Integration of Heterogeneous Modelling Techniques using High-Level Constraints

Benjamin Braatz

Bericht-Nr. 2003–10
ISSN 1436–9915

Integration of Heterogeneous Modelling Techniques using High-Level Constraints

Benjamin Braatz

E-mail: bbraatz@cs.tu-berlin.de

Abstract

The aim of this report is to present a formal framework for the integration of heterogeneous modelling techniques. This framework is based on the integration paradigm for data type and process modelling techniques by Ehrig and Orejas.

This specification frame, consisting of integrated signatures and models, is used as common syntax and semantics for the heterogeneous techniques. The concepts are divided into layers for static parts, dynamic state spaces and processes.

As structuring concepts union over common subparts and refinement are introduced.

For the specification of properties the abstract concept of logics of constraints is used. As examples of constraints conditional equations and pre- and post-conditions of state transformations are defined.

The integration of modelling techniques is also done in terms of constraints. More precisely, specifications from heterogeneous techniques are considered as so-called high-level constraints, which specify certain parts of the system.

Contents

1 Introduction **6**

2 Formal Framework **8**

2.1 Integrated Signatures and Model Categories 8

2.1.1 Data Type Layer 9

2.1.2 Data State Layer 12

2.1.3 Transformation Layer 15

2.1.4 Process Layer 20

2.1.5 Integration 25

2.2 Signature Morphisms and Restriction Construction 27

2.2.1 Data Type Layer 28

2.2.2 Data State Layer 31

2.2.3 Transformation Layer 35

2.2.4 Process Layer 38

2.2.5 Integration 41

2.3 Integrated Signature Specification Frame 43

3 Structuring Concepts **47**

3.1 Union of Signatures and Amalgamation of Models 47

3.1.1 Data Type Layer 48

3.1.2 Data State Layer 50

3.1.3 Transformation Layer 51

3.1.4 Process Layer 53

3.1.5 Integration 55

3.2 Signature Refinements and Abstraction of Models 56

3.2.1 Data Type Layer 56

3.2.2 Data State Layer 59

3.2.3 Transformation Layer 60
3.2.4 Process Layer 62
3.2.5 Integration 64

4 Constraints 66

4.1 Logics of Constraints and Specifications 66
4.1.1 Logics of Constraints 66
4.1.2 Specifications and Specification Morphisms 68
4.2 Constraints for Integrated Signatures 69
4.2.1 Data Type and Data State Constraints 69
4.2.2 Transformation and Process Constraints 71
4.3 High-Level Constraints 73
4.3.1 Instantiations as Constraints 73
4.3.2 CCS Process Algebra 75
4.3.3 Attributed Graph Transformation 77

5 Example 82

5.1 Basic Requirements 82
5.2 Abstract Asymmetric Encryption 83
5.3 Net Structure and Communication 86
5.4 RSA Algorithm 88
5.5 Net Structure Changes 92

6 Conclusion 94

Chapter 1

Introduction

The aim of this report is to present a framework for the integration of heterogeneous modelling techniques. For this purpose a common formal syntax and semantics is defined, which is based on the formal framework for the integration of data type and process modelling techniques by Ehrig and Orejas, presented in [EO01a, EO01b, EOB$^+$02b].

This framework divides systems into layers, which reflects the different roles of static parts, staying the same over the whole lifetime of the system, dynamic data states and transformations, describing the state space of the system, and processes on these state spaces.

Transformations have a local flavour, they just transform one data state into another one, without assuming other transformations to happen in advance or to follow. In contrast to that, processes describe the possible interactions of transformations, by assigning an interpretation in the state space to whole transition systems.

A large class of heterogeneous modelling techniques can be interpreted in this framework, as already shown in [EO01a]. Such interpretations define a semantics for the specifications of the techniques in terms of classes of integrated models satisfying the specification.

These interpretations are used to define "high-level" constraints on integrated signatures by adding the information, how the parts described by the specification can be found in the integrated signature. This leads to the possibility of combining specifications from different modelling techniques, which describe different parts of an integrated system.

For example, class diagrams could be used to describe the structure of the data type and the data states, algebraic specifications with initial semantics can describe the static operations of a system, rule-based approaches, such as graph transformation, can specify the effect of transformations and techniques with operational semantics, defined in terms of transition systems or related structures, can be interpreted to define processes of a system.

The report is organized as follows:

In Chapter 2 integrated signatures and models are introduced as a formal framework constituting the common syntax and semantics. Moreover, the relations between different signatures and models of them are captured in terms of signature morphisms and model restrictions.

In Chapter 3 structuring concepts for integrated signatures are examined. The union of signatures over a common subpart leads to the amalgamation of corresponding models, while the refinement of more abstract signatures by more concrete ones leads to the possibility of abstracting models of the concrete signature to become models of the abstract one.

In Chapter 4 logics of constraints are introduced as means to specify properties of systems. As examples of “low-level” constraints conditional equations and pre- and post-conditions for transformations are presented. Moreover the concept of “high-level constraints” is explained. As examples of high-level constraints, CCS processes and algebraic graph transformation are examined.

In Chapter 5 a larger example of a message system with asymmetric encryption is specified using the techniques developed in the report.

Finally, in Chapter 6 some remarks on related work and future perspectives conclude the report.

Acknowledgements:

I wish to thank Hartmut Ehrig, Fernando Orejas, Markus Klein and Martti Piirainen for many fruitful discussions and comments.

This work is partially supported by the DFG research project “Researcher Group Petri-Net-Technology” and the DFG project IOSIP within the priority programme “Integration of Software Specification Techniques for Applications in Engineering”.

Chapter 2

Formal Framework

In this chapter we will introduce the formal framework, which shall be used as a common syntax and semantics for different kinds of modelling techniques. This framework is a modification of the framework for the integration of data type and process modelling techniques proposed in [EO01a] and elobarated in [EO01b, EOB$^+$02b].

Systems can be divided into four layers introduced in [EO98]:

1. Data Types: This layer contains the static parts of a system in terms of data values and operations on them.

2. Data States and Transformations: On this layer the dynamic behaviour of a system is captured. There are a class of data states the system can be in and transformations between these states.

3. Processes: The reactive states and transitions are subject of this layer. It is used to model how transformations can be composed in the system.

4. System Architecture: On this layer means for structuring system descriptions are introduced.

The framework of this chapter will deal with the first three layers, where in the following we will divide the second layer into a data state and a transformation part. The fourth layer is addressed by the structuring concepts in Chapter 3.

2.1 Integrated Signatures and Model Categories

Integrated signatures are syntactic descriptions of the structure of systems, while on the semantical side integrated models are representations of systems whose structure is given by an integrated signature.

The relations between different models of the same signature are captured in terms of integrated model homomorphisms. A model homomorphism h describes how a more "general" model can be found in a less "general" one.

Model homomorphisms for integrated models were not considered so far in [EO01a, EO01b, EOB+02b].

To get proper model categories there also has to be a composition of homomorphisms, which is associative and has neutral identities for each model.

We will first look at the layers seperately in Subsections 2.1.1 through 2.1.4 and then define the integrated notions by just putting the pieces together in Subsection 2.1.5.

2.1.1 Data Type Layer

On the data type layer the static part of a system is described by an algebraic signature on the syntactical side.

Definition 2.1.1 (Data Type Signature)
A *data type signature* is an algebraic signature Σ_{DT}, which is a set $\Sigma_{DT,S}$ of *sort symbols* and a set $\Sigma_{DT,Op}$ of *operation symbols* with functions *dom* and *cod* assigning *domains* and *codomains* to operation symbols.

$$\begin{array}{ll} \Sigma_{DT} = (\Sigma_{DT,S}, \Sigma_{DT,Op}, dom, cod) & \\ \text{with} & \Sigma_{DT,S} \in \mathbf{Set} \\ & \Sigma_{DT,Op} \in \mathbf{Set} \\ & dom\colon \Sigma_{DT,Op} \to \Sigma^*_{DT,S} \\ & cod\colon \Sigma_{DT,Op} \to \Sigma_{DT,S} \cup \{\lambda\} \end{array}$$

We will write data type signatures as follows:

$$\begin{array}{lll} \text{data type:} & & \\ \text{sorts:} & s_1, \ldots & (s_1 \in \Sigma_{DT,S}) \\ \text{opns:} & op\colon dom(op) \to cod(op) & (op \in \Sigma_{DT,Op}) \end{array}$$

Operation symbols P with empty codomain $cod(P) = \lambda$ are interpreted as predicate symbols and written $P\colon dom(P)$. The category of algebraic signatures will be denoted by **AlgSig** in the following.

Data type models are partial algebras, which interpret the structure given by an algebraic signature by carrier sets for sort symbols and partial functions for operation symbols.

Definition 2.1.2 (Data Type Model)
Given a data type signature Σ_{DT}, a *data type model* is a partial Σ_{DT}-algebra M_{DT}, i.e. a sort-indexed family of sets M_s and an operation-indexed family of partial functions op_M.

$$\begin{array}{ll} M_{DT} = \left((M_s)_{s \in \Sigma_{DT,S}}, (op_M)_{op \in \Sigma_{DT,Op}}\right) & \\ \text{with} & M_s \in \mathbf{Set} \\ & op_M\colon M_{dom(op)} \to\!\!\!\!\circ\;\, M_{cod(op)} \\ \text{where} & M_{s_1 \ldots s_n} = M_{s_1} \times \ldots \times M_{s_n} \\ & M_\lambda = \{*\} \end{array}$$

Operation symbols *con* with empty domain $dom(con) = \lambda$ are interpreted as constants $con_M = con_M(*) \in M_{cod(op)}$ and operation symbols P with empty codomain $cod(P) = \lambda$ as predicates with $(x_1, \ldots, x_n) \in P_M$ if and only if $P_M(x_1, \ldots, x_n)$ defined with $P_M(x_1, \ldots, x_n) = *$ and, hence, $(x_1, \ldots, x_n) \notin P_M$ if and only if $P_M(x_1, \ldots, x_n)$ is undefined.

For operation symbols *op* we denote by the domain $Dom(op_M)$ of *op* in M_{DT} the set of all elements of $M_{dom(op)}$ for which op_M is defined.

A homomorphism between data types is a partial algebra homomorphism, which is a family of (total) functions between the sorts, being compatible with the operations. Elements of the sorts of the source algebra can be identified, leading to "confusion". The sorts of the target algebra can contain "junk", i. e. additional elements, which are not in the codomain of the homomorphism. For partial algebras there are different types of homomorphisms depending on how undefined values are treated. We choose a version where the image of an undefined operation application can be defined.

Definition 2.1.3 (Data Type Model Homomorphism)
Given two data type models M_{DT} and N_{DT}, a *data type model homomorphism* is a partial algebra homomorphism $h_{DT}: M_{DT} \to N_{DT}$, i.e. a sort-indexed family of total functions $h_s: M_s \to N_s$. These functions have to be compatible with the operations in the sense that they assign to the result of an operation application in M_{DT} the result of the operation application on the images of the arguments in N_{DT} (see Fig. 2.1). If an operation application is undefined in M_{DT}, the corresponding application in N_{DT} can either be defined (and the result is not restricted in any way) or undefined.

$$
\begin{array}{ll}
\multicolumn{2}{l}{h_{DT}: M_{DT} \to N_{DT} = (h_s: M_s \to N_s)_{s \in \Sigma_{DT,S}}} \\
\text{with} & h_{dom(op)}(Dom(op_M)) \subseteq Dom(op_N) \\
 & op_N \circ h_{dom(op)} \simeq h_{cod(op)} \circ op_M \\
\text{where} & h_{s_1 \ldots s_n} = h_{s_1} \times \ldots \times h_{s_n} \\
 & h_\lambda = \{* \mapsto *\} \\
 & f \simeq g \iff \forall x \in Dom(f) \cap Dom(g): f(x) = g(x)
\end{array}
$$

$$
\begin{tikzcd}
M_{dom(op)} \arrow[r, "h_{dom(op)}"] \arrow[d, "op_M"'] \arrow[dr, phantom, "(\simeq)"] & N_{dom(op)} \arrow[d, "op_N"] \\
M_{cod(op)} \arrow[r, "h_{cod(op)}"'] & N_{cod(op)}
\end{tikzcd}
$$

Figure 2.1: Data type homomorphisms

Now we define compositions and identities for data type model homomorphisms.

