sub)),\langle\rangle),tm,expn),\\
&post_{Id}(((indexExp,(((int,r),val),sub)),\langle\rangle),tm,expn))\\
&\text{with } r\neq indexAss
\end{aligned}$$

If there is no index expression, index 0 is applied:

$$\begin{aligned}
&(((recvSig,(s,recv)),\langle\rangle),tm,expn)\mapsto\\
&(string(lvar_S(expn,s,read))\frown\text{"[0]"},\text{""},\text{""})
\end{aligned}$$

For triggers with index assignment subexpression, a program is created that accepts signal occurences *for all* indices. The succeeding evaluation of the actually affected index is defined by the *assignment interpretation* of triggers, which is introduced later.

$$\begin{aligned}
&(((recvSig,(s,recv)),\\
&\quad\langle((indexExp,(((int,indexAss),val),sub)),\langle\rangle)\rangle),tm,expn)\mapsto\\
&(\text{"("}\frown string(lvar_S(expn,s,read))\frown\text{"[0]"},
\end{aligned}$$

$\frown ''\lor'' \frown \ldots \frown ''\lor'' \frown$ repeat for $1..n$
$string(lvars_S(expn, s, read)) \frown ''['' string(n)'']) '',$
$'''', '''')$
with $n = |sn_{expn}(s, expn)| - 1$

Conventional Index Expressions. Conventional index expressions of identifier items are given by hybel item trees:

$$(((indexExp, exptree), sub), tm, expn) \mapsto code(exptree, tm, expn)$$

Examples: **??**, **??**, **??**, **??**, **??**, **??**.

Default Rule. All other identifier item trees are mapped to the empty program:

$$(tree, tm, expn) \mapsto ('''', '''', ''''); \text{ else}$$

6.5.3 Assignment Interpretation of Hybel Item Trees

Some hybel item trees have different interpretations either for evaluation of their values or in the context of assignment. The latter is defined as a mapping

$$asscode : \text{tree}_o\, HybelItem \times ExpN \to String^3$$

As for *code*, there are main code, pre-code and post-code mappings for *asscode*:

$$\begin{aligned} &assmain : \text{tree}_o\, HybelItem \times ExpN \to String \\ &\quad tn \mapsto \pi_1\, asscode(tn) \\ &asspre : \text{tree}_o\, HybelItem \times ExpN \to String \\ &\quad tn \mapsto \pi_2\, asscode(tn) \\ &asspost : \text{tree}_o\, HybelItem \times ExpN \to String \\ &\quad tn \mapsto \pi_3\, asscode(tn) \end{aligned}$$

Variables. The assignment interpretation of any variable is determined by the assignment interpretation of its identifier item tree:

$$((((t, r), idtree), sub), expn) \mapsto asscode_{Id}(idtree, expn)$$
with $r \in \{var, derivVar\}$

Examples: **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**.

Sending of Signals. Signal raise statements are composed of (1) the sending of the signal itself, which is defined by the assignment interpretation of the identifier item tree, and (2) the writing of the actual parameter values. The latter is defined here. A dedicated index variable is used to choose the signal's index wrt. to its multiplicity for sending. It is assumed that this variable holds the appropriate value, prepared in the context of the raise statement.

$$((((\mathit{sigtype}, \mathit{sendSig}), \mathit{idtree}), \langle s_1, \ldots, s_n\rangle), \mathit{expn}) \mapsto$$

$(\mathit{assmain}_{Id}(\mathit{idtree}, \mathit{expn})^\frown$
$\mathit{string}(\mathit{lvar}_{SigParam}(\mathit{expn}, s, \mathit{write}, 1)) ^\frown "["$
$^\frown\ \mathit{string}(\mathit{lvar}_{sigIdx}(\mathit{expn}, s)) ^\frown "]" ^\frown ":=" ^\frown \mathit{main}(s_1, \mathit{cur}, \mathit{expn}) ^\frown ";\ "$
$^\frown \ldots ^\frown$ repeat for $1..n$
$\mathit{string}(\mathit{lvar}_{SigParam}(\mathit{expn}, s, \mathit{write}, n)) ^\frown "["$
$^\frown\ \mathit{string}(\mathit{lvar}_{sigIdx}(\mathit{expn}, s)) ^\frown "]" ^\frown ":=" ^\frown \mathit{main}(s_n, \mathit{cur}, \mathit{expn}) ^\frown ";\ ",$

$\mathit{asspre}_{Id}(\mathit{idtree}, \mathit{expn}) ^\frown \mathit{pre}(s_1, \mathit{cur}, \mathit{expn}) ^\frown \ldots ^\frown \mathit{pre}(s_n, \mathit{cur}, \mathit{expn}),$

$\mathit{asspost}_{Id}(\mathit{idtree}, \mathit{expn}) ^\frown \mathit{post}(s_1, \mathit{cur}, \mathit{expn}) ^\frown \ldots ^\frown \mathit{post}(s_1, \mathit{cur}, \mathit{expn}))$

with $\mathit{idtree} = ((\mathit{sendSig}, (s, \mathit{acc})), \mathit{sub}_{id})$

Receiving of Signals. The receiving of signals (defined by trigger expressions) leads to the assignment of local variables from the parameter values that are associated with the signal's occurrence. The handling of the signal itself is defined later by a rule for the included identifier item tree.

$$((((\mathit{sigtype}, \mathit{recvSig}), \mathit{idtree}), \langle s_1, \ldots, s_n\rangle), \mathit{expn}) \mapsto$$

$(\mathit{assmain}_{Id}(\mathit{idtree}, \mathit{expn})^\frown$
$\mathit{assmain}_{Id}(s_1, \mathit{expn}) ^\frown ":=" ^\frown \mathit{string}(\mathit{lvar}_{SigParam}(\mathit{expn}, s, \mathit{read}, 1))$
$^\frown "[" ^\frown \mathit{string}(\mathit{lvar}_{trgIdx}(\mathit{expn}, s)) ^\frown "];\ "$
$^\frown \ldots ^\frown$ repeat for $1..n$
$\mathit{assmain}_{Id}(s_n, \mathit{expn}) ^\frown ":=" ^\frown \mathit{string}(\mathit{lvar}_{SigParam}(\mathit{expn}, s, \mathit{read}, n))$
$^\frown "[" ^\frown \mathit{string}(\mathit{lvar}_{trgIdx}(\mathit{expn}, s)) ^\frown "];\ ",$

$\mathit{asspre}_{Id}(\mathit{idtree}, \mathit{expn}) ^\frown \mathit{asspre}_{Id}(s_1, \mathit{expn}) ^\frown \ldots ^\frown \mathit{asspre}_{Id}(s_n, \mathit{expn}),$

$\mathit{asspost}_{Id}(\mathit{idtree}, \mathit{expn}) ^\frown \mathit{asspost}_{Id}(s_1, \mathit{expn}) ^\frown \ldots ^\frown \mathit{asspost}_{Id}(s_n, \mathit{expn}))$

with $\mathit{idtree} = ((\mathit{recvSig}, (s, \mathit{acc})), \mathit{sub}_{id})$

Assignment Expressions. The assignment interpretation of simple assignments is almost identical with the expression interpretation; the only difference is that the resulting main code forms a program *statement*, rather than a program *expression*:

$$((((t, r), \mathit{val}), \mathit{sub}), \mathit{expn}) \mapsto$$

$\mathit{main}((((t, r), \mathit{val}), \mathit{sub}), \mathit{cur}, \mathit{expn}) ^\frown ";\ ",$
$\mathit{pre}((((t, r), \mathit{val}), \mathit{sub}), \mathit{cur}, \mathit{expn}),$
$\mathit{post}((((t, r), \mathit{val}), \mathit{sub}), \mathit{cur}, \mathit{expn}))$

with $r \in \{ass, diffAss\}$

Examples: **??**, **??**, **??**, **??**.

Assignment Group Expressions. A group of assignments is transformed into a set of loops that iterate all specified values for the bound integer variable and always execute the bound assignment with the respective value:

$$\begin{aligned}&((((t, r), idtree_{bnd}), \langle(((intSet, intSpecs), val),\\ &\quad \langle(((intSet, intRange), val_1), \langle s_{low,1}, s_{up,1}\rangle), \ldots,\\ &\quad (((intSet, intRange), val_n), \langle s_{low,n}, s_{up,n}\rangle)\rangle), boundExp\rangle), expn) \mapsto\end{aligned}$$

$$\begin{aligned}&string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown \text{"[0] :="} \frown main(s_{low,1}, cur, expn) \frown \text{"; "}\\ &\frown \text{"}\texttt{while}\text{(("} \frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown \text{"[0]} \le \text{"}\\ &\frown main(s_{up,1}, cur, expn) \frown \text{") \{"}\\ &\frown assmain(boundExp, expn) \frown\\ &\text{"++"} \frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown \text{"[0]; \}"}\\ &\frown \ldots \frown \text{ repeat for } 1..n\\ &string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown \text{"[0] :="} \frown main(s_{low,n}, cur, expn) \frown \text{"; "}\\ &\frown \text{"}\texttt{while}\text{(("} \frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown \text{"[0]} \le \text{"}\\ &\frown main(s_{up,n}, cur, expn) \frown \text{") \{"}\\ &\frown assmain(boundExp, expn) \frown\\ &\text{"++"} \frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown \text{"[0]; \}"},\end{aligned}$$

$$\begin{aligned}&pre(s_{low,1}, cur, expn) \frown pre(s_{up,1}, cur, expn) \frown asspre(boundExp, expn)\\ &\frown \ldots \frown \text{ repeat for } 1..n\\ &pre(s_{low,n}, cur, expn) \frown pre(s_{up,n}, cur, expn) \frown asspre(boundExp, expn),\end{aligned}$$

$$\begin{aligned}&post(s_{low,1}, cur, expn) \frown post(s_{up,1}, cur, expn) \frown asspost(boundExp, expn)\\ &\frown \ldots \frown \text{ repeat for } 1..n\\ &post(s_{low,n}, cur, expn) \frown post(s_{up,n}, cur, expn) \frown asspost(boundExp, expn))\end{aligned}$$

with $idtree_{bnd} = ((int, (v_{bnd}, acc)), sub_{id}) \land r \in \{assGroup, diffAssGroup\}$

Examples: **??**, **??**, **??**, **??**.

Non-Deterministic Integer Assignment Expressions. A non-deterministic integer assignment is mapped to a program that iterates all given specified integer values, implemented by a set of loops. For each value, the bound boolean expression is evaluated, and the corresponding integer value is stored, if satisfied. For this, an array variable that stores the satisfied integers (given by $lvar_{hits}$) and a size counter (from mapping $lvar_{hitCount}$) are used. Finally, from the integers with satisfied expression evaluation, a random one is chosen.

$(assTree, expn) \mapsto$

$(string(lvar_{hitCount}(expn, assTree)) \frown$ ":= 0; " $\frown$
$string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown$ "[0] :=" $\frown main(s_{low,1}, cur, expn) \frown$ "; "
$\frown$ "`while`((" $\frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown$ "[0] $\leq$"
$\frown main(s_{up,1}, cur, expn) \frown$ ") $\wedge$ (" $\frown$
$string(lvar_{hitCount}(expn, assTree) \frown$ "<" $\frown string(maxHitCount) \frown$ ")) {" $\frown$
"`if`(" $\frown main(boundExp, cur, expn) \frown$ ") {" $\frown$
$string(lvar_{hits}(expn, assTree))$
$\frown$ "[" $\frown string(lvar_{hitCount}(expn, assTree)) \frown$ "] :="
$\frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown$ "[0]; " $\frown$
"++" $\frown string(lvar_{hitCount}(expn, assTree)) \frown$ "; " $\frown$
"}" $\frown$
"++" $\frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown$ "[0]; " $\frown$
"}" $\frown$
$\frown \ldots \frown$ repeat for $1..n$
$string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown$ "[0] :=" $\frown main(s_{low,n}, cur, expn) \frown$ "; "
$\frown$ "`while`((" $\frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown$ "[0] $\leq$"
$\frown main(s_{up,n}, cur, expn) \frown$ ") $\wedge$ (" $\frown$
$string(lvar_{hitCount}(expn, assTree) \frown$ "<" $\frown string(maxHitCount) \frown$ ")) {" $\frown$
"`if`(" $\frown main(boundExp, cur, expn) \frown$ ") {" $\frown$
$string(lvar_{hits}(expn, assTree))$
$\frown$ "[" $\frown string(lvar_{hitCount}(expn, assTree)) \frown$ "] :="
$\frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown$ "[0]; " $\frown$
"++" $\frown string(lvar_{hitCount}(expn, assTree)) \frown$ "; " $\frown$
"}" $\frown$
"++" $\frown string(lvar_{V,V_{local}}(expn, v_{bnd})) \frown$ "[0]; " $\frown$
"}" $\frown$
"`if`(" $\frown string(lvar_{hitCount}(expn, assTree)) \frown$ "> 0) {" $\frown$
$assmain(varExp, expn) \frown$ ":=" $\frown string(lvar_{hits}(expn, assTree)) \frown$ "[" $\frown$
"`random`() mod" $\frown string(lvar_{hitCount}(expn, assTree)) \frown$ "; "
"}",

$pre(s_{low,1}, cur, expn) \frown pre(s_{up,1}, cur, expn) \frown pre(boundExp, cur, expn)$
$\frown \ldots \frown$ repeat for $1..n$
$pre(s_{low,n}, cur, expn) \frown pre(s_{up,n}, cur, expn) \frown pre(boundExp, cur, expn)$
$\frown asspre(varExp, expn)$

$post(s_{low,1}, cur, expn) \frown post(s_{up,1}, cur, expn) \frown post(boundExp, cur, expn)$
$\frown \ldots \frown$ repeat for $1..n$
$post(s_{low,n}, cur, expn) \frown post(s_{up,n}, cur, expn) \frown post(boundExp, cur, expn)$
$\frown asspost(varExp, expn)$

with $assTree = (((int, intNondetAss), ((int, (v_{bnd}, acc)), sub_{id})),$
$\langle varExp, (((intSet, intSpecs), val), \langle(((intSet, intRange), val_1),$
$\langle s_{low,1}, s_{up,1}\rangle), \ldots,$
$(((intSet, intRange), val_n), \langle s_{low,n}, s_{up,n}\rangle)\rangle), boundExp\rangle)$

Note that there is a defined maximum number *maxHitCount* of integers that can be stored intermediately, such that the non-determinism is restricted to the first *maxHitCount* integers for which the bound expression is satisfied.