Definition and Fact 2.1.4 (Data Type Homomorphism Composition)
Given two data type homomorphisms $g_{DT}: L_{DT} \to M_{DT}$ and $h_{DT}: M_{DT} \to N_{DT}$, the *composition* $h_{DT} \circ g_{DT}: L_{DT} \to N_{DT}$ is an integrated model homomorphism defined by the composition of the functions for all sorts.

$$h_{DT} \circ g_{DT} = ((h \circ g)_s)_{s \in \Sigma_{DT,S}} = (h_s \circ g_s)_{s \in \Sigma_{DT,S}}$$

The composition of data type model homomorphisms is associative, i. e. $(h_{DT} \circ g_{DT}) \circ f_{DT} = h_{DT} \circ (g_{DT} \circ f_{DT})$ for all $f_{DT}: K_{DT} \to L_{DT}$, $g_{DT}: L_{DT} \to M_{DT}$ and $h_{DT}: M_{DT} \to N_{DT}$.

Proof: We have to show that the composition results is a proper homomorphism and is associative.

Homomorphism Properties: The properties for h_{DT} and g_{DT} have to imply the property for $h_{DT} \circ g_{DT}$, i. e. the definedness of operation applications has to be preserved and the operation applications have to be compatible with the homomorphism.

- For the preservation of definedness of operation applications we get:

$$\begin{array}{ll} & h_{dom(op)}(g_{dom(op)}(Dom(op_L))) \\ \subseteq & (g_{DT} \text{ data type homomorphism, Def. 2.1.3}) \\ & h_{dom(op)}(Dom(op_M)) \\ \subseteq & (h_{DT} \text{ data type homomorphism, Def. 2.1.3}) \\ & Dom(op_N) \end{array}$$

- For the compatibility of the operation applications themselves we get:

$$\begin{array}{ll} & op_N \circ h_{dom(op)} \circ g_{dom(op)} \\ \simeq & (h_{DT} \text{ data type homomorphism, Def. 2.1.3}) \\ & h_{cod(op)} \circ op_M \circ g_{dom(op)} \\ \simeq & (g_{DT} \text{ data type homomorphism, Def. 2.1.3}) \\ & h_{cod(op)} \circ g_{cod(op)} \circ op_L \end{array}$$

Associativity: The associativity follows directly from the associatvity of composition for functions and families of functions.

Hence the composition of data type homomorphisms leads to data type homomorphisms and is associative. □

Definition and Fact 2.1.5 (Data Type Identities)
Given a data type model M_{DT}, the *identity* $id_{M_{DT}}: M_{DT} \to M_{DT}$ is a data type model homomorphism defined by identities for all sorts.

$$id_{M_{DT}} = (id_s)_{s \in \Sigma_{DT,S}} = (id_{M_s})_{s \in \Sigma_{DT,S}}$$

The identities of data type models are neutral w. r. t. composition, i. e. $h_{DT} \circ id_{M_{DT}} = h_{DT} = id_{N_{DT}} \circ h_{DT}$ for all $h_{DT}: M_{DT} \to N_{DT}$.

Proof: The homomorphism properties of identities are obviously satisfied, because identities preserve domains and $op_N \circ id_{dom(op)} = op_N = id_{cod(op)} \circ op_N$.

The neutrality of identities follows directly from the neutrality of identity functions. □

The category of all partial algebras and partial algebra homomorphisms for an algebraic signature Σ will be denoted by $PAlg(\Sigma)$.

Example: As an example let us consider a signature Nat_{DT} for the natural numbers.

$Nat_{DT} =$
data type:
sorts: nat
opns: $zero\colon nat$
$succ\colon nat \to nat$
$add\colon nat\, nat \to nat$
$sub\colon nat\, nat \to nat$

Now the model NAT_{DT} for this signature is the intended interpretation with natural numbers as sort NAT_{nat} and the constant 0, the successor function, addition and partial subtraction as operations.

$$
\begin{aligned}
&NAT_{nat} = \mathbb{N} \\
&zero_{NAT} = 0 \\
&succ_{NAT}(n) = n + 1 \\
&add_{NAT}(n, m) = n + m \\
&sub_{NAT}(n, m) = \begin{cases} n - m, \text{ if } n \geq m \\ \text{undefined, else} \end{cases}
\end{aligned}
$$

A second model MOD_{DT} for the same signature is given by the set $\{0, 1, 2\}$ and operations defined modulo 3.

$$
\begin{aligned}
&MOD_{nat} = \{0, 1, 2\} \\
&zero_{MOD} = 0 \\
&succ_{MOD}(n) = (n + 1) \bmod 3 \\
&add_{MOD}(n, m) = (n + m) \bmod 3 \\
&sub_{MOD}(n, m) = (n - m) \bmod 3
\end{aligned}
$$

Now there is a data type homomorphism $mod_{DT}\colon NAT_{DT} \to MOD_{DT}$ between the two models.

$$mod_{nat}(n) = n \bmod 3$$

Note that mod_{nat} is "defining" for all applications of sub that are undefined in NAT, because sub_{MOD} is a total function.

2.1.2 Data State Layer

On the data state layer the dynamic states of the system are described. On the syntactical side there is a data state signature which is an extension of a data type signature. On the semantical side this is interpreted by a class of partial algebras, which are extensions of a data type model.

The additional sorts and operations of the data state signature will be called *dynamic* as opposed to the *static* sorts and operations of the data type signature.

Definition 2.1.6 (Data State Signature)
Given a data type signature Σ_{DT}, a *data state signature* is an algebraic signature Σ_{DS} extending Σ_{DT}.

$$\begin{array}{l} \Sigma_{DS} \in \mathbf{AlgSig} \\ \text{with} \quad \Sigma_{DT} \subseteq \Sigma_{DS} \end{array}$$

We will write data state signatures as follows, where we ommit the parts already present in the data type signature:

$$\begin{array}{lll} \text{data state:} & & \\ \text{sorts:} & s_1, \ldots & (s_1 \in \Sigma_{DT,S} \setminus \Sigma_{DT,S}) \\ \text{opns:} & \mathit{op}\colon \mathit{dom}(\mathit{op}) \to \mathit{cod}(\mathit{op}) & (\mathit{op} \in \Sigma_{DT,Op} \setminus \Sigma_{DT,Op}) \end{array}$$

Definition 2.1.7 (Data State Model)
Given a data type model M_{DT} and a data state signature Σ_{DS}, a *data state model* is a class M_{DS} of *data states*, which are partial Σ_{DS}-algebras extending M_{DT}.

$$M_{DS} \subseteq \{\, A \in PAlg(\Sigma_{DS}) \mid A|_{\Sigma_{DT}} = M_{DT} \,\}$$

A homomorphism between data state models is a function between the classes of data states. This function is uniquely determined by the following conditions:

- The dynamic sorts are identical in a data state and its image.
- The dynamic operations are defined homomorphic, i. e. such that the data type homomorphism together with identities for dynamic sorts, is a partial algebra homomorphism between the data states.
- Operation domains for dynamic operations are preserved by the homomorphism, which means that operations are not allowed to be defined for the images of tuples, for which the source operation is not defined, and for additional elements in the data type sorts.

These conditions are necessary for the restriction construction in Section 2.2, where we need to ensure that for data states identified by the restriction the images under an homomorphism are also identified.

Data states can be identified by a data state homomorphism if they only differ in operation results identified by the data type homomorphism. There can be additional data states in the target model that are not in the codomain of the function.

Definition 2.1.8 (Data State Model Homomorphism)
Given a data type homomorphism $h_{DT}\colon M_{DS} \to N_{DS}$, and two data state models M_{DS} and N_{DS} w. r. t. M_{DT} and N_{DT}, a *data state model homomorphism* is a function $h_{DS}\colon M_{DS} \to N_{DS}$ assigning a data state in N_{DS} to each data state in M_{DS}, such that all dynamic sorts are identical and all dynamic operations are compatible with the data state homomorphisms $h_A\colon A \to h_{DS}(A)$, which

are extensions of h_{DT} with identities for the dynamic sorts. Additionally the domains of operations have to be preserved by the state homomorphisms.

$$\begin{array}{l}
h_{DS}: M_{DS} \to N_{DS} \\
\text{with} \quad \forall s \in \Sigma_{DS,S} \setminus \Sigma_{DT,S}: \\
\qquad h_{DS}(A)_s = A_s \\
\qquad h_A: A \to h_{DS}(A) = (h_{DT}, (id_{A_s})_{s \in \Sigma_{DS,S} \setminus \Sigma_{DT,S}}) \text{ part. alg. hom.} \\
\qquad \forall op \in \Sigma_{DS,Op} \setminus \Sigma_{DT,Op}: \\
\qquad\quad h_{A,dom(op)}(Dom(op_A)) = Dom(op_{h_{DS}(A)})
\end{array}$$

Since data state homomorphisms are functions, composition and identities are just composition and indentities for functions.

Definition and Fact 2.1.9 (Data State Homomorph. Composition)
Given two data state homomorphisms $g_{DS}: L_{DS} \to M_{DS}$ and $h_{DS}: M_{DS} \to N_{DS}$, the *composition* $h_{DS} \circ g_{DS}: L_{DS} \to N_{DS}$ is a data state homomorphism. The composition of data state homomorphisms is associative, i. e. $(h_{DS} \circ g_{DS}) \circ f_{DS} = h_{DS} \circ (g_{DS} \circ f_{DS})$ for all $f_{DS}: K_{DS} \to L_{DS}$, $g_{DS}: L_{DS} \to M_{DS}$ and $h_{DS}: M_{DS} \to N_{DS}$.

Proof: We again have to show the homomorphism properties and associativity of composition.

Homomorphism Properties: The homomorphism properties of $h_{DS} \circ g_{DS}$, that we have to show to be implied by those of h_{DS} and g_{DS}, are the identity of dynamic sorts, a homomorphism existing betweeen a data state and its image and the identity of the domains of dynamic operations.