Examples: **??**.

Default Rule. For all hybel item trees that have no assignment interpretation, the empty program is created:

$(tree, expn) \mapsto ("", "", "");$ else

6.5.4 Assignment Interpretation of Identifier Item Trees

Corresponding to hybel item trees, identifier item trees have an assignment interpretation. There is a mapping to program strings, which also depends on the expression node for which the program is created.

$asscode_{Id} : \text{tree}_{\text{o}}\, IdItem \times ExpN \to String^3$

There are main code, pre-code and post-code projections, given by the mappings

$assmain_{Id} : \text{tree}_{\text{o}}\, IdItem \times ExpN \to String$
$tn \mapsto \pi_1\, asscode_{Id}(tn)$
$asspre_{Id} : \text{tree}_{\text{o}}\, IdItem \times ExpN \to String$
$tn \mapsto \pi_2\, asscode_{Id}(tn)$
$asspost_{Id} : \text{tree}_{\text{o}}\, IdItem \times ExpN \to String$
$tn \mapsto \pi_3\, asscode_{Id}(tn)$

Variable Identifiers. Variable identifiers are mapped to the local array variable that represents *write* access to the HybridUML variable or signal. The array index results from the index expression:

$(((t, (v, acc)), \langle((indexExp, exptree), \langle\rangle)\rangle), expn) \mapsto$
$(string(lvar_{V, V_{local}}(expn, v, write, cur))$
$\frown "[" \frown main_{Id}(((indexExp, exptree), \langle\rangle), cur, expn) \frown "]",$
$pre_{Id}(((indexExp, exptree), \langle\rangle), cur, expn),$
$post_{Id}(((indexExp, exptree), \langle\rangle), cur, expn))$
with $t \in DT$

Variable identifiers without index expression are treated similarly, array index 0 is applied.

$(((t, (v, acc)), \langle\rangle), expn) \mapsto$
$(string(lvar_{V, V_{local}}(expn, v, write, cur)) \frown "\mathsf{[0]}", "", "")$
with $t \in DT$

Examples: **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**.

Sub-Variables of Structured Data Types. Access to a sub-variable of a variable of structured data type is defined recursively, in the same fashion as for $code_{Id}$.

$$
\begin{aligned}
&(((t,(v,acc)),\langle s_1,s_2\rangle),expn)\mapsto\\
&(string(lvar_{V,V_{local}}(expn,v,write,cur))\frown\text{"[}"\frown main_{Id}(s_1,cur,expn)\frown\text{"].''}\\
&\frown assmain_{Id}(s_2,expn),\\
&pre_{Id}(s_1,cur,expn)\frown asspre_{Id}(s_2,expn),\\
&post_{Id}(s_1,cur,expn)\frown asspost_{Id}(s_2,expn))\\
&\text{with } t\in DT_{struc}
\end{aligned}
$$

Without index expression, index 0 is applied:

$$
\begin{aligned}
&(((t,(v,acc)),\langle(((t_s,val_s),sub_s))\rangle),expn)\mapsto\\
&(string(lvar_{V,V_{local}}(expn,v,write,cur))\frown\text{"[0].''}\\
&\frown assmain_{Id}(((t_s,val_s),sub_s),expn),\\
&asspre_{Id}(((t_s,val_s),sub_s),expn),\\
&asspost_{Id}(((t_s,val_s),sub_s),expn))\\
&\text{with } t\in DT_{struc}\wedge t_s\neq indexExp
\end{aligned}
$$

Signal Identifiers on Signal Reception. The assignment interpretation of triggers is the consumption of the signal's occurrence. This is done by publishing the value *false* for the signal. Here, the value is written to the corresponding local variable. For a conventional index expression, the affected trigger index is given by the index expression; a dedicated trigger index variable is set to this index as a side-effect. This is supposed to be used by special code for the publication of the value, which is defined in section 6.5.7.

$$
\begin{aligned}
&(((recvSig,(s,recv)),\langle(((indexExp,(((int,r),val_1),sub_1)),\langle\rangle)\rangle),expn)\mapsto\\
&(string(lvar_{trgIdx}(expn,s))\frown\text{":=''}\\
&\qquad\frown main((((int,r),val_1),sub_1),cur,expn)\frown\text{"; ''}\frown\\
&(string(lvar_S(expn,s,write))\\
&\qquad\frown\text{"[''}\frown string(lvar_{trgIdx}(expn,s))\frown\text{"] := false; '',}\\
&pre((((int,r),val_1),sub_1),cur,expn),\\
&post((((int,r),val_1),sub_1),cur,expn))\\
&\text{with } r\neq indexAss
\end{aligned}
$$

Without index expression, index 0 is used:

$$
\begin{aligned}
&(((recvSig,(s,recv)),\langle\rangle),expn)\mapsto\\
&string(lvar_{trgIdx}(expn,s))\frown\text{":= 0; ''}\frown\\
&(string(lvar_S(expn,s,write))\\
&\qquad\frown\text{"[''}\frown string(lvar_{trgIdx}(expn,s))\frown\text{"] := false; '',}\\
&\text{"'', ""})
\end{aligned}
$$

On reception of a signal with index assignment expression, all indices of the signal have to be iterated, in order find an affected one. The first index for which the signal is active is taken, such that for several indices, the smallest wins:

$$(((\mathit{recvSig}, (s, \mathit{recv})),$$
$$\langle(((\mathit{indexExp}, (((\mathit{int}, \mathit{indexAss}), \mathit{val}_1), \langle s_{ass}\rangle)), \langle\rangle)\rangle), \mathit{expn}) \mapsto$$

$$(\mathit{string}(\mathit{lvar}_{\mathit{trgIdx}}(\mathit{expn}, s)) \frown \text{" := 0; "} \frown$$
$$\text{"}\mathtt{while}\text{ (("} \frown \mathit{string}(\mathit{lvar}_{\mathit{trgIdx}}(\mathit{expn}, s)) \frown \text{"<"} \frown \mathit{string}(n)$$
$$\frown \text{")} \wedge (\neg\text{"} \frown \mathit{string}(\mathit{lvar}_S(\mathit{expn}, s, \mathit{read}))$$
$$\frown \text{"["} \frown \mathit{string}(\mathit{lvar}_{\mathit{trgIdx}}(\mathit{expn}, s)) \frown \text{"])) \{"} \frown$$
$$\text{"++"} \frown \mathit{string}(\mathit{lvar}_{\mathit{trgIdx}}(\mathit{expn}, s)) \frown \text{"; "} \frown$$
$$\text{"\}"} \frown$$
$$\mathit{assmain}(s_{ass}, \mathit{expn}) \frown \text{":="} \frown \mathit{string}(\mathit{lvar}_{\mathit{trgIdx}}(\mathit{expn}, s)) \frown \text{"; "} \frown$$
$$\mathit{string}(\mathit{lvar}_S(\mathit{expn}, s, \mathit{write})) \frown \text{"["} \frown \mathit{string}(\mathit{lvar}_{\mathit{trgIdx}}(\mathit{expn}, s)) \frown \text{"]"}$$
$$\frown \text{":= false; "},$$

$$\mathit{asspre}(s_{ass}, \mathit{expn}),$$

$$\mathit{asspost}(s_{ass}, \mathit{expn}))$$

with $n = |sn_{expn}(s, expn)| - 1$

Signal Identifiers on Signal Sending. Sending of signals is realized by programs that write the value *true* to the corresponding local variable, such that this will be published to the signal's channel. A dedicated index variable holds the index value for this reason.

$$(((\mathit{sendSig}, (s, \mathit{send})), \langle s_{idx}\rangle), \mathit{expn}) \mapsto$$

$$(\mathit{string}(\mathit{lvar}_{\mathit{sigIdx}}(\mathit{expn}, s)) \frown \text{":="} \frown \mathit{main}_{Id}(s_{idx}, \mathit{cur}, \mathit{expn}) \frown \text{"; "} \frown$$
$$\mathit{string}(\mathit{lvar}_S(\mathit{expn}, s, \mathit{write})) \frown \text{"["} \frown \mathit{string}(\mathit{lvar}_{\mathit{sigIdx}}(\mathit{expn}, s)) \frown \text{"]"}$$
$$\frown \text{":= true; "},$$

$$\mathit{pre}_{Id}(s_{idx}, \mathit{cur}, \mathit{expn}),$$

$$\mathit{pre}_{Id}(s_{idx}, \mathit{cur}, \mathit{expn}))$$

Without an index expression, index 0 is used.

$$(((\mathit{sendSig}, (s, \mathit{send})), \langle\rangle), \mathit{expn}) \mapsto$$

$$(\mathit{string}(\mathit{lvar}_{\mathit{sigIdx}}(\mathit{expn}, s)) \frown \text{":= 0; "} \frown$$
$$\mathit{string}(\mathit{lvar}_S(\mathit{expn}, s, \mathit{write})) \frown \text{"["} \frown \mathit{string}(\mathit{lvar}_{\mathit{sigIdx}}(\mathit{expn}, s)) \frown \text{"]"}$$
$$\frown \text{":= true; "},$$

$$\text{"","")}$$

Conventional Index Expressions. Conventional index expressions of identifier items are given by hybel item trees:

$$(((\mathit{indexExp}, \mathit{exptree}), \mathit{sub}), \mathit{expn}) \mapsto \mathit{asscode}(\mathit{exptree}, \mathit{expn})$$

Examples: **??**, **??**, **??**, **??**, **??**.

Default Rule. The assignment interpretation of all remaining identifier item trees is the empty program:

$$(\mathit{tree}, \mathit{expn}) \mapsto ("","",""); \text{ else}$$

6.5.5 Initialization of Local Variables from Channels

Before any program code is executed that implements the evaluation or assignment interpretation of expression nodes, dedicated local variables must be initialized. That are the local variables which provide read access to values that are read from channels. All other local variables are initialized inside the respective program code.

Therefore, the mapping $\mathit{initcode} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String}^6$ provides the initialization code, such that the projections define the initialization of the following local variables:

1. The initialization code for local variables reading the current value of a HybridUML variable from a channel is given by

$$\begin{aligned} &\mathit{initcode}_{V,cur} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String} \\ &\quad \mathit{expn} \mapsto \pi_1\, \mathit{initcode}(\mathit{expn}) \end{aligned}$$

2. Local variables reading the previous value of a HybridUML variable are initialized by the code

$$\begin{aligned} &\mathit{initcode}_{V,prev} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String} \\ &\quad \mathit{expn} \mapsto \pi_2\, \mathit{initcode}(\mathit{expn}) \end{aligned}$$

3. The time stamp for the current value of a HybridUML variable is read and assigned by the code

$$\begin{aligned} &\mathit{initcode}_{V,curTck} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String} \\ &\quad \mathit{expn} \mapsto \pi_3\, \mathit{initcode}(\mathit{expn}) \end{aligned}$$

4. The previous time stamp is initialized with

$$\begin{aligned} &\mathit{initcode}_{V,prevTck} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String} \\ &\quad \mathit{expn} \mapsto \pi_4\, \mathit{initcode}(\mathit{expn}) \end{aligned}$$

5. Initializing variables holding the enabled-flags of HybridUML signals is defined as

$$\begin{aligned} &\mathit{initcode}_{S} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String} \\ &\quad \mathit{expn} \mapsto \pi_5\, \mathit{initcode}(\mathit{expn}) \end{aligned}$$

6. The initialization of signal parameters is given by

$$\begin{aligned} &\mathit{initcode}_{SigParam} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String} \\ &\quad \mathit{expn} \mapsto \pi_6\ \mathit{initcode}(\mathit{expn}) \end{aligned}$$

The initialization code assignment is then defined by a collection of several mappings. Sets of expression nodes are mapped to initialization code for the hybel trees that represent their expressions:

$$\begin{aligned} &\mathit{initcode} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String}^6 \\ &\mathit{expnset} \mapsto \mathit{initcode}_1(\{(\mathit{var}_{ht,read}(\mathit{ht}_{expn}(\mathit{expn})), \mathit{expn}) \mid \\ &\qquad \mathit{expn} \in \mathit{expnset} \cap \mathit{ExpN}_{spec}\} \\ &\qquad \cup \{(\mathit{sig}_{ht,read}(\mathit{ht}_{expn}(\mathit{expn})), \mathit{expn}) \mid \mathit{expn} \in \mathit{expnset} \cap \mathit{ExpN}_{spec}\}) \end{aligned}$$