- We first show that the dynamic sorts are identical:

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall s \in \Sigma_{DS,S} \setminus \Sigma_{DT,S}:} \\
 & h_{DS}(g_{DS}(A))_s \\
= & (h_{DS} \text{ data state homomorphism, Def. 2.1.8}) \\
 & g_{DS}(A)_s \\
= & (g_{DS} \text{ data state homomorphism, Def. 2.1.8})) \\
 & A_s
\end{array}$$

- The algebra homomorphisms $(h \circ g)_A: A \to h_{DS}(g_{DS}(A))$ between the data states of L_{DS} and N_{DS} can becomposed from the homomorphisms $g_A: A \to g_{DS}(A)$ for g_{DS} and $h_{g_{DS}(A)}: g_{DS}(A) \to h_{DS}(g_{DS}(A))$ for h_{DS} and are therefore proper homomorphisms.
- The last property to show is the identity of operation domains:

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall op \in \Sigma_{DS,Op} \setminus \Sigma_{DT,Op}:} \\
 & h_{g_{DS}(A),dom(op)}(g_{A,dom(op)}(Dom(op_A))) \\
= & (g_{DS} \text{ data state homomorphism, Def. 2.1.8}) \\
 & h_{g_{DS}(A),dom(op)}(Dom(op_{g_{DS}(A)})) \\
= & (h_{DS} \text{ data state homomorphism, Def. 2.1.8}) \\
 & Dom(op_{h_{DS}(g_{DS}(A))})
\end{array}$$

Associativity: The associativity of composition follows directly from the associativity of function composition, since data state homomorphisms are functions.

Hence the composition of data state homomorphisms leads to data state homomorphisms and is associative. □

Definition and Fact 2.1.10 (Data State Identities)
Given a data state model M_{DS}, the *identity* $id_{M_{DS}}: M_{DS} \to M_{DS}$ is a data state homomorphism. The identities of data state models are neutral w.r.t. composition, i.e. $h_{DS} \circ id_{M_{DS}} = h_{DS} = id_{N_{DS}} \circ h_{DS}$ for all $h_{DS}: M_{DS} \to N_{DS}$.

Proof: The identity is a proper homomorphism, because the dynamic sorts are identical by definition, identities are partial algebra homomorphisms and undefinedness is obviously preserved by identities.

The neutrality of identities follows directly from the neutrality of identities on classes. □

Example: We continue the example by adding a dynamic attribute *att* of sort *nat* to get the data state signature Nat_{DS}.

$Nat_{DS} = Nat_{DT}+$
data type:
opns: $att: nat$

In the model NAT_{DS} we get a data state for each natural number.

$$NAT_{DS} \quad = \{\, A_i = (NAT_{DT}, i) \,|\, i \in \mathbb{N} \,\}$$

Note that the data state, where *att* is undefined is not included in NAT_{DS}.

In the model MOD_{DS} we just have three data states, because MOD_{nat} only contains three values.

$$MOD_{DS} \quad = \{\, A'_i = (MOD_{DT}, i) \,|\, i \in \{0, 1, 2\} \,\}$$

Again we did not include the fourth possibility that *att* is undefined.

Now we have a data state homomorphism mod_{DS} between these state classes.

$$mod_{DS}(NAT_{DT}, n) = (MOD_{DT}, n \text{ mod } 3)$$

2.1.3 Transformation Layer

On this layer transformations between data states are introduced. There is a transformation signature consisting of transformation symbols with input and output parameters in the syntax.

Definition 2.1.11 (Transformation Signature)
Given a data state signature Σ_{DS}, a *transformation signature* is a triple (Σ_{Tr}, in, out) with a set Σ_{Tr} of *transformation symbols* and families $in(t)$ and $out(t)$ of

input and *output parameter variables*, typed by sorts of Σ_{DS}, for each transformation symbol t.

$$\begin{aligned}&\Sigma_{Tr} \in \mathbf{Set}\\ &in\colon \Sigma_{Tr} \to \mathbf{Set}^{\Sigma_{DS,S}}\\ &out\colon \Sigma_{Tr} \to \mathbf{Set}^{\Sigma_{DS,S}}\end{aligned}$$

We will write transformation signatures as follows:

$$\text{trafos:}\quad t(x_1\colon s_1,\ldots;x_1'\colon s_1',\ldots)\quad (t \in \Sigma_{Tr}, x_1 \in in(t)_{s_1}, x_1' \in out(t)_{s_1'})$$

A transformation defines which data states can be transformed by giving a set of transformation steps, which contain the source and target data state and assignments from the input parameters to the source data state and from the output parameters to the target data state.

Definition 2.1.12 (Transformation)
Given a data state model M_{DS} and a transformation symbol t with parameters $in(t)$ and $out(t)$, a *transformation* t_M is a subclass of the class $TrExpr(M_{DS},t)$ of all possible *transformation expressions* $t_{A,B}(a,b)$, where A is a data state that is transformed into B with parameter assignments a and b.

$$\begin{aligned}&t_M \subseteq TrExpr(M_{DS},t)\\ &\text{where}\quad TrExpr(M_{DS},t) = \{\quad t_{A,B}(a,b) \mid A,B \in M_{DS},\\ &\qquad\qquad\qquad\qquad\qquad\quad A \in M_{DS}, a\colon in(t) \to A,\\ &\qquad\qquad\qquad\qquad\qquad\quad B \in M_{DS}, b\colon out(t) \to B \quad \}\end{aligned}$$

These transformations allow different interpretations. If a transformation is meant to specify changes to a only a part of the data state it can do this either tight, keeping all other elements of the data state the same, or loosely, allowing arbitrary changes to the other parts. We do not enforce one of these choices, but when lifting transformations during amalgamation in Subsection 3.1.3 we apply a loose interpretation.

The transformation model is a family with a transformation for each transformation symbol.

Definition 2.1.13 (Transformation Model)
Given a transformation signature (Σ_{Tr}, in, out), a *transformation model* M_{Tr} contains a transformation t_M for each transformation symbol.

$$M_{Tr} = (t_M)_{t\in\Sigma_{Tr}}$$

We wish to allow transformation steps to interleave, if they are between the same data states. For this reason we introduce a second model, which consists of sets of possible transformation steps between two data states. As a special case the empty set is a possible transformation set between all data states, meaning that all changes occur due to the environment.

If transformations are specified tightly, they can only interleave, if they agree on some transformation steps. For example a transformation *sqr* squaring an attribute $att\colon \rightarrow nat$ could interleave with a transformation dbl doubling att on the step from $att = 2$ to $att = 4$ to become a transformation set $T_{2,4} = \{sqr_{2,4}, dbl_{2,4}\}$.

If transformations are specified loosely, they can interleave between all data states where both change a part of the system. We have no means of demanding these parts to be disjoint, because the parts supposed to be changed by the environment are not explicitly distinguished from the parts the transformation is willing to change itself, but can only indirectly be concluded from the observation, whether all possible changes are included in the transformation or only some are selected. However, interleaving transformation steps have to agree on all changes, hence also on the part both want to change.

Definition 2.1.14 (Transformation Set Model)
Given a transformation model M_{Tr}, the *transformation set model* M_{TS} is the class of all sets of transformation steps, which agree on the data states, that are transformed.

$$\begin{array}{l} M_{TS} = \{\, T_{A,B} \mid A, B \in M_{DS}, T_{A,B} \subseteq Trans(A,B)) \,\} \\ \text{where} \quad Trans(A,B) = \{\, t_{X,Y}(x,y) \in \bigcup M_{Tr} \mid X = A, Y = B \,\} \end{array}$$

We can interpret the data states and transformation sets as nodes and edges of a (large) graph, called data state transformation system. This graph represents the whole (data) state space of the modelled system. It will be used as a target for labelling process scenarios on the next layer.

Definition 2.1.15 (Data State Transformation System)
Given a data state model M_{DS} and a transformation set model M_{TS}, the *data state transformation system* (short DSTS) M_{DSTS} is a graph with the data states M_{DS} as nodes and the transformation sets M_{TS} as edges, where each transformation set $T_{A,B}$ is an edge between A and B.

$$\begin{array}{l} M_{DSTS} = (M_{DS}, M_{TS}, src, trg) \in \mathbf{Graph} \\ \text{with} \quad src(T_{A,B}) = A \text{ and } trg(T_{A,B}) = B \end{array}$$

A transformation model homomorphism is, if it exists, uniquely determined by the underlying data state homomorphism. It assigns to every transformation step a transformation step between the images of the data states with the images of the parameters.

Again transformation steps are identified, if they only differ in values identified by the data type homomorphisms. and the target model can contain additional transformation steps.

Definition 2.1.16 (Transformation Model Homomorphism)
Given a data state homomorphism $h_{DS}\colon M_{DS} \rightarrow N_{DS}$ and two transformation models M_{Tr} and N_{Tr} w.r.t. M_{DS} and N_{DS}, a *transformation model homomorphism* $h_{Tr}\colon M_{Tr} \rightarrow N_{Tr}$ is a family of functions $h_t\colon t_M \rightarrow t_N$ between the transformations in M_{Tr} and the transformations in N_{Tr} which is compatible with h_{DS} (see Fig. 2.2).

$$\begin{array}{l} h_{Tr}\colon M_{Tr} \rightarrow N_{Tr} = (h_t\colon t_M \rightarrow t_N)_{t \in \Sigma_{Tr}} \\ \text{with} \quad h_t(t_{A,B}(a,b)) = t_{h_{DS}(A),h_{DS}(B)}(h_A \circ a, h_B \circ b) \end{array}$$

$$
\begin{tikzcd}
in(t) \arrow[r, "a"] \arrow[dr, "a'"'] \arrow[dr, phantom, "(=)", near start] & A \arrow[rr, Rightarrow, "t_{A,B}(a,b)"'] \arrow[d, shift right, "h_A"'] \arrow[d, shift left, "h_{DS}"] & {} \arrow[d, phantom, "\downarrow h_t"] & B \arrow[d, shift right, "h_{DS}"'] \arrow[d, shift left, "h_B"] & out(t) \arrow[l, "b"'] \arrow[dl, "b'"] \arrow[dl, phantom, "(=)", near start] \\
& A' \arrow[rr, Rightarrow, "t_{A',B'}(a',b')"] & {} & B' &
\end{tikzcd}
$$

Figure 2.2: Data state and transformation homomorphisms

The composition of transformation homomorphisms and the transformation model identities are again based on the composition of functions and class identities.

Definition and Fact 2.1.17 (Transformation Hom. Composition)
Given two transformation homomorphisms $g_{Tr}: L_{Tr} \to M_{Tr}$ and $h_{Tr}: M_{Tr} \to N_{Tr}$, the *composition* $h_{Tr} \circ g_{Tr}: L_{Tr} \to N_{Tr}$ is a transformation homomorphism defined by the composition of the functions for each transformation symbol.

$$h_{Tr} \circ g_{Tr} = ((h \circ g)_t)_{t \in \Sigma_{Tr}} = (h_t \circ g_t)_{t \in \Sigma_{Tr}}$$

The composition of transformation homomorphisms is associative, i. e. $(h_{Tr} \circ g_{Tr}) \circ f_{Tr} = h_{Tr} \circ (g_{Tr} \circ f_{Tr})$ for all $f_{Tr}: K_{Tr} \to L_{Tr}$, $g_{Tr}: L_{Tr} \to M_{Tr}$ and $h_{Tr}: M_{Tr} \to N_{Tr}$.

Proof: Since there are no additional porperties, that need to be satisfied for transformation homomorphisms (they are uniquely determined by the underlying data state homomorphism), the composition of transformation homomorphisms is obviously a transformation homomorphism.

Associativity of the composition is again implied directly by associativity of function and function family composition. □

Definition and Fact 2.1.18 (Transformation Identities)
Given a transformation model M_{Tr}, the *identity* $id_{M_{Tr}}: M_{Tr} \to M_{Tr}$ is a transformation homomorphism defined by identities for all transformation symbols.

$$id_{M_{Tr}} = (id_t)_{t \in \Sigma_{Tr}} = (id_{t_M})_{t \in \Sigma_{Tr}}$$

The identities of transformation models are neutral w. r. t. composition, i. e. $h_{Tr} \circ id_{M_{Tr}} = h_{Tr} = id_{N_{Tr}} \circ h_{Tr}$ for all $h_{Tr}: M_{Tr} \to N_{Tr}$.