This is defined by the initialization of sets of variables and signals, within expression nodes:

$$\begin{aligned} &\mathit{initcode}_1 : \mathcal{P}(\mathcal{P}(V \cup S) \times \mathit{ExpN}_{spec}) \to \mathit{String}^6 \\ &\mathit{vse} \mapsto \mathit{initcode}_2(\{(\mathit{item}, \mathit{expn}) \mid \exists (\mathit{vs}, \mathit{expn}) \in \mathit{vse} \bullet \mathit{item} \in \mathit{vs}\}) \end{aligned}$$

Initialization of pairs of items, i.e. variables or signals, and expression nodes, is defined separately for the local HL3 variables listed above. Therefore, the local variables are determined from the HybridUML variables and signals, along with the expression node. For each local variable, the set of indices is chosen that corresponds to the multiplicity from the HybridUML specification. Each index is associated with a port that is used to read data from the associated channel:

$$\begin{aligned} &\mathit{initcode}_2 : \mathcal{P}((V \cup S) \times \mathit{ExpN}_{spec}) \to \mathit{String}^6 \\ &\mathit{vse} \mapsto (\\ &\mathit{initcode}_3(\{(\{(\mathit{port}_{VN_{basic}}(\mathit{vn}), \mathit{index}_{VN}(\mathit{vn})) \mid \mathit{vn} \in \mathit{vn}_{expn}(v, \mathit{expn})\}, \\ &\qquad \mathit{lvar}_{V,V_{local}}(\mathit{expn}, v, \mathit{read}, \mathit{cur}), \mathit{cur}) \mid v \in V \wedge (v, \mathit{expn}) \in \mathit{vse}\}), \\ &\mathit{initcode}_3(\{(\{(\mathit{port}_{VN_{basic}}(\mathit{vn}), \mathit{index}_{VN}(\mathit{vn})) \mid \mathit{vn} \in \mathit{vn}_{expn}(v, \mathit{expn})\}, \\ &\qquad \mathit{lvar}_{V,V_{local}}(\mathit{expn}, v, \mathit{read}, \mathit{prev}), \mathit{prev}) \mid v \in V \wedge (v, \mathit{expn}) \in \mathit{vse}\}), \\ &\mathit{initcode}_3(\{(\{(\mathit{port}_{VN_{basic}}(\mathit{vn}), \mathit{index}_{VN}(\mathit{vn})) \mid \mathit{vn} \in \mathit{vn}_{expn}(v, \mathit{expn})\}, \\ &\qquad \mathit{lvar}_{V,curTck}(\mathit{expn}, v), \mathit{curTck}) \mid v \in V \wedge (v, \mathit{expn}) \in \mathit{vse}\}), \\ &\mathit{initcode}_3(\{(\{(\mathit{port}_{VN_{basic}}(\mathit{vn}), \mathit{index}_{VN}(\mathit{vn})) \mid \mathit{vn} \in \mathit{vn}_{expn}(v, \mathit{expn})\}, \\ &\qquad \mathit{lvar}_{V,prevTck}(\mathit{expn}, v), \mathit{prevTck}) \mid v \in V \wedge (v, \mathit{expn}) \in \mathit{vse}\}), \\ &\mathit{initcode}_3(\{(\{(\mathit{port}_{SN_{basic}}(\mathit{sn}), \mathit{index}_{SN}(\mathit{sn})) \mid \mathit{sn} \in \mathit{sn}_{expn}(s, \mathit{expn})\}, \\ &\qquad \mathit{lvar}_{S}(\mathit{expn}, s, \mathit{read}), \mathit{cur}) \mid s \in S \wedge (s, \mathit{expn}) \in \mathit{vse}\}), \\ &\mathit{initcode}_3(\{(\{(\mathit{port}_{SN_{basic},param}(\mathit{sn}, n), \mathit{index}_{SN}(\mathit{sn})) \mid \mathit{sn} \in \mathit{sn}_{expn}(s, \mathit{expn})\}, \\ &\qquad \mathit{lvar}_{SigParam}(\mathit{expn}, s, \mathit{read}, n), \mathit{cur}) \mid \\ &\qquad s \in S \wedge (s, \mathit{expn}) \in \mathit{vse} \wedge 0 \le n < |\mathit{paramTypes}_S(s)|\})) \end{aligned}$$

Initialization for sets of local variables with corresponding sets of ports and indices are given wrt. to the kind of data access from the channel:

$$\mathit{Acc}_{initcode} = \{\mathit{cur}, \mathit{prev}, \mathit{curTck}, \mathit{prevTck}\}$$

$$\begin{aligned}
&\mathit{initcode}_3 : \mathcal{P}(\mathcal{P}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var}) \times \mathit{Acc}_{\mathit{initcode}} \rightarrow \mathit{String}\\
&(\mathit{pilv}, \mathit{acc}) \mapsto \mathit{initcode}_4(\mathrm{anyseq}_{\mathrm{seq(Port\times\mathbb{N}_0)\times Var}}\\
&\qquad (\{(\mathrm{anyseq}_{\mathrm{Port\times\mathbb{N}_0}}(\mathit{pi}), \mathit{lv}) \mid (\mathit{pi}, \mathit{lv}) \in \mathit{pilv}\}), \mathit{acc})
\end{aligned}$$

The sets of local variables and the sets of variable and signal nodes are evaluated in an arbitrary order.[2]

The entries of a sequence of local variables with corresponding port and index is then transformed into code, recursively:

$$\begin{aligned}
&\mathit{initcode}_4 : \mathrm{seq}(\mathrm{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var}) \times \mathit{Acc}_{\mathit{initcode}} \rightarrow \mathit{String}\\
&(\langle h \rangle \frown \mathit{sq}, \mathit{acc}) \mapsto \mathit{initcode}_5(h, \mathit{acc}) \frown \mathit{initcode}_4(\mathit{sq}, \mathit{acc})\\
&(\langle\rangle, \mathit{acc}) \mapsto ""
\end{aligned}$$

A single variable along with a sequence of ports and indices is mapped to a sequence of initialization statements, corresponding to the data access type:

$$\begin{aligned}
&\mathit{initcode}_5 : \mathrm{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Acc}_{\mathit{initcode}} \rightarrow \mathit{String}\\
&(\langle (p, n) \rangle \frown \mathit{sq}, \mathit{lv}, \mathit{acc}) \mapsto \mathit{string}(\mathit{lv}) \frown "[" \frown \mathit{string}(n) \frown "] :="\\
&\qquad \frown \mathit{initcode}_6(\mathit{acc}) \frown "(" \frown \mathit{string}(p) \frown "); " \frown \mathit{initcode}_5(\mathit{sq}, \mathit{lv}, \mathit{acc})\\
&\langle\rangle \mapsto ""
\end{aligned}$$

Here, the local variable is used directly, and the corresponding array index is taken from the property or signal node.
Finally, the program statement is determined from the data access type:

$$\begin{aligned}
&\mathit{initcode}_6 : \mathit{Acc}_{\mathit{initcode}} \rightarrow \mathit{String}\\
&\mathit{cur} \mapsto "\texttt{get}"\\
&\mathit{prev} \mapsto "\texttt{getPrevious}"\\
&\mathit{curTck} \mapsto "\texttt{getTime}"\\
&\mathit{prevTck} \mapsto "\texttt{getPreviousTime}"
\end{aligned}$$

Examples: All examples of section **??** initialize their local variables at the beginning of the code. Note two technical differences here: (1) Initialization and declaration of (some) variables is merged for the C++ code. Further, there are additional wrapper variables for technical reasons, e.g. for bounds-safe array access. (2) Some of the variables are references to the corresponding port's data buffer, therefore modification of these variables directly accesses the respective buffer. For this reason, the port's send statements do not explicitly contain the local variable, but use the port's data buffer by default.

6.5.6 Visibility Set Parameter Access

In addition to the initialization of local variables from channels, for some programs, a visibility set parameter is provided. This is accessed by the special HL3 statement `getVisParam`, such that it is stored in dedicated local variables.

[2]The mapping $\mathrm{anyseq}_X : \mathcal{P}(X) \mapsto \mathrm{seq}\, X$ is supposed to define an arbitrary sequence for all elements of a given set $x \subseteq X$, i.e.: $\forall\, x \subseteq X \bullet \mathrm{ran}(\mathrm{anyseq}_X(x)) = x \wedge |\mathrm{anyseq}_X(x)| = |x|$

Therefore, for a set of expression nodes, the corresponding variables are collected as a sequence, but omitting duplicates:

$$\begin{aligned}
&\mathit{visparam}_{\mathrm{seq}\,\mathit{ExpN}} : \mathrm{seq}\,\mathit{ExpN} \to \mathrm{seq}\,\mathit{Var}\\
&\langle\rangle \mapsto \langle\rangle\\
&\langle \mathit{expn}\rangle \frown \mathit{sq} \mapsto \mathit{lvar}_{\mathit{visSet}}(\mathit{expn}) \frown \mathit{visparam}_{\mathrm{seq}\,\mathit{ExpN}}(\mathit{sq})\\
&\qquad ;\ \mathit{expn} \in \mathrm{dom}\,\mathit{lvar}_{\mathit{visSet}} \wedge \mathit{lvar}_{\mathit{visSet}}(\mathit{expn}) \notin \mathit{visparam}_{\mathrm{seq}\,\mathit{ExpN}}(\mathit{sq})\\
&\langle \mathit{expn}\rangle \frown \mathit{sq} \mapsto \mathit{visparam}_{\mathrm{seq}\,\mathit{ExpN}}(\mathit{sq})\,;\ \text{else}
\end{aligned}$$

From a sequence of such variables, a program string of initializations is created:

$$\begin{aligned}
&\mathit{visparamcode}_{\mathrm{seq}\,\mathit{Var}} : \mathrm{seq}\,\mathit{Var} \to \mathit{String}\\
&\langle \mathit{var}\rangle \frown \mathit{sq} \mapsto \mathit{string}(\mathit{var}) \frown \text{"}{:=}\ \texttt{getVisParam}();\text{"} \frown \mathit{visparamcode}_{\mathrm{seq}\,\mathit{Var}}(\mathit{sq})\\
&\langle\rangle \mapsto \langle\rangle
\end{aligned}$$

Then, initialization code for expression node sets is given:

$$\begin{aligned}
&\mathit{initcode}_{\mathit{visparam}} : \mathcal{P}(\mathit{ExpN}) \mapsto \mathit{String}\\
&\mathit{expnset} \mapsto \mathit{visparamcode}_{\mathrm{seq}\,\mathit{Var}}(\mathit{visparam}_{\mathrm{seq}\,\mathit{ExpN}}(\mathrm{anyseq}_{\mathrm{ExpN}}(\mathit{expnset})))
\end{aligned}$$

Examples: C++ provides access to operation parameters implicitly. Of course, this is exploited for the generated code: **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**.

6.5.7 Publication of Local Variable Data to Channels

After program code is executed that implements the evaluation or assignment interpretation of expression nodes, the calculation results are contained in local variables. In order to publish the values, the variables' contents have to be written to channels, via corresponding ports.

The mapping $\mathit{pubcode} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String}^4$ defines the publication code, such that the projections define the publication of the following values:

1. The publication code for local variables containing a new value of a written HybridUML variable is given by

$$\begin{aligned}
&\mathit{pubcode}_V : \mathcal{P}(\mathit{ExpN}) \to \mathit{String}\\
&\qquad \mathit{expn} \mapsto \pi_1\,\mathit{pubcode}(\mathit{expn})
\end{aligned}$$

2. Publication of newly raised HybridUML signal (excluding its parameters) is defined as

$$\begin{aligned}
&\mathit{pubcode}_{S,\mathit{send}} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String}\\
&\qquad \mathit{expn} \mapsto \pi_2\,\mathit{pubcode}(\mathit{expn})
\end{aligned}$$

3. The publication of parameters of a raised signal is given by

$$\begin{aligned}
&\mathit{pubcode}_{\mathit{SigParam}} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String}\\
&\qquad \mathit{expn} \mapsto \pi_3\,\mathit{pubcode}(\mathit{expn})
\end{aligned}$$

4. Publication of the consumption of a trigger only affects the signal channel itself and is defined as

$$\begin{aligned}
&\mathit{pubcode}_{S,recv} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String}\\
&\qquad \mathit{expn} \mapsto \pi_4\, \mathit{pubcode}(\mathit{expn})
\end{aligned}$$

In the same fashion as the initialization code was defined in section 6.5.5, the publication code is defined by a collection of mappings. Sets of expression nodes are mapped to the publication code, corresponding to the hybel trees that represent the contained expressions:

$$\begin{aligned}
&\mathit{pubcode} : \mathcal{P}(\mathit{ExpN}) \to \mathit{String}^4\\
&\mathit{expnset} \mapsto \mathit{pubcode}_1(\{(\mathit{var}_{ht,write}(\mathit{ht}_{expn}(\mathit{expn})), \mathit{expn}) \mid\\
&\qquad \mathit{expn} \in \mathit{expnset} \cap \mathit{ExpN}_{spec}\}\\
&\qquad \cup \{(\mathit{sig}_{ht,write}(\mathit{ht}_{expn}(\mathit{expn})), \mathit{expn}) \mid \mathit{expn} \in \mathit{expnset} \cap \mathit{ExpN}_{spec}\}\\
&\qquad \cup \{(\mathit{sig}_{ht,read}(\mathit{ht}_{expn}(\mathit{expn})), \mathit{expn}) \mid \mathit{expn} \in \mathit{expnset} \cap \mathit{ExpN}_{spec}\})
\end{aligned}$$

This is given by the publication code for sets of variables and signals, within expression nodes:

$$\begin{aligned}
&\mathit{pubcode}_1 : \mathcal{P}(\mathcal{P}(V \cup S) \times \mathit{ExpN}_{spec}) \to \mathit{String}^4\\
&\mathit{vse} \mapsto \mathit{pubcode}_2(\{(\mathit{item}, \mathit{expn}) \mid \exists (\mathit{vs}, \mathit{expn}) \in \mathit{vse} \bullet \mathit{item} \in \mathit{vs}\})
\end{aligned}$$

For the sets of HybridUML variables and signals within an expression node, publication code is distinguished as described above, i.e. sending of (1) written variables, (2) raised signals, (3) the corresponding signal parameters, and (4) consumption of signals are handled separately.