Proof: As for the composition, there is nothing to show, here, since transformation homomorphisms do not have to satisfy any additional properties.

The neutrality is, once more, implied directly by the neutrality of identities for sets and families of sets. □

For the intepretation of the data state and transformation set models as grpahs, we get a graph homomorphism induced by a data state homomorphism and the corresponding transformation homomorphism.

Definition 2.1.19 (DSTS Homomorphism)
Given a data state homomorphism $h_{DS}: M_{DS} \to N_{DS}$, a transformation model homomorphism $h_{Tr}: M_{Tr} \to N_{Tr}$ and the two corresponding DSTSs M_{DSTS} and N_{DSTS}, the graph homomorphism $h_{DSTS}: M_{DSTS} \to N_{DSTS}$ is induced by h_{DS} and h_{Tr}.

$$\begin{array}{ll} \multicolumn{2}{l}{h_{DSTS}: M_{DSTS} \to N_{DSTS}} \\ \text{with} & h_{DSTS}(A) = h_{DS}(A) \\ & h_{DSTS}(T_{A,B}) = \{\, h_t(t_{A,B}(a,b)) \mid t_{A,B}(a,b) \in T_{A,B} \,\} \end{array}$$

Example: We enhance the example by a transformation signature with transformation symbols *inc* and *set*, where *inc* gets an input and an output parameter, which are supposed to be assigned to the original resp. the incremented value of the counter, and *set* just gets an output parameter, which is supposed to be assigned to the newly set value.

$$\begin{array}{ll} \text{trafos:} & inc(n\colon nat; m\colon nat) \\ & set(; m\colon nat) \end{array}$$

In NAT_{Tr} the transformation inc_{NAT} increments the counter and the transformation set_{NAT} sets it to the specified value.

$$\begin{array}{lll} inc_{NAT} & = \{\, inc_{A_i,A_j}(a,b) \mid & i,j \in \mathbb{N}, j = i+1, \\ & & a\colon \{n\} \to A_i, a(n) = i, \\ & & b\colon \{m\} \to A_j, b(m) = j \,\} \\ set_{NAT} & = \{\, set_{A_i,A_j}(\emptyset,b) \mid & i,j \in \mathbb{N}, \\ & & b\colon \{m\} \to A_j, b(m) = j \,\} \end{array}$$

In MOD_{Tr} the transformation inc_{MOD} becomes circular, while the transformation set_{MOD} can switch between all pairs of data states.

$$\begin{array}{lll} inc_{MOD} & = \{\, inc_{A'_i,A'_j}(a,b) \mid & i,j \in \{0,1,2\}, (i = 0 \wedge j = 1) \vee \\ & & (i = 1 \wedge j = 2) \vee (i = 2 \wedge j = 0), \\ & & a\colon \{n\} \to A'_i, a(n) = i, \\ & & b\colon \{m\} \to A'_j, b(m) = j \,\} \\ set_{MOD} & = \{\, set_{A'_i,A'_j}(\emptyset,b) \mid & i,j \in \{0,1,2\}, \\ & & b\colon \{m\} \to A'_j, b(m) = j \,\} \end{array}$$

Again we get a homomorphism mod_{Tr}, which assigns to a transformation step in NAT_{Tr} the corresponding transformation step modulo 3 in MOD_{Tr}. It consists of two functions mod_{inc} and mod_{set} for the transformation symbols.

$$\begin{array}{l} mod_{inc}(inc_{A_i,A_j}(a,b)) = inc_{A'_{i \bmod 3},A'_{j \bmod 3}}(a',b') \\ mod_{set}(set_{A_i,A_j}(\emptyset,b) = set_{A'_{i \bmod 3},A'_{j \bmod 3}}(\emptyset,b') \end{array}$$

$$
\begin{tikzcd}
\{n\} \arrow[r, "a"] \arrow[dr, "a'"'] \arrow[dr, phantom, bend left, "(=)"] & A_i \arrow[rr, Rightarrow, "inc_{A_i,A_j}(a,b)"'] \arrow[d, Rightarrow, "mod_{A_i}"', "mod_{DS}"] & {} \arrow[d, phantom, "\overset{\top}{mod_{inc}}\downarrow"] & A_j \arrow[d, Rightarrow, "mod_{DS}"', "mod_{A_j}"] & \{m\} \arrow[l, "b"'] \arrow[dl, "b'"] \arrow[dl, phantom, bend right, "(=)"'] \\
 & A'_{i \bmod 3} \arrow[rr, Rightarrow, "inc_{A'_{i \bmod 3},A'_{j \bmod 3}}(a',b')"] & {} & A'_{j \bmod 3} &
\end{tikzcd}
$$

2.1.4 Process Layer

On the process layer the reactive behaviour of the system is modelled in terms of processes.

Definition 2.1.20 (Process Signature)
Given a data state signature Σ_{DS}, a *process signature* is a triple (Σ_{Pr}, in, out) with a set of *process symbols* p and families $in(t)$ and $out(t)$ of *input* and *output parameter variables*, typed by sorts of Σ_{DS}.

$$
\begin{aligned}
&\Sigma_{Pr} \in \mathbf{Set}\\
&in\colon \Sigma_{Pr} \to \mathbf{Set}^{\Sigma_{DT,S}}\\
&out\colon \Sigma_{Pr} \to \mathbf{Set}^{\Sigma_{DT,S}}
\end{aligned}
$$

Similar to transformation signatures, we will write process signatures as follows:

$$\text{procs:}\quad p(x_1\colon s_1,\ldots; x'_1\colon s'_1,\ldots) \quad (p \in \Sigma_{Pr}, x_1 \in in(p)_{s_1}, x'_1 \in out(p)_{s'_1})$$

Processes in the model contain scenarios, called reactive state transition systems, which are graphs with initial and final states.

Definition 2.1.21 (Reactive State Transition System)
Given a process symbol p, a *reactive state transition system* (short RSTS) M_p is a graph G_{M_p} with initial nodes I_{M_p} and final nodes F_{M_p}.

$$
\begin{aligned}
&M_p = \big(G_{M_p}, I_{M_p}, F_{M_p}\big)\\
&\text{with}\quad G_{M_p} = \big(N_{M_p}, E_{M_p}, src, trg\big) \in \mathbf{Graph}\\
&\qquad\quad\ I_{M_p} \subseteq N_{M_p} F_{M_p} \subseteq N_{M_p}
\end{aligned}
$$

Similar to transformations processes are defined as subclasses of the class of all process expressions. A process expression captures the external view of a process occuring in the system, i. e. the initial and the final data state, between which the process occurs, together with possible parameters.

Definition 2.1.22 (Process)
Given a data state model M_{DS} and a process symbol p, a *process* p_M is a subclass of the class $PrExpr(M_{DS}, p)$ of all possible *process expressions* $p_{A,B}(a, b)$, where A is a data state that is transformed into B with parameter assignments a and b.

$$
\begin{aligned}
&p_M \subseteq PrExpr(M_{DS}, p)\\
&\text{where}\quad PrExpr(M_{DS}, p) = \{\quad p_{A,B}(a,b) \mid A, B \in M_{DS},\\
&\qquad\qquad\qquad\qquad\qquad\qquad A \in M_{DS}, a\colon in(p) \to A,\\
&\qquad\qquad\qquad\qquad\qquad\qquad B \in M_{DS}, b\colon out(p) \to B \quad \}
\end{aligned}
$$

Now, in the process model there are RSTSs and processes related by a function assigning to each process expression a set of occurences of the RSTS in the DSTS, such that a path from an initial to a final reactive state is mapped to a path from the initial to the final data state. This set is not allowed to be empty. Hence, for every process expression in the process, there is at least one occurrence.

Definition 2.1.23 (Process Model)
Given a data state transition system M_{DSTS} and a process signature (Σ_{Pr}, in, out), a *process model* M_{Pr} contains a RSTS M_p, a process p_M and a function Occ_{M_p} for each process symbol p. Occ_{M_p} assigns to each process expression a set of *occurences* of G_{M_p} in M_{DSTS}.

$$\begin{array}{ll} M_{Pr} = \left(M_p, p_M, Occ_{M_p}\right)_{p \in \Sigma_{Pr}} & \\ \text{with} \quad Occ_{M_p}: p_M \to \mathcal{P}(Mor(G_{M_p}, M_{DSTS})) & \\ \qquad\quad Occ_{M_p}(p_{A,B}(a,b)) \subseteq \{ & occ: G_{M_p} \to M_{DSTS} \mid \\ & \exists i \in I_{M_p} \exists f \in F_{M_p}: \\ & \quad occ(i) = A, occ(f) = B, i \xrightarrow{E^*_{M_p}} f \quad \} \\ \qquad\quad Occ_{M_p}(p_{A,B}(a,b)) \neq \emptyset & \end{array}$$

Homomorphisms between processes consist of graph homomorphisms between the RSTSs and functions between the processes, similar to transformation model homomorphisms. The graph homomorphisms have to preserve initial and final states. Moreover, they have to be compatible with the occurrences and the DSTS homomorphisms.

Reactive states can (but do not have to) be identified, if the corresponding data states are identified for all occurrences. Process expressions are similar to transformation steps identified, if and only if they only differ in values identified by h_{DT}. The target process can have additional reactive, initial and final states, process expressions and occurences.

Definition 2.1.24 (Process Model Homomorphism)
Given a data state homomorphism $h_{DS}: M_{DS} \to N_{DS}$, the corresponding DSTS homomorphism $h_{DSTS}: M_{DSTS} \to N_{DSTS}$ and two process models M_{Pr} and N_{Pr} w.r.t. M_{DSTS} and N_{DSTS}, a *process model homomorphism* $h_{Pr}: M_{Pr} \to N_{Pr}$ consists of a family of *RSTS homomorphisms* $h_{G_p}: M_{G_p} \to N_{G_p}$ and a family of *process expression mappings* $h_p: p_M \to p_N$. The RSTS homomorphisms have to preserve initial and final states. Moreover, for every occurence occ of a process expression in M there has to be an occurrence occ' of the translated process expression in N, which is compatible with h_{DSTS} and h_{G_p} (see Fig. 2.3).

$$\begin{array}{l} h_{Pr}: M_{Pr} \to N_{Pr} = \left(h_{G_p}: G_{M_p} \to G_{N_p}, h_p: p_M \to p_N\right)_{p \in \Sigma_{Pr}} \\ \text{with} \quad i \in I_{M_p} \implies h_{G_p}(i) \in I_{N_p} \\ \qquad\quad f \in F_{M_p} \implies h_{G_p}(f) \in F_{N_p} \\ \qquad\quad h_p(p_{A,B}(a,b)) = p_{h_{DS}(A), h_{DS}(B)}(h_A \circ a, h_B \circ b) \\ \qquad\qquad\quad \exists occ \in Occ_{M_p}(p_{A,B}(a,b)) \\ \qquad\quad \implies \exists occ' \in Occ_{N_p}(h_p(p_{A,B}(a,b))): \\ \qquad\qquad\qquad occ' \circ h_{G_p} = h_{DSTS} \circ occ \end{array}$$

$$
\begin{tikzcd}
G_{M_p} \arrow[r, "h_{G_p}"] \arrow[d, "\exists occ \in Occ_{M_p}(p_{A,B}(a,b))"'] \arrow[dr, phantom, "\Longrightarrow"] & G_{N_p} \arrow[d, "\exists occ' \in Occ_{N_p}(h_p(p_{A,B}(a,b)))"] \\
M_{DSTS} \arrow[r, "h_{DSTS}"'] & N_{DSTS}
\end{tikzcd}
$$

Figure 2.3: Process homomorphisms

We can now define composition of process model homomorphisms and process model identities by component-wise composition resp. identities for graphs and process expression classes.