The local variables which hold the new values are determined from the HybridUML variables and signals, along with the expression node. For each local variable, the set of indices corresponding to the multiplicity is created. The ports for writing are determined and attached to the respective indices.

$$\begin{aligned}
&\mathit{pubcode}_2 : \mathcal{P}((V \cup S) \times \mathit{ExpN}_{spec}) \to \mathit{String}^4\\
&\mathit{vse} \mapsto (\\
&\mathit{pubcode}_{3,V}(\{(\{(\mathit{port}_{VN_{basic}}(\mathit{vn}), \mathit{index}_{VN}(\mathit{vn})) \mid \mathit{vn} \in \mathit{vn}_{expn}(v, \mathit{expn})\},\\
&\qquad \mathit{lvar}_{V,V_{local}}(\mathit{expn}, v, \mathit{read}, \mathit{cur}), \mathit{lvar}_{visSet}(\mathit{expn})) \mid\\
&\qquad v \in V \wedge (v, \mathit{expn}) \in \mathit{vse}\}),\\
&\mathit{pubcode}_{3,S,write}(\{(\{(\mathit{port}_{SN_{basic}}(\mathit{sn}), \mathit{index}_{SN}(\mathit{sn})) \mid \mathit{sn} \in \mathit{sn}_{expn}(s, \mathit{expn})\},\\
&\qquad \mathit{lvar}_S(\mathit{expn}, s, \mathit{write}), \mathit{lvar}_{visSet}(\mathit{expn}), \mathit{lvar}_{sigIdx}(\mathit{expn}, s)) \mid\\
&\qquad s \in S \wedge (s, \mathit{expn}) \in \mathit{vse}\}),\\
&\mathit{pubcode}_{3,S,write}(\{(\{(\mathit{port}_{SN_{basic},param}(\mathit{sn}, n), \mathit{index}_{SN}(\mathit{sn})) \mid\\
&\qquad \mathit{sn} \in \mathit{sn}_{expn}(s, \mathit{expn})\},\\
&\qquad \mathit{lvar}_{SigParam}(\mathit{expn}, s, \mathit{write}, n), \mathit{lvar}_{visSet}(\mathit{expn}), \mathit{lvar}_{sigIdx}(\mathit{expn}, s)) \mid\\
&\qquad s \in S \wedge (s, \mathit{expn}) \in \mathit{vse} \wedge 0 \le n < |\mathit{paramTypes}_S(s)|\}),\\
&\mathit{pubcode}_{3,S,read0}(\{\mathit{expn} \mid \exists\, s \in S \bullet (s, \mathit{expn}) \in \mathit{vse}\}) ^\frown\\
&\qquad \mathit{pubcode}_{3,S,read}(\{(\{(\mathit{port}_{SN_{basic}}(\mathit{sn}), \mathit{index}_{SN}(\mathit{sn})) \mid\\
&\qquad \mathit{sn} \in \mathit{sn}_{expn}(s, \mathit{expn})\},
\end{aligned}$$

$$\begin{aligned}
&\mathit{lvar}_S(\mathit{expn}, s, \mathit{read}), \mathit{lvar}_{\mathit{visSet}}(\mathit{expn}), \mathit{lvar}_{\mathit{trgIdx}}(\mathit{expn}, s), \mathit{lvar}_{\mathit{vis}}(\mathit{expn}))\\
&\mid s \in S \wedge (s, \mathit{expn}) \in \mathit{vse}\}))
\end{aligned}$$

For the respective kinds of values, the sets of variables with attached ports and indices are mapped to sequences, and the variables are iterated, in the same fashion as it was done for the initialization code:

$$\begin{aligned}
&\mathit{pubcode}_{3,V} : \mathcal{P}(\mathcal{P}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Var}) \to \mathit{String}\\
&v \mapsto \mathit{pubcode}_{4,V}(\mathrm{anyseq}_{\mathrm{seq}(\mathrm{Port}\times\mathbb{N}_0)\times\mathrm{Var}\times\mathrm{Var}}\\
&\qquad(\{(\mathrm{anyseq}_{\mathrm{Port}\times\mathbb{N}_0}(\mathit{pi}), \mathit{lv}, \mathit{visSet}) \mid (\mathit{pi}, \mathit{lv}, \mathit{visSet}) \in v\}))\\
&\mathit{pubcode}_{3,S,\mathit{write}} : \mathcal{P}(\mathcal{P}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Var} \times \mathit{Var}) \to \mathit{String}\\
&v \mapsto \mathit{pubcode}_{4,S,\mathit{write}}(\mathrm{anyseq}_{\mathrm{seq}(\mathrm{Port}\times\mathbb{N}_0)\times\mathrm{Var}\times\mathrm{Var}\times\mathrm{Var}}\\
&\qquad(\{(\mathrm{anyseq}_{\mathrm{Port}\times\mathbb{N}_0}(\mathit{pi}), \mathit{lv}, \mathit{visSet}, \mathit{sigIdx}) \mid (\mathit{pi}, \mathit{lv}, \mathit{visSet}, \mathit{sigIdx}) \in v\}))\\
&\mathit{pubcode}_{3,S,\mathit{read}} : \mathcal{P}(\mathcal{P}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Var} \times \mathit{Var} \times \mathit{Var}) \to \mathit{String}\\
&v \mapsto \mathit{pubcode}_{4,S,\mathit{write}}(\mathrm{anyseq}_{\mathrm{seq}(\mathrm{Port}\times\mathbb{N}_0)\times\mathrm{Var}\times\mathrm{Var}\times\mathrm{Var}\times\mathrm{Var}}\\
&\qquad(\{(\mathrm{anyseq}_{\mathrm{Port}\times\mathbb{N}_0}(\mathit{pi}), \mathit{lv}, \mathit{visSet}, \mathit{sigIdx}, \mathit{vis}) \mid\\
&\qquad(\mathit{pi}, \mathit{lv}, \mathit{visSet}, \mathit{sigIdx}, \mathit{vis}) \in v\}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{pubcode}_{4,V} : \mathrm{seq}(\mathrm{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Var}) \to \mathit{String}\\
&\langle h\rangle \frown \mathit{sq} \mapsto \mathit{pubcode}_{5,V}(h) \frown \mathit{pubcode}_{4,V}(\mathit{sq})\\
&\langle\rangle \mapsto ""\\
&\mathit{pubcode}_{4,S,\mathit{write}} : \mathrm{seq}(\mathrm{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Var} \times \mathit{Var}) \to \mathit{String}\\
&\langle h\rangle \frown \mathit{sq} \mapsto \mathit{pubcode}_{5,S,\mathit{write}}(h) \frown \mathit{pubcode}_{4,S,\mathit{write}}(\mathit{sq})\\
&\langle\rangle \mapsto ""\\
&\mathit{pubcode}_{4,S,\mathit{read}} : \mathrm{seq}(\mathrm{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Var} \times \mathit{Var} \times \mathit{Var})\\
&\qquad\to \mathit{String}\\
&\langle h\rangle \frown \mathit{sq} \mapsto \mathit{pubcode}_{5,S,\mathit{read}}(h) \frown \mathit{pubcode}_{4,S,\mathit{read}}(\mathit{sq})\\
&\langle\rangle \mapsto ""
\end{aligned}$$

A single variable along with a sequence of ports and indices is mapped to a sequence of publication statements, given by mappings that correspond to the kind of value. Values for HybridUML variables are simply written to the channel. Besides the variable *lv* that holds the new value, the visibility set *visSet* is available, and used directly.

$$\begin{aligned}
&\mathit{pubcode}_{5,V} : \mathrm{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Var} \to \mathit{String}\\
&(\langle(p, n)\rangle \frown \mathit{sq}, \mathit{lv}, \mathit{visSet}) \mapsto "\texttt{put}(" \frown \mathit{string}(p) \frown "," \frown \mathit{string}(\mathit{visSet}) \frown ","\\
&\qquad\frown \mathit{string}(\mathit{lv}) \frown "[" \frown \mathit{string}(n) \frown "]);~" \frown \mathit{pubcode}_{5,V}((\mathit{sq}, \mathit{lv}, \mathit{visSet}))\\
&(\langle\rangle, \mathit{sq}, \mathit{lv}, \mathit{visSet}) \mapsto ""
\end{aligned}$$

Raised signals, as well as signal parameters, are only written if the signal index coincides with the special local signal index variable's value, which is calculated by the code that precedes this publication code. Therefore, it is wrapped by a conditional statement. The available variables are *lv* and *visSet* as above, and the signal index variable *sigIdx* for the comparison.

$$\begin{aligned}
&\mathit{pubcode}_{5,S,write} : \mathrm{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Var} \times \mathit{Var} \to \mathit{String}\\
&(\langle(p,n)\rangle \frown sq, lv, \mathit{visSet}, \mathit{sigIdx}) \mapsto\\
&\qquad \text{"}\texttt{if(}\text{"} \frown \mathit{string}(\mathit{sigIdx}) \frown \text{"==" } \frown \mathit{string}(n) \frown \text{"})\{\text{"} \frown\\
&\qquad \text{"}\texttt{put(}\text{"} \frown \mathit{string}(p) \frown \text{","} \frown \mathit{string}(\mathit{visSet}) \frown \text{","}\\
&\qquad \frown \mathit{string}(lv) \frown \text{"[" } \frown \mathit{string}(n) \frown \text{"]);"}\\
&\qquad \frown \text{"\}"} \frown \mathit{pubcode}_{5,S,write}((sq, lv, \mathit{visSet}, \mathit{sigIdx}))\\
&(\langle\rangle, sq, lv, \mathit{visSet}, \mathit{sigIdx}) \mapsto \text{""}
\end{aligned}$$

Triggers are only consumed, if they are actually received; therefore, the signal index is compared with the special trigger index variable, similarly to the raising of signals, and therefore embedded into a conditional statement. Within, the visibility set for the publication is adapted such that only the sending port itself acts as recipient, such that the consumption of the signal is only locally effective. Therefore, the signal will remain active for all other abstract machines. The variables that are used here are *lv* and *visSet* as before, the trigger index variable *trgIdx* for the index comparison, and the visibility variable *vis*, which is used to define the local visibility of the publication.

$$\begin{aligned}
&\mathit{pubcode}_{5,S,read} : \mathrm{seq}(\mathit{Port} \times \mathbb{N}_0) \times \mathit{Var} \times \mathit{Var} \times \mathit{Var} \times \mathit{Var} \to \mathit{String}\\
&(\langle(p,n)\rangle \frown sq, lv, \mathit{visSet}, \mathit{trgIdx}, \mathit{vis}) \mapsto\\
&\qquad \text{"}\texttt{if(}\text{"} \frown \mathit{string}(\mathit{trgIdx}) \frown \text{"==" } \frown \mathit{string}(n) \frown \text{"})\{\text{"} \frown\\
&\qquad \text{"}\texttt{setRight(}\text{"} \frown \mathit{string}(\mathit{vis}) \frown \text{","} \frown \mathit{string}(p) \frown \text{");"} \frown\\
&\qquad \text{"}\texttt{clear(}\text{"} \frown \mathit{string}(\mathit{visSet}) \frown \text{");"} \frown\\
&\qquad \text{"}\texttt{addEntry(}\text{"} \frown \mathit{string}(\mathit{visSet}) \frown \text{","} \frown \mathit{string}(\mathit{vis}) \frown \text{");"} \frown\\
&\qquad \text{"}\texttt{put(}\text{"} \frown \mathit{string}(p) \frown \text{","} \frown \mathit{string}(\mathit{visSet}) \frown \text{","}\\
&\qquad \frown \mathit{string}(lv) \frown \text{"[" } \frown \mathit{string}(n) \frown \text{"]);"} \frown\\
&\qquad \frown \text{"\}"} \frown \mathit{pubcode}_{5,S,read}((sq, lv, \mathit{visSet}, \mathit{trgIdx}, \mathit{vis}))\\
&(\langle\rangle, sq, lv, \mathit{visSet}, \mathit{trgIdx}, \mathit{vis}) \mapsto \text{""}
\end{aligned}$$

As a prerequisite for the calculation of the visibility set above, the mapping $\mathit{pubcode}_{4,S,read0}$ defines code to precede all conditional statements for signal consumption, that prepares the publication time and visibility *vis* for the value's publication. In order to iterate the affected expression nodes beforehand, the mapping $\mathit{pubcode}_{3,S,read0}$ provides a sequence for this.

$$\begin{aligned}
&\mathit{pubcode}_{3,S,read0} : \mathcal{P}(\mathit{ExpN}_{spec}) \to \mathit{String}\\
&\mathit{expnset} \mapsto \mathit{pubcode}_{4,S,read0}(\mathrm{anyseq}_{\mathrm{ExpN_{spec}}}(\mathit{expnset}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{pubcode}_{4,S,read0} : \mathrm{seq}\,\mathit{ExpN}_{spec} \to \mathit{String}\\
&\langle \mathit{expn}\rangle \frown sq \mapsto\\
&\qquad \mathit{string}(\mathit{lvar}_{newTck}(\mathit{expn})) \frown \text{":="} \frown \text{"}\texttt{getCurrentTime();}\text{"} \frown\\
&\qquad \text{"}\texttt{setRight(}\text{"} \frown \mathit{string}(\mathit{lvar}_{newTck}(\mathit{expn})) \frown \text{",}\texttt{right(}\text{"}\\
&\qquad \frown \mathit{string}(\mathit{lvar}_{newTck}(\mathit{expn})) \frown \text{")+1;"} \frown\\
&\qquad \text{"}\texttt{setLeft(}\text{"} \frown \mathit{string}(\mathit{lvar}_{vis}(\mathit{expn})) \frown \text{","}
\end{aligned}$$

$\frown string(lvar_{newTck}(expn)) \frown$ "); "
$\frown pubcode_{4,S,read0}(sq)$
$\langle\rangle \mapsto$ ""

Examples: **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**. Note that the port's send statements do not explicitly contain a reference to a local variable, because they use a port's internal data buffer. Local variables for write access are defined as references to the corresponding buffer on initialization.

6.6 HybridUML Abstract Subject Execution

This section provides the effects of the operations of abstract subjects. In section 3.5, HL3 operational rules were defined for the execution of abstract subjects. For each of these rules, a corresponding *effect function* was declared, but not defined, because their definitions depend on the high-level formalism for which a HL3 model is created. Therefore, the definitions of the effect functions given here are specific to HybridUML.

There are two kinds of abstract subjects – (1) Abstract Machines and (2) the Selector. Correspondingly, the definitions are structured into sections 6.6.1 and 6.6.2.

The operations' definitions rely on some *internal state* of abstract subjects, which is also specific for the high-level formalism HybridUML. Internal state is different for abstract machines and the selector, therefore the HybridUML internal state is the union of both:

$$IntState = IntState_{Am} \cup IntState_{sel}$$

6.6.1 Abstract Machines

The sequential behavior of an abstract machine $am \in Am$ is defined on the basis of the tree of mode instance nodes of the associated agent instance node $mtree_{AIN}(ain_{Am}(am))$ – it defines the *static* structure of a hierarchic state-machine. The dynamic structure is defined as internal state $IntState_{Am}$ of abstract machines, and consists of a (1) *history* mapping and a (2) *current control point* mapping.

Internal State

The internal state of HybridUML abstract machines is defined by

$$IntState_{Am} = HIST \times CURCP$$
$$HIST = MIN_{spec} \to MIN_{spec} \cup \{\lambda\}$$
$$CURCP = MIN_{spec} \to CPIN \cup \{\lambda\}$$

The *history* mapping $hist \in HIST$ assigns a currently active submode for each mode instance node, or λ to denote that there is none. For a given history and a root mode instance node, the history implies the current *mode configuration*

$$modeconf : HIST \times (MIN_{spec} \cup \{\lambda\}) \to \operatorname{seq} MIN_{spec}$$
$$(hist, min) \mapsto \langle min\rangle \frown modeconf(hist, hist(min))\ ;\ min \neq \lambda$$
$$(hist, \lambda) \mapsto \langle\rangle$$

which is the sequence of active mode instance nodes descending the mode instance node tree. The mode configuration is *complete*, if the sequence ends with a leaf node, that is a node representing a mode that has no submodes. Note that the history further contains submodes of *inactive* mode instance nodes, to be restored for the mode configuration later.

The history always assigns mode instance nodes to their own submodes:

$$\begin{aligned}&\forall\, hist \in HIST, min \in MIN_{spec} \bullet\\ &\qquad hist(min) \neq \lambda \Rightarrow min = head(path_{MIN}(hist(min)))\end{aligned}$$

Each mode instance node may have a current control point instance node, defined by a mapping $curcp \in CURCP$, which defines a control point of the mode itself:

$$\begin{aligned}&\forall\, curcp \in CURCP, min \in MIN_{spec} \bullet\\ &\qquad curcp(min) \neq \lambda \Rightarrow min = min_{CPIN}(curcp(min))\end{aligned}$$

The current control point mapping denotes which transitions are possibly enabled – only transitions that originate from a current control point may ever fire.

Effect of Abstract Machines' Initialization

The initialization of abstract machines defines an internal state $s_i \in IntState_{Am}$ such that (1) the history is empty, and (2) there is exactly one current control point – the default entry of the abstract machine's top-level mode instance node:

$$\begin{aligned}&init_{Am} : Am \rightarrow IntState\\ &am \mapsto (\{min \mapsto \lambda \mid min \in MIN_{spec}\},\\ &\qquad \{min \mapsto \lambda \mid min \in MIN_{spec}\} \oplus \{min \mapsto de_{min} \mid\\ &\qquad\ \ min = \mathrm{node}_{\mathrm{MIN_{spec}}}(mtree_{AIN}(ain_{Am}(am))) \wedge\\ &\qquad\ \ de_M(mode_{MI}(mi_{MIN}(min))) = cp_{CPI}(cpi_{CPIN}(de_{min}))\})\end{aligned}$$

Effect of Abstract Machines' Update

The updating of an abstract machine determines (1) its currently enabled transitions, (2) the enabledness of the abstract machine for a flow step, and (3) the enabling of particular flows for participation in such a flow step. This *update* functionality is defined step-by-step by use of several functions:

Effect of Evaluation Programs. For the determination of enabled transitions, flows, and flow-enabledness, the conditions that are given by HybridUML trigger expressions, guard expressions, and invariant constraint expressions, are evaluated. Therefore, programs that implement the evaluation interpretation of expression nodes are created and executed, and their boolean result is evaluated.

For this, we define the *effect of evaluation programs*. In contrast to the execution of HybridUML flow constraints and actions, the progress of evaluation programs for HybridUML triggers, guards, and invariant constraints wrt. time is *not* considered explicitly:

1. Assignment programs for HybridUML flow constraints and actions (and triggers) were defined in section 6.4.2, and are directly represented by HL3 flows and transitions. They act as HL3 program subjects, and their execution semantics is defined per program statement, by the progress rules for program statements in section 3.6.

2. Evaluation programs for HybridUML triggers, guards, and invariant constraints are defined below. For their execution, the effect of evaluation programs is defined on the basis of the same program statement effects from section 3.6, but no progress of time is defined here. However, for the complete *update* operation, an execution time duration is assumed by the progress rule for operations of abstract subjects (section 3.5.1).
   The programs that are generated for triggers, guards, and invariant constraints may be executed in parallel, within the HL3 scheduling phase *update_phase*, but they do not interfere, because they all use their own local variables (defined in section 6.4.1), and do not publish any values on channels. Therefore it is sufficient to consider the overall execution time for the *update* operation.

The effect of an execution of a complete evaluation program is given by

$$\begin{aligned}
&\epsilon* : \mathit{Program} \times \mathit{Param}_{prog} \times \mathit{Subject} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \\
&\quad \to \mathit{VAR}_{mwrite} \\
&(prg, param, s, c, v) \mapsto \epsilon*(\epsilon_{prg}(prg, param, s, c, v), param, s, c, \\
&\quad (\epsilon_{var}(prg, param, s, c, v), \epsilon_{chan}(prg, param, s, c, v))) \\
&\quad ; prg \neq \langle\rangle \\
&(\langle\rangle, param, s, c, v) \mapsto (\epsilon_{var}(prg, param, s, c, v), \epsilon_{chan}(prg, param, s, c, v))
\end{aligned}$$

For a program *prg*, the sequence of effects of the program's statements result in the effect of the program, as soon as the program is processed completely. The boolean result of the program is contained in the resulting variable valuation.

Evaluation of Boolean Expression Nodes. Each expression node that contains a boolean expression, as defined by HybridUML trigger expressions, guard expressions, and invariant constraint expressions, (1) defines an evaluation program, and (2) provides a boolean result on the execution of this program.

The program's definition is given by the mapping $prg_{ExpN,eval}$ of section 6.5. Then, the boolean result of the execution of a program is given by its effect (defined by $\epsilon*$), and the dedicated return variable's value is returned:

$$\begin{aligned}
&exec_{bexp} : \mathit{BExpN}_{spec} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \to \mathbb{B} \\
&(expn, c, v) \mapsto \\
&\quad \sigma_{Var}(\epsilon*(prg_{ExpN,eval}(\langle expn\rangle), \lambda, ain_{ExpN}(expn), c, v)) \\
&\qquad (retvar_{BExpN}(expn))
\end{aligned}$$

Filtering of Initial Transitions. For the determination of enabled transitions, the first step is the filtering of initial transitions: Transitions that *initialize* modes can only be enabled, if the mode's history is empty. The following mapping returns *false* for initial transition nodes of mode instance nodes with non-empty history:

$$\begin{aligned}
&\mathit{checkInitTrans}_{TN} : \mathit{TN}_{spec} \times \mathit{IntState}_{Am} \to \mathbb{B}\\
&(tn, (\mathit{hist}, \mathit{curcp})) \mapsto\\
&\qquad (\mathit{min}_{CPIN}(\mathit{src}_{TN_{spec}}(tn)) = \mathit{min}_{TN}(tn)\\
&\qquad\ \wedge \mathit{cp}_{CPI}(\mathit{cpi}_{CPIN}(\mathit{src}_{TN_{spec}}(tn)))\\
&\qquad\qquad = \mathit{de}_M(\mathit{mode}_{MI}(\mathit{mi}_{MIN}(\mathit{min}_{TN}(tn)))))\\
&\qquad \Rightarrow \mathit{hist}(\mathit{min}_{TN}(tn)) = \lambda
\end{aligned}$$

Here, always if (1) the transition's parent mode coincides with the parent mode of its source control point, (2) such that the source control point is the default entry point of that mode, then the mode's history must be empty for the given transition to be enabled.

Enabling of Transitions. Transition nodes can be enabled or disabled, depending on the trigger and guard expressions. For HybridUML, enabled transitions are *urgent*, if they have a trigger expression that is satisfied. Enabled transitions without trigger expression are not urgent, i.e. they do not need to be executed, but can. Additionally, initial transitions are disabled, whenever the parent mode has non-empty history.

$$\begin{aligned}
&\mathit{isEnabled}_{TN} : \mathit{TN}_{spec} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \to \mathbb{B}\\
&(tn, s_i, c, v) \mapsto\\
&\qquad \mathit{checkInitTrans}_{TN}(tn, s_i) \wedge\\
&\qquad \bigwedge_{txn \in \mathit{trg}_{TN_{spec}}(tn) \cup \mathit{grd}_{TN_{spec}}(tn)} \mathit{exec}_{bexp}(txn, c, v)\\
&\mathit{isUrgent}_{TN} : \mathit{TN}_{spec} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \to \mathbb{B}\\
&(tn, s_i, c, v) \mapsto \mathit{isEnabled}_{TN}(tn, s_i, c, v) \wedge \mathit{trg}_{TN_{spec}}(tn) \neq \varnothing
\end{aligned}$$

Note that trigger and guard expressions are evaluated by the execution of the corresponding evaluation programs, as defined before.

Enabled Transitions of Control Points. For each control point instance node, the set of enabled transitions consists of its outgoing transitions that are enabled:

$$\begin{aligned}
&\mathit{enabledTrans}_{CPIN} : \mathit{CPIN}_{spec} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread}\\
&\qquad \to \mathcal{P}(\mathit{TN}_{spec})\\
&(\mathit{cpin}, s_i, c, v) \mapsto\\
&\qquad \{tn \in \mathit{TN}_{spec} \mid \mathit{src}_{TN_{spec}}(tn) = \mathit{cpin} \wedge \mathit{isEnabled}_{TN}(tn, s_i, c, v)\}
\end{aligned}$$

Analogously, urgent transitions are collected for control points:

$$\begin{aligned}
&\mathit{urgentTrans}_{CPIN} : \mathit{CPIN}_{spec} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread}\\
&\qquad \to \mathcal{P}(\mathit{TN}_{spec})\\
&(\mathit{cpin}, s_i, c, v) \mapsto\\
&\qquad \{tn \in \mathit{TN}_{spec} \mid \mathit{src}_{TN_{spec}}(tn) = \mathit{cpin} \wedge \mathit{isUrgent}_{TN}(tn, s_i, c, v)\}
\end{aligned}$$

Enabled Transitions of Mode Configurations. For a mode configuration with root mode instance node *min*, the currently enabled transitions are collected: (1) The enabled transitions of the root mode's current control point are included, if there is one, and (2) all enabled transitions of its active submode are added recursively, if the history is not empty:

$$\begin{aligned}
&\mathit{enabledTrans}_{MIN} : \mathit{MIN}_{spec} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \\
&\qquad \to \mathcal{P}(\mathit{TN}_{spec}) \\
&(\mathit{min}, (\mathit{hist}, \mathit{curcp}), c, v) \mapsto \{\mathit{tn} \in \mathit{TN}_{spec} \mid \\
&\qquad \exists\, \mathit{cpin} \in \mathit{CPIN}_{spec} \bullet (\mathit{curcp}(\mathit{min}) = \mathit{cpin} \\
&\qquad\quad \wedge\ \mathit{tn} \in \mathit{enabledTrans}_{CPIN}(\mathit{cpin}, (\mathit{hist}, \mathit{curcp}), c, v)) \\
&\qquad \vee\ \exists\, \mathit{min}_2 \in \mathit{MIN}_{spec} \bullet (\mathit{hist}(\mathit{min}) = \mathit{min}_2 \\
&\qquad\quad \wedge\ \mathit{tn} \in \mathit{enabledTrans}_{MIN}(\mathit{min}_2, (\mathit{hist}, \mathit{curcp}), c, v))\}
\end{aligned}$$