Definition and Fact 2.1.25 (Process Homomorphism Composition) Given two process model homomorphisms $g_{Pr}: L_{Pr} \to M_{Pr}$ and $h_{Pr}: M_{Pr} \to N_{Pr}$, the *composition* $h_{Pr} \circ g_{Pr}: L_{Pr} \to N_{Pr}$ is a process model homomorphism defined by component-wise composition of the graph homomorphisms and process expression mappings.

$$h_{Pr} \circ g_{Pr} = \Big((h \circ g)_{G_p}, (h \circ g)_p\Big)_{p \in \Sigma_{Pr}} = \big(h_{G_p} \circ g_{G_p}, h_p \circ g_p\big)_{p \in \Sigma_{Pr}}$$

The composition of process model homomorphisms is associative, i.e. $(h_{Pr} \circ g_{Pr}) \circ f_{Pr} = h_{Pr} \circ (g_{Pr} \circ f_{Pr})$ for all $f_{Pr}: K_{Pr} \to L_{Pr}$, $g_{Pr}: L_{Pr} \to M_{Pr}$ and $h_{Pr}: M_{Pr} \to N_{Pr}$.

Proof: Once more, we have to show the homomorphism properties and the associativity.

Homomorphism Properties: The homomorphism properties of the composition consist of $h_{G_p} \circ g_{G_p}$ being a graph homomorphism, that preserves initial and final states, $h_p \circ g_p$ being a process expression mapping and the existence of compatible occurrences.

- $h_{G_p} \circ g_{G_p}$ is a graph homomorphism, because graph homomorphisms are closed under composition.
- $h_{G_p} \circ g_{G_p}$ preserves initial and final states, because h_{G_p} and g_{G_p} preserve initial and final states:

$$
\begin{array}{ll}
 & i \in I_{L_p} \quad \text{resp.} \quad f \in F_{L_p} \\
\Longrightarrow & (g_{Pr} \text{ process model homomorphism, Def. 2.1.24}) \\
 & g_{G_p}(i) \in I_{M_p} \quad \text{resp.} \quad g_{G_p}(f) \in F_{M_p} \\
\Longrightarrow & (h_{Pr} \text{ process model homomorphism, Def. 2.1.24}) \\
 & h_{G_p} \circ g_{G_p}(i) \in I_{M_p} \quad \text{resp.} \quad h_{G_p} \circ g_{G_p}(f) \in F_{M_p}
\end{array}
$$

- $h_p \circ g_p$ is a process expression mapping analogously to the composition of transformation expression mappings.

- The existence of compatible occurences is achieved by transitivity of the implication (see Fig. 2.4):

$$
\begin{array}{ll}
& \exists occ \in Occ_{L_p}(p_{A,B}(a,b)) \\
\Longrightarrow & (g_{Pr} \text{ process model homomorphism, Def. 2.1.24}) \\
& \exists occ' \in Occ_{M_p}(g_p(p_{A,B}(a,b))): \\
& \qquad occ' \circ g_{G_p} = g_{DSTS} \circ occ \\
\Longrightarrow & (h_{Pr} \text{ process model homomorphism, Def. 2.1.24}) \\
& \exists occ' \in Occ_{M_p}(g_p(p_{A,B}(a,b))) \\
& \exists occ'' \in Occ_{N_p}(h_p \circ g_p(p_{A,B}(a,b))): \\
& \qquad occ' \circ g_{G_p} = g_{DSTS} \circ occ \qquad (1) \\
& \wedge \quad occ'' \circ h_{G_p} = h_{DSTS} \circ occ' \quad (2) \\
\Longrightarrow & \exists occ' \in Occ_{M_p}(g_p(p_{A,B}(a,b))) \\
& \exists occ'' \in Occ_{N_p}(h_p \circ g_p(p_{A,B}(a,b))): \\
& \qquad occ'' \circ h_{G_p} \circ g_{G_p} \\
& \overset{(2)}{=} \quad h_{DSTS} \circ occ' \circ g_{G_p} \\
& \overset{(1)}{=} \quad h_{DSTS} \circ g_{DSTS} \circ occ \\
\Longrightarrow & \exists occ'' \in Occ_{N_p}(h_p \circ g_p(p_{A,B}(a,b))): \\
& \qquad occ'' \circ h_{G_p} \circ g_{G_p} = h_{DSTS} \circ g_{DSTS} \circ occ
\end{array}
$$

Associativity: The associativity follows from the associativity of composition for graph homomorphisms, functions and families of them.

Hence the composition leads to proper process homomorphisms and is associative. $\square$

$$
\begin{tikzcd}
G_{L_p} \arrow[r, "g_{G_p}"] \arrow[d, "occ"'] \arrow[dr, phantom, "(1)"] & G_{M_p} \arrow[r, "h_{G_p}"] \arrow[d, "occ'"] \arrow[dr, phantom, "(2)"] & G_{N_p} \arrow[d, "occ''"] \\
L_{DSTS} \arrow[r, "g_{DSTS}"'] & M_{DSTS} \arrow[r, "h_{DSTS}"'] & N_{DSTS}
\end{tikzcd}
$$

Figure 2.4: Compoitionality of occurence existence

Definition and Fact 2.1.26 (Process Identities)
Given a process model M_{Pr}, the *identity homomorphism* $id_{M_{Pr}}: M_{Pr} \to M_{Pr}$ is a process homomorphism defined by identities on all graphs and processes.

$$id_{M_{Pr}} = \left(id_{G_p}, id_p\right)_{p \in \Sigma_{Pr}} = \left(id_{G_{M_p}}, id_{p_M}\right)_{p \in \Sigma_{Pr}}$$

The identities of process models are neutral w. r. t. composition, i. e. $h \circ id_M = h = id_N \circ h$ for all $h: M \to N$.

Proof: The homomorphism properties of the identity are again obvious: Identity is a graph homomorphism, compatibility for process expressions is achieved as for transformation symbols, initial and final states, as well as occurrences are identical.

Neutrality of identities follows from the neutrality of graph and class identities and families of them. $\square$

Example: For the example of natural numbers we introduce a process symbol *cnt* with an input and an output parameter.

procs: $cnt(n\colon nat; m\colon nat)$

This process shall be interpreted as counting up three times in NAT_{Pr}. Hence, the process cnt_{NAT} consists of process expressions, where the second algebra has a counter, which is the counter of the first one plus 3, and the parameters are assigned to the values of the counters.

$$\begin{aligned} cnt_{NAT} \quad = \{\, cnt_{A_i,A_j}(a,b) \mid \quad & i,j \in \mathbb{N}, j = i+3, \\ & a\colon \{n\} \to A_i, a(n) = i, \\ & b\colon \{m\} \to A_j, b(m) = j \,\} \end{aligned}$$

The RSTS NAT_{cnt} is given by the graph $G_{NAT_{cnt}}$ in Fig. 2.5 with the initial state $state_0$ and the final state $state_3$. In our model we choose to have exactly one occurrence for each expression in the process, namely the occurrence depicted in Fig. 2.5.

$$\begin{tikzcd}
G_{NAT_{cnt}} \arrow[rr, "occ \in Occ_{NAT_{cnt}}(cnt_{A_i,A_{i+3}}(\ldots))"] & & NAT_{DSTS} \\
state_0 \arrow[rr, mapsto] \arrow[d, "step_1"'] \arrow[drr, phantom, "\longmapsto"] & & A_i \arrow[d, "\{inc_{A_i,A_{i+1}}(\ldots)\}"] \\
state_1 \arrow[rr, mapsto] \arrow[d, "step_2"'] \arrow[drr, phantom, "\longmapsto"] & & A_{i+1} \arrow[d, "\{inc_{A_{i+1},A_{i+2}}(\ldots)\}"] \\
state_2 \arrow[rr, mapsto] \arrow[d, "step_3"'] \arrow[drr, phantom, "\longmapsto"] & & A_{i+2} \arrow[d, "\{inc_{A_{i+2},A_{i+3}}(\ldots)\}"] \\
state_3 \arrow[rr, mapsto] & & A_{i+3}
\end{tikzcd}$$

Figure 2.5: Process cnt_{NAT}

In MOD_{Pr} we also interpret the process as counting up three times, but, since our model is constructed modulo 3, this process contains a circle in the data state transformations. We choose to reflect this circle in the RSTS, leading to the graph $G_{MOD_{cnt}}$ in Fig. 2.6. Now, the initial and final state are both $state_0$. In the process cnt_{MOD} we have three process expressions.

$$\begin{aligned} cnt_{MOD} \quad = \{\, cnt_{A'_i,A'_i}(a,b) \mid \quad & i \in \{0,1,2\}, \\ & a\colon \{n\} \to A'_i, a(n) = i, \\ & b\colon \{m\} \to A'_i, b(m) = i \,\} \end{aligned}$$

We take just the one occurrence depicted in Fig. 2.6 for each process expression.

Between these processes there is a process homomorphism consisting of the graph homomorphism $mod_{G_{cnt}}$ in Fig. 2.7 and the process expression mapping

$$G_{MOD_{cnt}} \xrightarrow{occ \in Occ_{MOD_{cnt}}(cnt_{A'_i, A'_i}(...))} MOD_{DSTS}$$

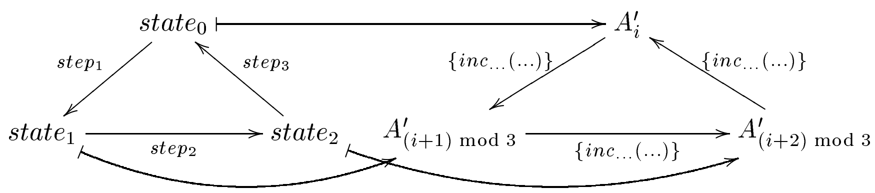

Figure 2.6: Process cnt_{MOD}

mod_{cnt}. It identifies $state_0$ and $state_3$ of $G_{NAT_{cnt}}$ to become $state_0$ in $G_{MOD_{cnt}}$. The compatibility with mod_{DS} is achieved, because A_i and A_{i+3} are identified to become $A'_{i \bmod 3}$ by mod_{DS}.

$$G_{NAT_{cnt}} \xrightarrow{mod_{G_{cnt}}} G_{MOD_{cnt}}$$

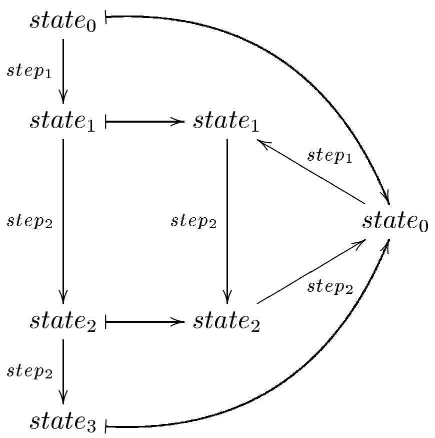

Figure 2.7: Process homomorphism mod_{cnt}

2.1.5 Integration

We now combine the notions of the different layers into integrated definitions.

Integrated signatures combine the syntactical sides of the layers.