Similarly, urgent transitions are determined:

$$\begin{aligned}
&\mathit{urgentTrans}_{MIN} : \mathit{MIN}_{spec} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \\
&\qquad \to \mathcal{P}(\mathit{TN}_{spec}) \\
&(\mathit{min}, (\mathit{hist}, \mathit{curcp}), c, v) \mapsto \{\mathit{tn} \in \mathit{TN}_{spec} \mid \\
&\qquad \exists\, \mathit{cpin} \in \mathit{CPIN}_{spec} \bullet (\mathit{curcp}(\mathit{min}) = \mathit{cpin} \\
&\qquad\quad \wedge\ \mathit{tn} \in \mathit{urgentTrans}_{CPIN}(\mathit{cpin}, (\mathit{hist}, \mathit{curcp}), c, v)) \\
&\qquad \vee\ \exists\, \mathit{min}_2 \in \mathit{MIN}_{spec} \bullet (\mathit{hist}(\mathit{min}) = \mathit{min}_2 \\
&\qquad\quad \wedge\ \mathit{tn} \in \mathit{urgentTrans}_{MIN}(\mathit{min}_2, (\mathit{hist}, \mathit{curcp}), c, v))\}
\end{aligned}$$

Enabled Transitions of Abstract Machines. The first part of the *update* result – the set of enabled transitions of an abstract machine – is given by the transition nodes that are enabled for the top-level mode of the abstract machine:

$$\begin{aligned}
&\mathit{enabledTrans}_{Am} : \mathit{Am} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \to \mathcal{P}(\mathit{Trans}) \\
&(\mathit{am}, s_i, c, v) \mapsto \{t \in \mathit{Trans} \mid \mathit{tn}_{trans}(t) \in \\
&\qquad \mathit{enabledTrans}_{MIN}(\mathrm{node}_{\mathrm{MIN}_{\mathrm{spec}}}(\mathit{mtree}_{AIN}(\mathit{ain}_{Am}(\mathit{am}))), s_i, c, v)\}
\end{aligned}$$

Satisfied Invariant Constraints of Mode Configurations. The first step for the determination of the second part of the *update* result, that is the check for the abstract machine's enabledness for a flow step, is the evaluation of the invariant constraints of mode configurations. Therefore, their conjunction is evaluated by

$$\begin{aligned}
&\mathit{checkInv}_{MIN} : \mathit{MIN}_{spec} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \to \mathbb{B} \\
&(\mathit{min}, (\mathit{hist}, \mathit{curcp}), c, v) \mapsto \bigwedge_{\mathit{mxn} \in \mathit{inv}_{\mathit{MIN}_{spec}}(\mathit{min})} \mathit{exec}_{bexp}(\mathit{mxn}, c, v) \\
&\qquad \wedge\ ((\exists\, \mathit{min}_2 \in \mathit{MIN}_{spec} \bullet \mathit{hist}(\mathit{min}) = \mathit{min}_2) \\
&\qquad\quad \Rightarrow \mathit{checkInv}_{MIN}(\mathit{min}_2, (\mathit{hist}, \mathit{curcp}), c, v))
\end{aligned}$$

That is, the invariant constraints of the root mode are checked, and recursively the invariant constraints for the currently active submode, if there is one. Note that for the evaluation of invariant constraints, a corresponding program is executed, as defined before.

Stable Mode Configurations. Additionally, flow-enabledness requires a *stable* mode configuration. A mode configuration is stable, if it is complete, and for all active modes, the current control point is the respective *default exit point.* Therefore, for the mode configuration's root mode instance node, (1) the current control point is checked, and (2) if it is no leaf mode, then a non-empty history is required. (3) Finally, the history mode is examined recursively.

$$\begin{aligned}
&\mathit{checkDx}_{MIN} : \mathit{MIN}_{spec} \times \mathit{IntState}_{Am} \to \mathbb{B}\\
&((s_M, mi, ain), (hist, curcp)) \mapsto\\
&\quad \exists(min_2, cpi) \in \mathit{CPIN}_{spec} \bullet (curcp(s_M, mi, ain) = (min_2, cpi)\\
&\qquad \wedge\ cp_{CPI}(cpi) = dx_M(mode_{MI}(mi)))\\
&\quad \wedge\\
&\quad (submode_M(mode_{MI}(mi)) \neq \varnothing\\
&\quad \Rightarrow \exists\ min_2 \in \mathit{MIN}_{spec} \bullet\\
&\qquad min_2 = hist(min) \wedge \mathit{checkDx}_{MIN}(min_2, (hist, curcp)))
\end{aligned}$$

Flow-Enabledness of Mode Configurations. A mode configuration permits a flow of time, whenever (1) its invariants are satisfied, (2) the configuration is stable, and additionally, (3) there are no urgent transitions.

$$\begin{aligned}
&\mathit{isFlowPossible}_{MIN} : \mathit{MIN}_{spec} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \to \mathbb{B}\\
&(min, s_i, c, v) \mapsto \mathit{checkDx}_{MIN}(min, s_i) \wedge \mathit{checkInv}_{MIN}(min, s_i, c, v)\\
&\quad \wedge\ \mathit{urgentTrans}_{MIN}(min, s_i, c, v) = \varnothing
\end{aligned}$$

Flow-Enabledness of Abstract Machines. The permission of an abstract machine to let time pass – the second part of the *update* result – is determined by the top-level mode of the abstract machine:

$$\begin{aligned}
&\mathit{isFlowPossible}_{Am} : \mathit{Am} \times \mathit{IntState}_{Am} \times \mathit{CONST}_m \times \mathit{VAR}_{mread} \to \mathbb{B}\\
&(am, s_i, c, v) \mapsto\\
&\quad \mathit{isFlowPossible}_{MIN}(\mathrm{node}_{\mathrm{MIN}_{\mathrm{spec}}}(\mathit{mtree}_{AIN}(ain_{Am}(am))), s_i, c, v)
\end{aligned}$$

Activation of Associated Flow Expressions. The third part of the *update* result is defined on the basis of a given mode instance node tree: All associated flow expressions are activated correspondingly to the tree's mode configuration: (1) On activation of the root mode's flow expression, the current active submode's flow expressions are activated, too, recursively. All inactive submodes' flow expressions are deactivated. (2) On deactivation of the root mode's flow expressions, all submodes' flow expressions are deactivated, too.

$$\begin{aligned}
&\mathit{activateFlows}_{MIN} : \mathit{MIN}_{spec} \times \mathit{IntState}_{Am} \times \mathbb{B} \to (\mathit{FlowExpN} \nrightarrow \mathbb{B})\\
&(min, (hist, curcp), b) \mapsto\\
&\quad \mathit{flow}_{MIN_{spec}}(min) \times \{b\} \cup\\
&\qquad \bigcup_{min_2 \in chld} \mathit{activateFlows}_{MIN}(min_2, (hist, curcp), \mathit{false}) \cup\\
&\qquad \bigcup_{min_2 \in \{hist(min)\} \setminus \{\lambda\}} \mathit{activateFlows}_{MIN}(min_2, (hist, curcp), b)\\
&;\ chld = \mathrm{children}_{\mathrm{MIN}_{\mathrm{spec}}}(\mathit{tree}_{MIN}(min)) \setminus \{hist(min)\}
\end{aligned}$$

Activation of Abstract Machines' Flow Expressions. The flow expressions of an abstract machine's top-level mode are always *activated.* From the corresponding mode configuration, the activation or deactivation of flow expressions of submodes result.

$$\begin{aligned}
&\mathit{activateFlows}_{Am} : \mathit{Am} \times \mathit{IntState}_{Am} \to (\mathit{Flow} \nrightarrow \mathbb{B})\\
&(am, s_i) \mapsto \{f \mapsto b \mid \mathit{fxn}_{Flow}(f) \mapsto b \in\\
&\qquad \mathit{activateFlows}_{MIN}(\mathrm{node}_{\mathrm{MIN}_{\mathrm{spec}}}(\mathit{mtree}_{AIN}(\mathit{ain}_{Am}(am))), s_i, \mathit{true})\}
\end{aligned}$$

Definition of *update*. The effect of the operation *update* of a HybridUML abstract machine *am* is the combination of (1) the flow-enabledness of *am*, (2) the enabled transitions of *am*, and (3) the activation state of the associated flows:

$$\begin{aligned}
&\mathit{update} : \mathit{Am} \times \mathit{IntState} \times \mathit{CONST}_m \times \mathit{VAR}_{mread}\\
&\qquad \to \mathit{IntState} \times \mathit{AmState} \times (\mathit{Flow} \nrightarrow \mathbb{B})\\
&(am, s_i, c, v) \mapsto\\
&\qquad (s_i, (\mathit{isFlowPossible}_{Am}(am, s_i, c, v), \mathit{enabledTrans}_{Am}(am, s_i, c, v)),\\
&\qquad\ \ \mathit{activateFlows}_{Am}(am, s_i))\\
&\qquad ;\ s_i \in \mathit{IntState}_{Am}
\end{aligned}$$

Formally, the result of *update* for $s_i \notin \mathit{IntState}_{Am}$ is arbitrary; but this is not effective since init_{Am}, *update*, and *notify* always define a resulting state $s_i \in \mathit{IntState}_{Am}$.

Further note that HybridUML abstract machines only read, but do not modify their internal state on update.

Effect of Abstract Machines' Notification

The notification of a HybridUML abstract machine occurs whenever in a preceding *transition_phase* of the HL3 model execution a transition was executed that is associated with the abstract machine. Then the abstract machine adjusts its internal state by taking the transition without executing its actions, such that the transition's source control point is left and its target control point is entered, potentially affecting the history as well.

Entering of Control Points. When a control point instance node *cpin* is entered, it becomes the current control point instance node of its parent mode instance node, i.e. $\mathit{curcp}(\mathit{min}_{CPIN}(\mathit{cpin})) = \mathit{cpin}$. The mapping

$$\mathit{setcurcp} : \mathit{IntState}_{Am} \times \mathit{CPIN}_{spec} \to \mathit{CURCP}$$

defines this, but the entering of a control point has several further implications. (1) If the control point is the default entry, then the history is resumed recursively for the mode, if possible:

$$\begin{aligned}
&((\mathit{hist}, \mathit{curcp}), \mathit{cpin}) \mapsto\\
&\qquad \mathit{setcurcp}((\mathit{hist}, \mathit{curcp} \oplus \{\mathit{min}_{CPIN}(\mathit{cpin}) \mapsto \mathit{cpin}\}), \mathit{de}_h)\\
&;\ \mathit{cp}_{CPI}(\mathit{cpi}_{CPIN}(\mathit{cpin})) = \mathit{de}_M(\mathit{mode}_{MI}(\mathit{mi}_{MIN}(\mathit{min}_{CPIN}(\mathit{cpin}))))\\
&\qquad \wedge\ \mathit{min}_{CPIN}(\mathit{de}_h) = \mathit{hist}(\mathit{min}_{CPIN}(\mathit{cpin}))\\
&\qquad \wedge\ \mathit{cp}_{CPI}(\mathit{cpi}_{CPIN}(\mathit{de}_h)) = \mathit{de}_M(\mathit{mode}_{MI}(\mathit{mi}_{MIN}(\mathit{min}_{CPIN}(\mathit{de}_h))))
\end{aligned}$$

(2) If for a leaf mode the control point is the default entry, then it is directly transferred to the default exit:

$$\begin{aligned}
&((\mathit{hist}, \mathit{curcp}), \mathit{cpin}) \mapsto \mathit{setcurcp}((\mathit{hist}, \mathit{curcp}), \mathit{dx})\\
&;\ \mathit{cp}_{\mathit{CPI}}(\mathit{cpi}_{\mathit{CPIN}}(\mathit{cpin})) = \mathit{de}_M(\mathit{mode}_{\mathit{MI}}(\mathit{mi}_{\mathit{MIN}}(\mathit{min}_{\mathit{CPIN}}(\mathit{cpin}))))\\
&\qquad \wedge \mathit{cp}_{\mathit{CPI}}(\mathit{cpi}_{\mathit{CPIN}}(\mathit{dx})) = \mathit{dx}_M(\mathit{mode}_{\mathit{MI}}(\mathit{mi}_{\mathit{MIN}}(\mathit{min}_{\mathit{CPIN}}(\mathit{cpin}))))\\
&\qquad \wedge \mathit{submode}_M(\mathit{mode}_{\mathit{MI}}(\mathit{mi}_{\mathit{MIN}}(\mathit{min}_{\mathit{CPIN}}(\mathit{cpin})))) = \varnothing
\end{aligned}$$

(3) If the control point is the default exit, then all parent modes are also set to their default exits recursively:

$$\begin{aligned}
&((\mathit{hist}, \mathit{curcp}), \mathit{cpin}) \mapsto\\
&\qquad \mathit{setcurcp}((\mathit{hist}, \mathit{curcp} \oplus \{\mathit{min}_{\mathit{CPIN}}(\mathit{cpin}) \mapsto \mathit{cpin}\}), \mathit{dx}_p)\\
&;\ \mathit{cp}_{\mathit{CPI}}(\mathit{cpi}_{\mathit{CPIN}}(\mathit{cpin})) = \mathit{dx}_M(\mathit{mode}_{\mathit{MI}}(\mathit{mi}_{\mathit{MIN}}(\mathit{min}_{\mathit{CPIN}}(\mathit{cpin}))))\\
&\qquad \wedge \mathit{cp}_{\mathit{CPI}}(\mathit{cpi}_{\mathit{CPIN}}(\mathit{de}_p)) = \mathit{dx}_M(\mathit{mode}_{\mathit{MI}}(\mathit{mi}_{\mathit{MIN}}(\mathit{min}_{\mathit{CPIN}}(\mathit{de}_p))))\\
&\qquad \wedge \mathit{mi}_{\mathit{MIN}}(\mathit{min}_{\mathit{CPIN}}(\mathit{cpin}))\\
&\qquad\quad \in \mathit{submode}_M(\mathit{mode}_{\mathit{MI}}(\mathit{mi}_{\mathit{MIN}}(\mathit{min}_{\mathit{CPIN}}(\mathit{dx}_p))))
\end{aligned}$$

(4) By default, the current control point of the mode is just set:

$$((\mathit{hist}, \mathit{curcp}), \mathit{cpin}) \mapsto \mathit{curcp} \oplus \{\mathit{min}_{\mathit{CPIN}}(\mathit{cpin}) \mapsto \mathit{cpin}\}\ ;\ \text{else}$$

Leaving of Control Points. Leaving of a control point instance node *cpin* assigns the special value λ to the parent mode instance node, in order to indicate that it has no current control point (anymore): $\mathit{curcp}(\mathit{min}_{\mathit{CPIN}}(\mathit{cpin})) = \lambda$. This is defined by the mapping

$$\begin{aligned}
&\mathit{unsetcurcp} : \mathit{IntState}_{\mathit{Am}} \times \mathit{CPIN}_{\mathit{spec}} \rightarrow \mathit{CURCP}\\
&((\mathit{hist}, \mathit{curcp}), \mathit{cpin}) \mapsto \mathit{unsethistcp}((\mathit{hist}, \mathit{unsetparcp}(\mathit{curcp} \oplus\\
&\qquad \{\mathit{min}_{\mathit{CPIN}}(\mathit{cpin}) \mapsto \lambda\}, \mathit{min}_{\mathit{CPIN}}(\mathit{cpin}))), \mathit{min}_{\mathit{CPIN}}(\mathit{cpin}))
\end{aligned}$$

The complete mode configuration gets unset automatically, i.e. the control points of all parent modes and history modes are unset, too, recursively in both cases:

$$\begin{aligned}
&\mathit{unsetparcp} : \mathit{CURCP} \times \mathit{MIN}_{\mathit{spec}} \rightarrow \mathit{CURCP}\\
&(\mathit{curcp}, \mathit{min}) \mapsto \mathit{unsetparcp}(\mathit{curcp} \oplus \{\mathit{min}_p \mapsto \lambda\}, \mathit{min}_p)\\
&\qquad ;\ \mathit{mi}_{\mathit{MIN}}(\mathit{min}) \in \mathit{submode}_M(\mathit{mode}_{\mathit{MI}}(\mathit{mi}_{\mathit{MIN}}(\mathit{min}_p)))\\
&(\mathit{curcp}, \mathit{min}) \mapsto \mathit{curcp}\ ;\ \text{else}
\end{aligned}$$

$$\begin{aligned}
&\mathit{unsethistcp} : \mathit{IntState}_{\mathit{Am}} \times \mathit{MIN}_{\mathit{spec}} \rightarrow \mathit{CURCP}\\
&((\mathit{hist}, \mathit{curcp}), \mathit{min}) \mapsto \mathit{unsethistcp}((\mathit{hist}, \mathit{curcp} \oplus \{\mathit{min}_h \mapsto \lambda\}), \mathit{min}_h)\\
&\qquad ;\ \mathit{min}_h \neq \lambda \wedge \mathit{min}_h = \mathit{hist}(\mathit{min})\\
&((\mathit{hist}, \mathit{curcp}), \mathit{min}) \mapsto \mathit{curcp}\ ;\ \text{else}
\end{aligned}$$

Modification of the History. On the firing of a transition, the history is modified, corresponding to the transition's target control point:

$$\mathit{newhist} : \mathit{HIST} \times \mathit{TN}_{\mathit{spec}} \rightarrow \mathit{HIST}$$

In the majority of cases, transitions (indirectly) connect submodes, such that the history of their own parent mode is set to the mode of the target control point. Because of the syntactical constraints for HybridUML transitions (see section 4.2), this situation is identified by the direction of the control point, i.e. when it is an entry point:

$$\begin{aligned}&(\mathit{hist}, \mathit{tn}) \mapsto \mathit{hist} \oplus \{\mathit{min}_{TN}(\mathit{tn}) \mapsto \mathit{min}_{CPIN}(\mathit{tar}_{TN_{spec}}(\mathit{tn}))\}\\ &;\ \mathit{kind}_{CP}(\mathit{cp}_{CPI}(\mathit{tar}_{TN_{spec}}(\mathit{tn}))) = \mathit{entry}\end{aligned}$$

Otherwise, i.e. if the control point is an exit point, the transition leads to the exit point of its own parent mode. Since this must not be the default exit (for the same syntactical restrictions as above), the history is *cleared* for that mode:

$$\begin{aligned}&(\mathit{hist}, \mathit{tn}) \mapsto \mathit{hist} \oplus \{\mathit{min}_{TN}(\mathit{tn}) \mapsto \lambda\}\\ &;\ \mathit{kind}_{CP}(\mathit{cp}_{CPI}(\mathit{tar}_{TN_{spec}}(\mathit{tn}))) = \mathit{exit}\end{aligned}$$

Firing of Transitions. A transition fires by (1) leaving its source control point, (2) modifying its parent mode's history, and (3) entering its target control point:

$$\begin{aligned}&\mathit{fire} : \mathit{IntState}_{Am} \times \mathit{TN}_{spec} \to \mathit{IntState}_{Am}\\ &((\mathit{hist}, \mathit{curcp}), \mathit{tn}) \mapsto (\mathit{setcurcp}(\\ &\qquad(\mathit{newhist}(\mathit{hist}, \mathit{src}_{TN_{spec}}(\mathit{tn})),\\ &\qquad\ \ \mathit{unsetcurcp}((\mathit{hist}, \mathit{curcp}), \mathit{src}_{TN_{spec}}(\mathit{tn}))),\\ &\qquad\mathit{tar}_{TN_{spec}}(\mathit{tn})),\\ &\quad\ \mathit{newhist}(\mathit{hist}, \mathit{src}_{TN_{spec}}(\mathit{tn})))\end{aligned}$$

Definition of *notify*. The effect of the operation *notifyTrans* of a HybridUML abstract machine *am* is then defined by the firing of the given transition, provided that it is a transition of *am*:

$$\begin{aligned}&\mathit{notify} : \mathit{Am} \times \mathit{IntState} \times \mathit{Trans} \to \mathit{IntState}\\ &(\mathit{am}, s_i, t) \mapsto \mathit{fire}(s_i, \mathit{tn}_{trans}(t))\ ;\ s_i \in \mathit{IntState}_{Am} \wedge \mathit{am}_{huml,trans}(t) = \mathit{am}\\ &(\mathit{am}, s_i, t) \mapsto s_i\ ;\ \text{else}\end{aligned}$$

6.6.2 HybridUML Selector

The selector defined in this section is the *HybridUML simulation selector*. It is tailored for the simulation of HybridUML specifications, such that a maximum set of HybridUML-specific HL3 model executions is defined. Any non-determinism of the HybridUML specification is actually simulated non-deterministically. In contrast, a *HybridUML test selector* would restrict the set of executions by applying some elaborate test selection algorithm, or a *HybridUML implementation selector* could solve non-determinism by making fixed choices.

Internal State

The HybridUML selector only has internal state that models non-determinism. We do not define this explicitly, but assume that there is a function

$$rndstep_{selector} : IntState_{sel} \to IntState_{sel}$$

that calculates some kind of random sequence on the internal state.

Effect of Selector's Initialization

The initialization of the selector has two separate effects: (1) Corresponding to the notion of the internal state of the selector given above, we assume that the initialization defines an appropriate starting point for the random sequence modeling non-determinism. (2) Since the selector has the responsibility to check the initial HL3 model state for well-formedness, the initial valuation of the model's channels is checked whether it satisfies the init state constraints of the agent instance nodes, represented by the abstract machines of the HL3 model.

Initial State Well-Formedness Check. In order to check the init state constraints of the HybridUML model (see also sections 6.4 and 5.1 for discussions of init state constraints), expression nodes that represent the constraints are evaluated. This is done in the same fashion as for trigger expressions, guard expressions, and invariant constraint expressions in section 6.6.1, for the definition of the abstract machine's operation *update*. Expression nodes define a respective program that is executed and that returns a boolean result value. The initial state is well-formed, if the conjunction of all constraints' results is satisfied:

$$\begin{aligned}
&check_{initState} : S \to \mathbb{B} \\
&\quad (c, v) \mapsto \textstyle\bigwedge_{ixn \in isc_{Am}(c,v)} \quad exec_{bexp}(ixn, \\
&\qquad\qquad\qquad\qquad (subject_{var}(c), chan_{port}(c), subject_{port}(c)), \\
&\qquad\qquad\qquad\qquad (modelTime(v), \sigma_{Var}(v), \kappa_{Chan}(v)))
\end{aligned}$$

The init state constraints are provided by the abstract machines of the HL3 model:

$$\begin{aligned}
&isc_{Am} : S \to \mathcal{P}(BExpN_{spec}) \\
&(c, v) \mapsto \bigcup_{m \in am(c)} allisc_{AIN_{spec}}(ain_{Am}(m))
\end{aligned}$$

Definition of $\mathbf{init_{sel}}$. The effect of the operation *init* of the HybridUML simulation selector is (1) the creation of the internal state, and (2) the well-formedness check of the channels' initial valuation:

$$\begin{aligned}
&init_{sel} : S \to IntState \times \mathbb{B} \\
&s \mapsto (s_i, check_{initState}(s)) \\
&\text{with appropriate internal state } s_i
\end{aligned}$$

Remember from section 3.5.2 that the consequence of a *failed* well-formedness check is that there is no valid execution of the complete HL3 model.

Effect of Selector's Selection

The calculation of a selection by the selector either (1) defines a set of transitions to be executed in a *transition_phase*, or (2) chooses a *flow_phase*, such that the currently activated flows are calculated. Additionally, HybridUML signals are unset before a succeeding *flow_phase*, in order to implement a zero-time duration for signals.

In the following, several functions are defined and combined to the complete functionality of the selector's operation *getSelection*.

Flow-Enabledness of the Simulation. The possibility of a flow step of the simulation is determined in a straight-forward way: a flow step is only admissible, if all abstract machines allow it:

$$\begin{aligned}&\mathit{selectflow} : \Sigma_{Am} \to \mathbb{B}\\ &s_{Am} \mapsto \forall\, m \in \operatorname{dom} s_{Am} \bullet \mathit{flow}_{AmState}(s_{Am}(m))\end{aligned}$$

Collection of Enabled Transitions. For the selection of transitions, the separate sets of transitions from the abstract machines are collected, such that a set of transition sets results that contains the set of enabled transitions for each abstract machine:

$$\begin{aligned}&\mathit{collecttrans} : \Sigma_{Am} \to \mathcal{P}(\mathcal{P}(\mathit{Trans}))\\ &s_{Am} \mapsto \{\mathit{trs} \in \mathcal{P}(\mathit{Trans}) \mid \exists\, m \in \operatorname{dom} s_{Am} \bullet \mathit{trs} = \mathit{trans}_{AmState}(s_{Am}(m))\}\end{aligned}$$

Non-Conflicting Transitions. The HybridUML semantics is an *interleaving semantics*, i.e. logically there are no parallel HL3 transitions (i.e. discrete steps), but transitions are executed sequentially. The strict definition of this interleaving would lead to transition phases of the HL3 model execution that only execute one single HL3 transition, with the drawback that for n available light weight processes, only one could be active, while the others would be idle. As an optimization of the execution (with $n > 1$), transitions that are *non-conflicting* can be executed in parallel, without modifying the interleaving semantics.

Since local HL3 variables are exclusively accessed by each subject, and therefore no racing conditions occur *during* the parallel execution, only the execution *sequence* is significant. Two transitions are non-conflicting, whenever neither of both transitions relies on the results of the other one, such that any execution sequence leads to the same result, which is the result of the logical sequence of the interleaving semantics.

This is guaranteed, if the HL3 channels from which one transition reads values (the *read set*) are disjoint from the channels on which the other transition writes (the *write set*):

$$\begin{aligned}&\mathit{nonconfl} : \mathit{Trans} \times \mathit{Trans} \to \mathbb{B}\\ &(t_1, t_2) \mapsto \mathit{readset}(t_1) \cap \mathit{writeset}(t_2) = \varnothing\\ &\qquad \wedge\ \mathit{readset}(t_2) \cap \mathit{writeset}(t_1) = \varnothing\end{aligned}$$

For each two transitions, *nonconfl* maps to *true* if the transitions are non-conflicting, and to *false* if they are in conflict.