Definition 2.1.27 (Integrated Signature)
An *integrated signature* Σ consists of

- a data type signature Σ_{DT} as in Def. 2.1.1,
- a data state signature Σ_{DS} w.r.t. Σ_{DT} as in Def. 2.1.6,

- a transformation signature (Σ_{Tr}, in, out) w.r.t. Σ_{DS} as in Def. 2.1.11 and
- a process signature (Σ_{Pr}, in, out) w.r.t. Σ_{DS} as in Def. 2.1.20.

Integrated models combine the semantical sides, where we take only those parts not induced by other ones, which is why we did not include M_{TS} and M_{DSTS}, which can be derived from M_{Tr} and M_{DS}.

Definition 2.1.28 (Integrated Model)
An *integrated model* M of an integrated signature Σ consists of

- a data type model M_{DT} as in Def. 2.1.2,
- a data state model M_{DS} w.r.t. M_{DT} as in Def. 2.1.7,
- a transformation model M_{Tr} w.r.t. M_{DS} as in Def. 2.1.13 and
- a process model M_{Pr} w.r.t. M_{DS} and M_{Tr} as in Def. 2.1.23.

Integrated model homomorphisms, as representations of the structural relations between models, combine the homomorphism notions. We include h_{DS} and h_{Tr}, although they are uniquely determined by h_{DT}, because their existence is not guaranteed, but is needed for an integrated model homomorphism. The DSTS homomorphism h_{DSTS} is not included, because it is induced by h_{DS} and h_{Tr}.

Definition 2.1.29 (Integrated Model Homomorphism)
An *integrated model homomorphism* $h: M \to N$ for an integrated signature Σ and integrated models M and N of Σ consists of

- a data type model homomorphism h_{DT} as in Def. 2.1.3,
- a data state model homomorphism h_{DS} w.r.t. h_{DT} as in Def. 2.1.8,
- a transformation model homomorphism h_{Tr} w.r.t. h_{DS} as in Def. 2.1.16 and
- a process model homomorphism h_{Pr} w.r.t. h_{DS} and h_{Tr} as in Def. 2.1.24.

In order to get a model category we also combine the definitions of associative compositions and neutral identities to become integrated notions. The needed facts are obviously implied by the corresponding facts on the layers.

Definition and Fact 2.1.30 (Integrated Homomorphism Composition)
Given two integrated model homomorphisms $g: L \to M$ and $h: M \to N$ for an integrated signature Σ, the *composition* $h \circ g: L \to N$ is an integrated model homomorphism defined by

- $(h \circ g)_{DT} = h_{DT} \circ g_{DT}$ as in Def. 2.1.4,
- $(h \circ g)_{DS} = h_{DS} \circ g_{DS}$ as in Def. 2.1.9,
- $(h \circ g)_{Tr} = h_{Tr} \circ g_{Tr}$ as in Def. 2.1.17 and

- $(h \circ g)_{Pr} = h_{Pr} \circ g_{Pr}$ as in Def. 2.1.25.

The composition of integrated model homomorphisms is associative, i.e. $(h \circ g) \circ f = h \circ (g \circ f)$ for all $f: K \to L$, $g: L \to M$ and $h: M \to N$.

Proof: The associativity of composition follows from the component-wise associativity of the different layers. $\square$

Definition and Fact 2.1.31 (Integrated Identities)
Given an integrated model M for an integrated signature Σ, the *identity homomorphism* $id_M: M \to M$ is an integrated model homomorphism defined by

- $(id_M)_{DT} = id_{M_{DT}}$ as in Def. 2.1.5,
- $(id_M)_{DS} = id_{M_{DS}}$ as in Def. 2.1.10,
- $(id_M)_{Tr} = id_{M_{Tr}}$ as in Def. 2.1.18 and
- $(id_M)_{Pr} = id_{M_{Pr}}$ as in Def. 2.1.26.

The identities of integrated models are neutral w.r.t. composition, i.e. $h \circ id_M = h = id_N \circ h$ for all $h: M \to N$.

Proof: The neutrality of identities follows from the compoent-wise neutrality on the different layers. $\square$

The following theorem combines the facts for compositionality, associativity of compositions and neutrality of identities into the notion of category, and is therefore a main result of this section.

Theorem 2.1.32 (Integrated Model Category)
The integrated models for an integrated signature Σ with integrated model homomorphisms, composition and identities form a category, denoted by $Mod(\Sigma)$.

Proof: There are two category axioms that need to hold:

Existence and Associativity of Composition: See Def. and Fact 2.1.30

Existence and Neutrality of Identities: See Def. and Fact 2.1.31

$\square$

2.2 Signature Morphisms and Restriction Construction

Signature morphisms capture the structural relation between different signatures. They express how the elements of "smaller" signatures can be found in "larger" ones.

This is interpreted by a restriction construction on models in the opposite direction that forgets the parts which are only contained in the second signature. In the general case different elements of the first signature can be assigned to the same element of the second one, leading to a duplication of the corresponding model parts in the restriction.

The restriction can also be applied to homomorphisms, where it is compatible with composition and identities leading to a functor.

As in the previous section, we first look at the layers in detail in Subsections 2.2.1 through 2.2.4 and then combime them into integrated notions in Subsection 2.2.5.

2.2.1 Data Type Layer

On the data type layer we have an algebraic signature morphism on the syntactical side. It maps sort and operation symbols, possibly identifying some symbols, where the maps have to be compatible with the domains and codomains.

Definition 2.2.1 (Data Type Signature Morphism)
Given two data type signatures Σ_{DT} and Σ'_{DT}, a *data type signature morphism* is an algebraic signature morphism $f_{DT}: \Sigma_{DT} \to \Sigma'_{DT}$, which contains a mapping $f_{DT,S}: \Sigma_{DT,S} \to \Sigma'_{DT,S}$ of sorts and a mapping $f_{DT,Op}: \Sigma_{DT,Op} \to \Sigma'_{DT,Op}$ of operation symbols, such that the images of domain and codomain for each operation symbol *op* are the domain and codomain of the image of *op* (see Fig. 2.8(a) and Fig. 2.8(b)).

$$\begin{aligned}
&f_{DT}: \Sigma_{DT} \to \Sigma'_{DT} = (f_{DT,S}, f_{DT,Op})\\
&\text{with}\quad f_{DT,S}: \Sigma_{DT,S} \to \Sigma'_{DT,S}\\
&\qquad\quad f_{DT,Op}: \Sigma_{DT,Op} \to \Sigma'_{DT,Op}\\
&\qquad\quad dom \circ f_{DT,Op} = (f_{DT,S})^* \circ dom\\
&\qquad\quad cod \circ f_{DT,Op} = (f_{DT,S})^* \circ cod
\end{aligned}$$

$$\begin{tikzcd}
\Sigma_{DT,Op} \arrow[r, "f_{DT,Op}"] \arrow[d, "dom"'] \arrow[dr, phantom, "(=)"] & \Sigma'_{DT,Op} \arrow[d, "dom"] \\
(\Sigma_{DT,S})^* \arrow[r, "(f_{DT,S})^*"'] & (\Sigma'_{DT,S})^*
\end{tikzcd}$$

(a) Operation domains

$$\begin{tikzcd}
\Sigma_{DT,Op} \arrow[r, "f_{DT,Op}"] \arrow[d, "cod"'] \arrow[dr, phantom, "(=)"] & \Sigma'_{DT,Op} \arrow[d, "cod"] \\
\Sigma_{DT,S} \cup \{\lambda\} \arrow[r, "(f_{DT,S})^*"'] & \Sigma'_{DT,S} \cup \{\lambda\}
\end{tikzcd}$$

(b) Operation codomains

Figure 2.8: Data type signature morphism

The data type restriction is the usual forgetful functor for algebras (see e. g. [EM85]) constructing a model of Σ_{DT} from a model of Σ'_{DT} by forgetting sorts and operations not in the codomain of f_{DT}, and duplicating and renaming the other ones according to f_{DT}. The restriction to a subsignature used in the definition of data state models (Def. 2.1.7) is a special case of this restriction. Since inclusions are injective and map symbols identically, nothing is renamed or duplicated here, only the additional parts are forgotten.

Definition and Fact 2.2.2 (Data Type Model Restriction)
Given a data type signature morphism $f_{DT}: \Sigma_{DT} \to \Sigma'_{DT}$ and a Σ'_{DT}-model M'_{DT}, the *restriction* of M'_{DT} is a Σ_{DT}-model $V_{f_{DT}}(M'_{DT}) = M_{DT}$, where each carrier set M_s resp. operation op_M is given by the carrier set $M'_{f_{DT,S}(s)}$ resp. operation $f_{DT,Op}(op)_{M'}$ for the image of the corresponding symbol in Σ'_{DT}.

$$
\begin{array}{ll}
V_{f_{DT}}(M'_{DT}) = M_{DT} & \\
\text{with} & M_s = M'_{f_{DT,S}(s)} \\
 & op_M = f_{DT,Op}(op)_{M'}
\end{array}
$$

Proof: We need to show that M_{DT} is a proper Σ_{DT}-model, i.e. that the functionalities of the operations are correct. Since op_M is defined as $f_{DT,Op}(op)_{M'}$, its functionality is $op_M: M'_{dom(f_{DT,Op}(op))} \to M'_{cod(f_{DT,Op}(op))}$. By domain and codomain compatibility of signature morphisms and the definition of restriction for sorts we get the correct functionality $op_M: M_{dom(op)} \to M_{cod(op)}$:

$$
\begin{array}{lccc}
 & M'_{dom(f_{DT,Op}(op))} & \to & M'_{cod(f_{DT,Op}(op))} \\
= & \multicolumn{3}{l}{\text{(compatibility for signature morphism, Def. 2.2.1)}} \\
 & M'_{(f_{DT,S})^*(dom(op))} & \to & M'_{(f_{DT,S})^*(cod(op))} \\
= & \multicolumn{3}{l}{\text{(restriction of sorts, Def. 2.2.2)}} \\
 & M_{dom(op)} & \to & M_{cod(op)}
\end{array}
$$

□

We can also apply the restriction construction to homomorphisms, where we get a restricted homomorphism between the restricted data types. In this case the restriction is compatible with composition and identities.

Definition and Fact 2.2.3 (Data Type Homomorphism Restriction)
Given a data type signature morphism $f_{DT}: \Sigma_{DT} \to \Sigma'_{DT}$ and a Σ'_{DT}-homomorphism $h'_{DT}: M'_{DT} \to N'_{DT}$, the *restriction* of h'_{DT} is a Σ_{DT}-homomorphism $V_{f_{DT}}(h'_{DT}) = h_{DT}: M_{DT} \to N_{DT}$ between the forgetful functor images $M_{DT} = V_{f_{DT}}(M'_{DT})$ and $N_{DT} = V_{f_{DT}}(N'_{DT})$, where the function $h_s: M_s \to N_s$ for each sort symbol $s \in \Sigma_{DT,S}$ is given by the function for the image sort symbol.

$$
\begin{array}{ll}
V_{f_{DT}}(h'_{DT}: M'_{DT} \to N'_{DT}) = h_{DT}: M_{DT} \to N_{DT} & \\
\text{with} & M_{DT} = V_{f_{DT}}(M'_{DT}), N_{DT} = V_{f_{DT}}(N'_{DT}) \\
 & h_s: M_s \to N_s = h'_{f_{DT,S}(s)}: M'_{f_{DT,S}(s)} \to N'_{f_{DT,S}(s)}
\end{array}
$$

The restriction is compatible with composition and identities:

$$
\begin{array}{ll}
 & V_{f_{DT}}(h'_{DT} \circ g'_{DT}) = V_{f_{DT}}(h'_{DT}) \circ V_{f_{DT}}(g'_{DT}) \\
\text{and} & V_{f_{DT}}(id_{M'_{DT}}) = id_{V_{f_{DT}}(M'_{DT})}
\end{array}
$$

Proof: We need to show the homomorphism properties of the restricted homomorphism and the compatibility with composition and identities.

Homomorphism Properties: The homomorphism properties, i.e. preservation of definedness and compatibility with operation applications, of h_{DT} must be implied by those of h'_{DT}.

- The preservation of definedness is obtained from the corresponding property of h'_{DT}:

$$\begin{array}{ll} & h_{dom(op)}(Dom(op_M)) \\ = & \text{(restr. of oper. and hom., Def. 2.2.2 and 2.2.3)} \\ & h'_{(f_{DT,S})^*(dom(op))}(Dom(f_{DT,Op}(op)_{M'})) \\ = & \text{(compatibility for signature morphism, Def. 2.2.1)} \\ & h'_{dom(f_{DT,Op}(op))}(Dom(f_{DT,Op}(op)_{M'})) \\ \subseteq & (h'_{DT} \text{ data type homomorphism, Def. 2.1.3)} \\ & Dom(f_{DT,Op}(op)_{N'}) \\ = & \text{(restriction of operations, Def. 2.2.2)} \\ & Dom(op_N) \end{array}$$

- The compatibility of operation applications is also implied by the corresponding compatibility for h'_{DT}:

$$\begin{array}{ll} & op_N \circ h_{dom(op)} \\ = & \text{(restr. of oper. and hom., Def. 2.2.2 and 2.2.3)} \\ & f_{DT,Op}(op)_{N'} \circ h'_{(f_{DT,S})^*(dom(op))} \\ = & \text{(compatibility for signature morphism, Def. 2.2.1)} \\ & f_{DT,Op}(op)_{N'} \circ h'_{dom(f_{DT,Op}(op))} \\ \simeq & (h'_{DT} \text{ hommmorphism, Def. 2.1.3)} \\ & h'_{cod(f_{DT,Op}(op))} \circ f_{DT,Op}(op)_{M'} \\ = & \text{(compatibility for signature morphism, Def. 2.2.1)} \\ & h'_{(f_{DT,S})^*(cod(op))} \circ f_{DT,Op}(op)_{M'} \\ = & \text{(restr. of oper. and hom., Def. 2.2.2 and 2.2.3)} \\ & h_{cod(op)} \circ op_M \end{array}$$

Compatibility with Composition: We can reduce this property to the composition for families of functions, which is well-known to be compatible with the component-wise composition of the included functions:

$$\begin{array}{ll} & V_{f_{DT}}(h'_{DT} \circ g'_{DT}) \\ = & \text{(restriction of homomorphisms, Def. 2.2.3)} \\ & ((h' \circ g')_{f_{DT,S}(s)})_{s \in \Sigma_{DT,S}} \\ = & \text{(composition of homomorphisms, Def. 2.1.4)} \\ & (h'_{f_{DT,S}(s)} \circ g'_{f_{DT,S}(s)})_{s \in \Sigma_{DT,S}} \\ = & \text{(composition of families of functions)} \\ & (h'_{f_{DT,S}(s)})_{s \in \Sigma_{DT,S}} \circ (g'_{f_{DT,S}(s)})_{s \in \Sigma_{DT,S}} \\ = & \text{(restriction of homomorphisms, Def. 2.2.3)} \\ & V_{f_{DT}}(h'_{DT}) \circ V_{f_{DT}}(g'_{DT}) \end{array}$$

Compatibility with Identities: This property is even easier to show, since it just needs the application of the corresponding definitions:

$$\begin{array}{ll} & V_{f_{DT}}(id_{M'_{DT}}) \\ = & \text{(restriction of homomorphisms, Def. 2.2.3)} \\ & (id_{M'_{f_{DT,S}(s)}})_{s \in \Sigma_{DT,S}} \\ = & \text{(restriction of sorts, Def. 2.2.2)} \\ & (id_{V_{f_{DT}}(M'_{DT})_s})_{s \in \Sigma_{DT,S}} \\ = & \text{(data type identity, Def. 2.1.5)} \\ & id_{V_{f_{DT}}(M'_{DT})} \end{array}$$

Hence the restriction of data type homomorphisms leads to proper data type homomorphisms and is compatible with composition and identities. □

2.2.2 Data State Layer

On the data state layer we have a second algebraic signature morphism which is compatible with the one on the data type layer. We additionally require the morphism to map dynamic elements of Σ_{DS} to dynamic elements of Σ'_{DT}.

Definition 2.2.4 (Data State Signature Morphism)
Given a data type signature morphism $f_{DT}\colon \Sigma_{DT} \to \Sigma'_{DT}$ and two data state signatures Σ_{DS} and Σ'_{DS}, a *data state signature morphism* is an alebraic signature morphism $f_{DS}\colon \Sigma_{DS} \to \Sigma'_{DS}$, which is an extension of f_{DT} (see Fig. 2.9) and maps dynamic sorts $s \in \Sigma_{DS,S} \setminus \Sigma_{DT,S}$ to dynamic sorts $f_{DS,S}(s) \in \Sigma'_{DS,S} \setminus \Sigma'_{DT,S}$ and dynamic operations $op \in \Sigma_{DS,Op} \setminus \Sigma_{DT,Op}$ to dynamic operations $f_{DS,S}(op) \in \Sigma'_{DS,Op} \setminus \Sigma'_{DT,Op}$.

$$
\begin{array}{ll}
f_{DS}\colon \Sigma_{DS} \to \Sigma'_{DS} & \\
\text{with} & f_{DT} \subseteq f_{DS} \\
& f_{DS,S}(\Sigma_{DS,S} \setminus \Sigma_{DT,S}) \subseteq \Sigma'_{DS,S} \setminus \Sigma'_{DT,S} \\
& f_{DS,S}(\Sigma_{DS,Op} \setminus \Sigma_{DT,Op}) \subseteq \Sigma'_{DS,Op} \setminus \Sigma'_{DT,Op}
\end{array}
$$

$$
\begin{tikzcd}
\Sigma_{DT} \arrow[r, "f_{DT}"] \arrow[d, hook] \arrow[dr, phantom, "(=)"] & \Sigma'_{DT} \arrow[d, hook] \\
\Sigma_{DS} \arrow[r, "f_{DS}"'] & \Sigma'_{DS}
\end{tikzcd}
$$

Figure 2.9: Data state signature morphism

The restriction of a data state model is achieved by applying the forgetful functor to all data states in the model. This construction identifies data states, whose differences are forgotten by the functor, because they only differed in elements of sorts or applications of operations, that have no correspondence in Σ_{DS}.

Definition and Fact 2.2.5 (Data State Model Restriction)
Given a data state signature morphism $f_{DS}\colon \Sigma_{DS} \to \Sigma'_{DS}$ and a Σ'_{DS}-model M'_{DS}, the *restriction* of M'_{DS} is a Σ_{DS}-model $Restr_{f_{DS}}(M'_{DS}) = M_{DS}$ consisting of all forgetful functor images for the data states in M'_{DS}.

$$
\begin{array}{ll}
Restr_{f_{DS}}(M'_{DS}) = M_{DS} & \\
\text{with} & M_{DS} = \{\, V_{f_{DS}}(A') \mid A' \in M'_{DS} \,\}
\end{array}
$$

Proof: We have to show, that M_{DS} is a Σ_{DS}-model w.r.t. the data type $V_{f_{DT}}(M'_{DT}) = M_{DT}$, i.e. that the restriction to the data type signature is

M_{DT} for all data states $A \in M_{DS}$:

$$\begin{array}{ll} & A|_{\Sigma_{DT}} \\ = & \text{(data state restriction, Def. 2.2.5)} \\ & V_{f_{DS}}(A')|_{\Sigma_{DT}} \text{ for some } A' \in M'_{DS} \\ = & \text{(compatibility for signature morphisms, Def. 2.2.4)} \\ & V_{f_{DT}}(A'|_{\Sigma'_{DT}}) \\ = & (M'_{DS} \text{ data state model, Def. 2.1.7)} \\ & V_{f_{DT}}(M'_{DT}) = M_{DT} \end{array}$$

□

In the restriction of data state homomorphisms we need the very strong requirements, we assumed for them in Section 2.1, because we have to ensure that for two data states of M'_{DS}, which are identified by the restriction, also their images under the homomorphism are identified.

Definition and Fact 2.2.6 (Data State Homomorphism Restriction) Given a data state signature morphism $f_{DS}: \Sigma_{DS} \to \Sigma'_{DS}$ and a Σ'_{DS}-homomorphism $h'_{DS}: M'_{DS} \to N'_{DS}$, the *restriction* of h'_{DS} is a Σ_{DS}-homomorphism $Restr_{f_{DS}}(h'_{DS}) = h_{DS}: M_{DS} \to N_{DS}$ between $M_{DS} = Restr_{f_{DS}}(M'_{DS})$ and $N_{DS} = Restr_{f_{DS}}(N'_{DS})$ assigning to the forgetful functor image $V_{f_{DS}}(A')$ of a data state A' the forgetful functor image of $h'_{DS}(A')$.

$$\begin{array}{l} Restr_{f_{DS}}(h'_{DS}: M'_{DS} \to N'_{DS}) = h_{DS}: M_{DS} \to N_{DS} \\ \text{with} \quad M_{DS} = Restr_{f_{DS}}(M'_{DS}), N_{DS} = Restr_{f_{DS}}(N'_{DS}) \\ \qquad\quad h_{DS}(V_{f_{DS}}(A')) = V_{f_{DS}}(h'_{DS}(A')) \end{array}$$

The restriction is compatible with composition and identities:

$$\begin{array}{ll} & Restr_{f_{DS}}(h'_{DS} \circ g'_{DS}) = Restr_{f_{DS}}(h'_{DS}) \circ Restr_{f_{DS}}(g'_{DS}) \\ \text{and} & Restr_{f_{DS}}(id_{M'_{DS}}) = id_{Restr_{f_{DS}}(M'_{DS})} \end{array}$$

Proof: We have to show the well-definedness of the restriction, the homomorphism properties and that compositions and identities are preserved.

Well-Definedness: Since the restriction is defined by referring to not necessarily unique data states of M'_{DS}, whose forgetful functor image the data state of M_{DS} is, we need to show, that in cases data states are identified by the restriction, their images under the homomorphism are also identified:

$$V_{f_{DS}}(A') = V_{f_{DS}}(B') \implies V_{f_{DS}}(h'_{DS}(A')) = V_{f_{DS}}(h'_{DS}(B'))$$

- First we show this for the sorts, where we need the requirement, that the dynamic sorts are identical in states related by a homomorphism.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall s \in \Sigma_{DS,S} \setminus \Sigma_{DT,S}:} \\
 & V_{f_{DS}}(h'_{DS}(A'))_s \\
= & \text{(restriction of sorts, Def. 2.2.2)} \\
 & h'_{DS}(A')_{f_{DS,S}(s)} \\
= & (h'_{DS} \text{ data state model homomorphism, 2.1.8)} \\
 & A'_{f_{DS,S}(s)} = V_{f_{DS}}(A')_s \\
= & \text{(assumption } V_{f_{DS}}(A') = V_{f_{DS}}(B')) \\
 & V_{f_{DS}}(B')_s = B'_{f_{DS,S}(s)} \\
= & (h'_{DS} \text{ data state model homomorphism, 2.1.8)} \\
 & h'_{DS}(B')_{f_{DS,S}(s)} \\
= & \text{(restriction of sorts, Def. 2.2.2)} \\
 & V_{f_{DS}}(h'_{DS}(B'))_s
\end{array}
$$

- Now, for the operations, we first show the equality of $op'_{h'_{DS}(A)} \circ h'_{A',dom(op)}$ and $op'_{h'_{DS}(B)} \circ h'_{B',dom(op)}$, for which we need the homomorphism properties (see Fig. 2.10).