The read sets and write sets are defined as mappings from transitions to sets of channels. All channels for HybridUML variables and signals are collected for which an expression node exists, such that the variable or signal is accessed (read or written, resp.) by the corresponding expression. From section 6.3, mappings to determine variables and signals that are accessed by expressions are applied:

$$\begin{aligned}
&\mathit{readset} : \mathit{Trans} \rightarrow \mathcal{P}(\mathit{CHN}_{\mathit{Var}} \cup \mathit{CHN}_{\mathit{Sig}})\\
&t \mapsto \{c \in \mathit{CHN}_{\mathit{Var}} \mid \exists\, \mathit{vn} \in \mathit{vn}_{\mathit{conn},\mathit{max},\mathit{CHN}_{\mathit{Var}}}(c) \bullet\\
&\qquad \exists\, \mathit{expn} \in \mathit{trg}_{\mathit{TN}_{\mathit{spec}}}(\mathit{tn}_{\mathit{trans}}(t)) \cup \operatorname{ran} \mathit{act}_{\mathit{TN}_{\mathit{spec}}}(\mathit{tn}_{\mathit{trans}}(t)) \bullet\\
&\qquad \mathit{var}_{\mathit{VN}}(\mathit{vn}) \in \mathit{var}_{\mathit{ht},\mathit{read}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}))\}\\
&\qquad \cup\\
&\qquad \{c \in \mathit{CHN}_{\mathit{Sig}} \mid \exists\, \mathit{sn} \in \mathit{sn}_{\mathit{conn},\mathit{max},\mathit{CHN}_{\mathit{Sig}}}(c) \bullet\\
&\qquad\; \exists\, \mathit{expn} \in \mathit{trg}_{\mathit{TN}_{\mathit{spec}}}(\mathit{tn}_{\mathit{trans}}(t)) \cup \operatorname{ran} \mathit{act}_{\mathit{TN}_{\mathit{spec}}}(\mathit{tn}_{\mathit{trans}}(t)) \bullet\\
&\qquad\; \mathit{sig}_{\mathit{SN}}(\mathit{sn}) \in \mathit{sig}_{\mathit{ht},\mathit{read}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}))\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{writeset} : \mathit{Trans} \rightarrow \mathcal{P}(\mathit{CHN}_{\mathit{Var}} \cup \mathit{CHN}_{\mathit{Sig}})\\
&t \mapsto \{c \in \mathit{CHN}_{\mathit{Var}} \mid \exists\, \mathit{vn} \in \mathit{vn}_{\mathit{conn},\mathit{max},\mathit{CHN}_{\mathit{Var}}}(c) \bullet\\
&\qquad \exists\, \mathit{expn} \in \mathit{trg}_{\mathit{TN}_{\mathit{spec}}}(\mathit{tn}_{\mathit{trans}}(t)) \cup \operatorname{ran} \mathit{act}_{\mathit{TN}_{\mathit{spec}}}(\mathit{tn}_{\mathit{trans}}(t)) \bullet\\
&\qquad \mathit{var}_{\mathit{VN}}(\mathit{vn}) \in \mathit{var}_{\mathit{ht},\mathit{write}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}))\}\\
&\qquad \cup\\
&\qquad \{c \in \mathit{CHN}_{\mathit{Sig}} \mid \exists\, \mathit{sn} \in \mathit{sn}_{\mathit{conn},\mathit{max},\mathit{CHN}_{\mathit{Sig}}}(c) \bullet\\
&\qquad\; \exists\, \mathit{expn} \in \mathit{trg}_{\mathit{TN}_{\mathit{spec}}}(\mathit{tn}_{\mathit{trans}}(t)) \cup \operatorname{ran} \mathit{act}_{\mathit{TN}_{\mathit{spec}}}(\mathit{tn}_{\mathit{trans}}(t)) \bullet\\
&\qquad\; \mathit{sig}_{\mathit{SN}}(\mathit{sn}) \in \mathit{sig}_{\mathit{ht},\mathit{write}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn})) \cup \mathit{sig}_{\mathit{ht},\mathit{read}}(\mathit{ht}_{\mathit{expn}}(\mathit{expn}))\}
\end{aligned}$$

Note that read signals are included in the write set, because signals are consumed on reception, and therefore the corresponding value is written on the respective channel.

Sets of Non-Conflicting Transitions. From sets of transition sets, every possible allowed transition combination can be generated, such that (1) at most one transition per input set is included, and (2) there are no conflicting transitions in each output set. This prepares the selection of a set of transitions of the HL3 model for a (possibly) succeeding *transition_phase*. Every abstract machine can take one transition at most, because their behavior is *sequential.* Their transitions may not interfere, in order to guarantee a valid execution optimization for an arbitrary interleaving of the transitions, because every possible execution sequence of these transitions is allowed.

$$\begin{aligned}
&\mathit{nonconfltrans} : \mathcal{P}(\mathcal{P}(\mathit{Trans})) \rightarrow \mathcal{P}(\mathcal{P}(\mathit{Trans}))\\
&\{\mathit{trs}_1, \ldots, \mathit{trs}_n\} \mapsto \{\mathit{trs} \in \mathcal{P}(\bigcup_{i=1}^{n} \mathit{trs}_i) \mid \forall\, t_1, t_2 \in \mathit{trs}, k, l \in \{1..n\} \bullet\\
&\qquad ((t_1 \in \mathit{trs}_k \wedge t_2 \in \mathit{trs}_l \wedge t_1 \neq t_2 \Rightarrow \mathit{trs}_k \neq \mathit{trs}_l)\\
&\qquad \wedge \mathit{nonconfl}(t_1, t_2))\}
\end{aligned}$$

Choice of a Transition Set. From the set of allowed transition combinations, one is chosen:

$$\mathit{selecttrans} : \Sigma_{\mathit{Am}} \times \mathit{IntState}_{\mathit{sel}} \rightarrow \mathcal{P}(\mathit{Trans})$$

$$\begin{array}{l} \quad (s_{Am}, s_i) \mapsto \mathit{trs} \\ \text{such that} \\ \mathit{trs} \in \mathit{nonconfltrans}(\mathit{collecttrans}(s_{Am})) \\ \wedge\, (\exists\, \mathit{trs}_2 \in \mathit{nonconfltrans}(\mathit{collecttrans}(s_{Am})) \bullet \mathit{trs}_2 \neq \varnothing) \Rightarrow \mathit{trs} \neq \varnothing \end{array}$$

Here, mainly the given internal selector state determines which of the available transition sets is chosen. The choice is not entirely non-deterministic, because $\mathit{trs} \neq \varnothing$ is always preferred. Otherwise the HL3 model execution could step through any number of empty transition phases, and therefore the model's real-time execution could fail (see section 3.4.1 for successful model executions) even if successful runs exist for the model.

Choice of Flow or Transition Phase. Based on the availability of transitions and the enabledness for a flow step, the selector chooses either a *transition_phase* or a *flow_phase*. (1) If both transitions are available and a flow step is possible, depending on the internal selector state, a random choice is made, such that either the transitions are removed or the flow step is disabled. (2) Otherwise, the transition set and the flow flag are left untouched. HybridUML models exist for which a *transition_phase* with empty transition set can result here, they are deemed to be *not well-formed*.

$$\begin{array}{l} \mathit{selectflowortrans} : \Sigma_{Am} \times \mathit{IntState}_{sel} \to \mathbb{B} \times \mathcal{P}(\mathit{Trans}) \\ \quad (s_{Am}, s_i) \mapsto (\mathit{flow}, \mathit{trs}) \end{array}$$

$$\begin{array}{l} (\mathit{selectflow}(s_{Am}) \wedge \mathit{selecttrans}(s_{Am}, s_i) \neq \varnothing) \Rightarrow \\ \quad ((\mathit{flow}, \mathit{trs}) \in \{(\mathit{true}, \varnothing), (\mathit{false}, \mathit{selecttrans}(s_{Am}, s_i))\}) \\ (\neg \mathit{selectflow}(s_{Am}) \vee \mathit{selecttrans}(s_{Am}, s_i) = \varnothing) \Rightarrow \\ \quad ((\mathit{flow}, \mathit{trs}) = (\mathit{selectflow}(s_{Am}), \mathit{selecttrans}(s_{Am}, s_i))) \end{array}$$

Creation of a Selection. To all transitions which are selected, a visibility set is added that defines when the results of the respective action would become visible. For HybridUML, this is always the current tick, incremented by 0.1, i.e. no time passes, but causality between succeeding transition phases is modeled. This publication time tick is the same for all potential recipients:

$$\begin{array}{l} \mathit{selection} : \Sigma_{Am} \times \mathit{IntState}_{sel} \times \mathit{ModelTime} \to \mathit{Selection} \\ (s_{Am}, s_i, \mathit{tick}) \mapsto (\pi_1\, \mathit{selectflowortrans}(s_{Am}, s_i), \\ \quad \{\mathit{tr} \mapsto \mathit{Port} \times \{\mathit{tick} + 0.1\} \mid \mathit{tr} \in \pi_2\, \mathit{selectflowortrans}(s_{Am}, s_i)\}) \end{array}$$

Resetting of Signals. As a side-effect of the selection, the HybridUML selector resets signals whenever model time evolves, i.e. when it selects a flow phase. This ensures that signals have no time duration. To all available signal channels, the value *false* is written for all possible recipients, in order to indicate that no signal is active anymore. For this, it is assumed that the given port set P contains ports for exactly the respective channels. The publication time is the current time tick, increased by 0.1, which will be earlier than current model time in the next update phase (when signals will be read next). That means, signals are reset immediately.

$$\begin{aligned}&\mathit{resetSignals} : \mathit{ModelTime} \times \mathcal{P}(\mathit{Port}) \rightarrow \mathit{ChanState}\\&(\mathit{tick}, P) \mapsto (\{\mathit{tick} + 0.1\} \times P) \times \{\mathit{false}\}\end{aligned}$$

Definition of *select*. The main effect of the operation *getSelection* of the HybridUML simulation selector is the selection between *flow_phase* and *transition_phase*, along with the transitions and their visibility sets. Additionally, the signal channels' state may be modified, due to the resetting of signals. Finally, the internal state is adjusted.

$$\begin{aligned}&\mathit{select} : \mathit{ModelTime} \times \mathcal{P}(\mathit{Port}) \times \mathit{IntState} \times \Sigma_{Am}\\&\qquad \rightarrow \mathit{IntState} \times \mathit{Selection} \times \mathit{ChanState}\\&(\mathit{tick}, P, s_i, s_{Am}) \mapsto (\mathit{rndstep}_{selector}(s_i), \mathit{selection}(s_{Am}, s_i, \mathit{tick}), s_{chan})\\&\text{with } s_{chan} = \begin{cases} \mathit{resetSignals}(\mathit{tick}, P) & \text{if } \pi_1\, \mathit{selectflowortrans}(s_{Am}, s_i)\\ \varnothing & \text{else}\end{cases}\end{aligned}$$

Bibliography

[AGLS01] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 33–48, 2001.

[AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[BBB$^+$99] Tom Bienmller, Jrgen Bohn, Henning Brinkmann, Udo Brockmeyer, Werner Damm, Hardi Hungar, and Peter Jansen. Verification of automotive control units. In *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 319–341, 1999.

[BBHP03] Kirsten Berkenkötter, Stefan Bisanz, Ulrich Hannemann, and Jan Peleska. HybridUML Profile for UML 2.0. SVERTS Workshop at the ≪UML≫ 2003 Conference, October 2003. http://www-verimag.imag.fr/EVENTS/2003/SVERTS/.

[BZL04] Stefan Bisanz, Paul Ziemann, and Arne Lindow. Integrated Specification, Validation and Verification with HybridUML and OCL applied to the BART Case Study. In Eckehard Schnieder and Géza Tarnai, editors, *FORMS/FORMAT 2004. Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages 191–203, Braunschweig, December 2004. Proceedings of Symposium FORMS/FORMAT 2004, Braunschweig, Germany, 2nd and 3rd December 2004. ISBN 3-9803363-8-7.

[Efk05] Christof Efkemann. Development and evaluation of a hard real-time scheduling modification for linux 2.6. Master's thesis, Universität Bremen, April 2005.

[Hen96] Thomas A. Henzinger. The theory of Hybrid Automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[HHK03] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[JLSE99] Karl Henrik Johansson, John Lygeros, Shankar Sastry, and Magnus Egerstedt. Simulation of zeno hybrid automata. In *Proceedings of the IEEE Conference on Decision and Control*, volume 4, pages 3538–3543, December 1999.

[KMP00] Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. *Acta Informatica*, 36(11):836–912, 2000.

[Kop97a] Hermann Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Applications*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1997.

[Kop97b] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1997.

[OMG] Object Management Group. Object Constraint Language (OCL) specification. http://www.uml.org.

[OMG06] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/docs/formal/06-01-01.pdf, January 2006.

[Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[RRS03] Mauno Rönnkö, Anders P. Ravn, and Kaisa Sere. Hybrid Action Systems. *Theoretical Computer Science*, 290:937–973, January 2003.

[RTA] Real-Time Application Interface (RTAI). http://www.rtai.org. Originally founded by the Department of Aerospace Engineering of Politecnico di Milano (DIAPM), http://www.aero.polimi.it/~rtai/about/index.html.

[RTL] RTLinux. http://www.rtlinux.org. FSMLabs, Inc.

[ZRH93] Chaochen Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 763 of *Lecture Notes in Computer Science*, pages 36–59. The Computer Society of the IEEE, 1993. Extended abstract.