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall op \in \Sigma_{DS,Op} \setminus \Sigma_{DT,Op}, f_{DS,Op}(op) = op':} \\
 & op_{V_{f_{DS}}(h'_{DS}(A'))} \circ h_{V_{f_{DS}}(A'),dom(op)} \\
= & \text{(restriction of operations, Def. 2.2.2)} \\
 & op'_{h'_{DS}(A')} \circ h'_{A',dom(op')} \\
= & (h'_{A'} \text{ partial algebra homomorphism, Def. 2.1.3)} \\
 & h'_{A',cod(op')} \circ op'_{A'} = h_{V_{f_{DS}}(A'),cod(op)} \circ op_{V_{f_{DS}}(A')} \\
= & \text{(assumption } V_{f_{DS}}(A') = V_{f_{DS}}(B')) \\
 & h_{V_{f_{DS}}(B'),cod(op)} \circ op_{V_{f_{DS}}(B')} = h'_{B',cod(op')} \circ op'_{B'} \\
= & (h'_{B'} \text{ partial algebra homomorphism, Def. 2.1.3)} \\
 & op'_{h'_{DS}(B')} \circ h'_{B',dom(op')} \\
= & \text{(restriction of operations, Def. 2.2.2)} \\
 & op_{V_{f_{DS}}(h'_{DS}(B'))} \circ h_{V_{f_{DS}}(B'),dom(op)}
\end{array}
$$

$$
\begin{tikzcd}
Dom(op'_{h'_{DS}(A')}) \arrow[d, "op'_{h'_{DS}(A')}"'] \arrow[rrrr, bend left, leftrightarrow, "="] & & Dom(op'_{A'}) = Dom(op'_{B'}) \arrow[ll, "h'_{A',dom(op')}"'] \arrow[rr, "h'_{B',dom(op')}"] \arrow[d, "op'_{A'}=op'_{B'}"] & & Dom(op'_{h'_{DS}(B')}) \arrow[d, "op'_{h'_{DS}(B')}"] \\
h'_{DS}(A')_{cod(op)} \arrow[rr, phantom, "(=)", yshift=8ex] \arrow[rrrr, bend right, leftrightarrow, "="'] & & A'_{cod(op)} = B'_{cod(op)} \arrow[ll, "h'_{A',cod(op')}"] \arrow[rr, "h'_{B',cod(op')}"'] \arrow[rr, phantom, "(=)", yshift=8ex] & & h'_{DS}(B')_{cod(op)}
\end{tikzcd}
$$

Figure 2.10: Restriction of data state homomorphism h'_{DS}

The state homomorphism $h'_{A',dom(op')} = h_{B',dom(op')}$ is surjective on the operation domain $Dom(op'_{h'_{DS}(A')}) = Dom(op'_{h'_{DS}(B')})$, because state homomorphisms preserve domains. Hence, the needed equality follows immeditately:

$$
\begin{array}{ll}
& op_{V_{f_{DS}}(h'_{DS}(A'))} \circ h'_{A',dom(op')} = \\
& op_{V_{f_{DS}}(h'_{DS}(B'))} \circ h'_{B',dom(op')} \\
\Longrightarrow & (h'_{A',dom(op')} = h'_{B',dom(op')} \text{ surjective}) \\
& op_{V_{f_{DS}}(h'_{DS}(A'))} = op_{V_{f_{DS}}(h'_{DS}(B'))}
\end{array}
$$

Homomorphism Properties: The properties, we need to show, are the identity of dynamic sorts, the h_A being partial algebra homomorphisms and the identity of dynamic operation domains. Since M_{DS} consists only of forgetful functor images of M'_{DS}, we can always assume the data states $A \in M_{DS}$ to be $A = V_{f_{DS}}(A')$ for some $A' \in M'_{DS}$.

- First we show, that the identity of dynamic sorts in $h_{DS}(A)$ is implied by the identity of dynamic sorts for $h'_{DS}(A')$. Here we need the requirement, that f_{DS} maps dynamic sorts to dynamic sorts.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall s \in \Sigma_{DS,S} \setminus \Sigma_{DT,S}, f_{DS,S}(s) = s':} \\
& h_{DS}(V_{f_{DS}}(A'))_s \\
= & \text{(restriction of data state homomorphism, Def. 2.2.6)} \\
& V_{f_{DS}}(h'_{DS}(A'))_s \\
= & \text{(partial algebra restriction, Def. 2.2.2)} \\
& h'_{DS}(A')_{s'} \\
= & (s' \in \Sigma'_{DS,S} \setminus \Sigma'_{DT,S}, \text{ Def. 2.2.4 and 2.1.8)} \\
& A'_{s'} \\
= & \text{(partial algebra restriction, Def. 2.2.2)} \\
& V_{f_{DS}}(A')_s
\end{array}
$$

- The partial algebra homomorphism $h_A: A \to h_{DS}(A)$ can be constructed as the restriction $V_{f_{DS}}(h'_{A'}: A' \to h'_{DS}(A'))$ according to Definition 2.2.3.
- For the preservation of operation domains we need the compatibility of f_{DS} with dynamic operations and the preservation of operation domains for $h'_{DS}(A')$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\forall op \in \Sigma_{DS,Op} \setminus \Sigma_{DT,Op}, f_{DS,Op}(op) = op':} \\
& h_{V_{f_{DS}}(A'),dom(op)}(Dom(op_{V_{f_{DS}}(A')})) \\
= & \text{(restriction of data state homomorph., Def. 2.2.6)} \\
& V_{f_{DS}}(h'_{A'})_{dom(op)}(Dom(op_{V_{f_{DS}}(A')})) \\
= & \text{(restriction of oper. and hom., Def. 2.2.2 and 2.2.3)} \\
& h'_{A',dom(op')}(Dom(op'_{A'})) \\
= & (op' \in \Sigma'_{DS,Op} \setminus \Sigma'_{DT,Op}, \text{ Def. 2.2.4 and 2.1.8)} \\
& Dom(op'_{h'_{DS}(A')}) \\
= & \text{(restriction of operations, Def. 2.2.2)} \\
& Dom(op_{V_{f_{DS}}(h'_{DS}(A'))}) \\
= & \text{(restriction of data state homomorph., Def. 2.2.6)} \\
& Dom(op_{h_{DS}(V_{f_{DS}}(A'))})
\end{array}
$$

Compatibility with Composition: Without loss of generality we also show the compatibility of restriction with composition only for states $V_{f_{DS}}(A')$, since for all data states in M_{DS} there exists such an A' in M'_{DS}:

$$\begin{array}{rl} & Restr_{f_{DS}}(h'_{DS} \circ g'_{DS})(V_{f_{DS}}(A')) \\ = & \text{(restriction of data state model hom., Def. 2.2.6)} \\ & V_{f_{DS}}(h'_{DS} \circ g'_{DS}(A')) \\ = & \text{(composition of data state model hom., 2.1.9)} \\ & V_{f_{DS}}(h'_{DS}(g'_{DS}(A'))) \\ = & \text{(restriction of data state model hom., Def. 2.2.6)} \\ & Restr_{f_{DS}}(h'_{DS})(V_{f_{DS}}(g'_{DS}(A'))) \\ = & \text{(restriction of data state model hom., Def. 2.2.6)} \\ & Restr_{f_{DS}}(h'_{DS}) \circ Restr_{f_{DS}}(g'_{DS})(V_{f_{DS}}(A')) \end{array}$$

Compatibility with Identities: Again we show this for data states $V_{f_{DS}}(A')$:

$$\begin{array}{rl} & Restr_{f_{DS}}(id_{M'_{DS}})(V_{f_{DS}}(A')) \\ = & \text{(restriction of data state model hom., Def. 2.2.6)} \\ & V_{f_{DS}}(id_{M'_{DS}}(A')) \\ = & \text{(identity of data state models, 2.1.10)} \\ & V_{f_{DS}}(A') \\ = & \text{(identity of data state models, 2.1.10)} \\ & id_{Restr_{f_{DS}}(M'_{DS})}(V_{f_{DS}}(A')) \end{array}$$

Hence the restriction of data state homomorphisms is well-defined, satisfies the homomorphism properties and preserves composition and identities. □

2.2.3 Transformation Layer

For transformation and process symbols, we have not only domain sorts, but parameter variables. This leads to parameter mappings, which give the possibility to have additional parameters and parameter identifications in a transformation signature morphism.

Definition 2.2.7 (Transformation Signature Morphism)
Given a data state signature morphism $f_{DS}: \Sigma_{DS} \to \Sigma'_{DS}$ and two corresponding transformation signatures (Σ_{Tr}, in, out) and (Σ'_{Tr}, in, out), a *transformation signature morphism* $f_{Tr}: (\Sigma_{Tr}, in, out) \to (\Sigma'_{Tr}, in, out)$ is a triple $(f_{Tr}, f_{in}, f_{out})$ with a function $f_{Tr}: \Sigma_{Tr} \to \Sigma'_{Tr}$, which assigns to each transformation symbol $t \in \Sigma_{Tr}$ a transformation symbol $f_{Tr}(t) \in \Sigma'_{Tr}$, and two families f_{in} and f_{out} of function families $f_{in(t)}: in(t) \to in(f_{Tr}(t))$ and $f_{out(t)}: out(t) \to out(f_{Tr}(t))$ assigning parameters of $f_{Tr}(t)$ with matching type to the parameters of each transformation symbol $t \in \Sigma_{Tr}$.

$$\begin{array}{ll} \multicolumn{2}{l}{f_{Tr}: (\Sigma_{Tr}, in, out) \to (\Sigma'_{Tr}, in, out) = (f_{Tr}, f_{in}, f_{out})} \\ \text{with} & f_{Tr}: \Sigma_{Tr} \to \Sigma'_{Tr} \\ & (f_{in(t)}: in(t) \to in(f_{Tr}(t)))_{t \in \Sigma_{Tr}} \\ & f_{in(t)} = (f_{in(t),s}: in(t)_s \to in(f_{Tr}(t))_{f_{DS,S}(s)})_{s \in \Sigma_{DS,S}} \\ & (f_{out(t)}: out(t) \to out(f_{Tr}(t)))_{t \in \Sigma_{Tr}} \\ & f_{out(t)} = (f_{out(t),s}: out(t)_s \to out(f_{Tr}(t))_{f_{DS,S}(s)})_{s \in \Sigma_{DS,S}} \end{array}$$

For each transformation symbol t of Σ_{Tr}, we obtain a restriction function, which constructs the transformation steps of t_M for a given transformation $f_{Tr}(t)_{M'}$.

Definition and Fact 2.2.8 (Transformation Expression Restriction)
Given a transformation signature morphism $f_{Tr}\colon (\Sigma_{Tr}, in, out) \to (\Sigma'_{Tr}, in, out)$, a transformation symbol $t \in \Sigma_{Tr}$ with $f_{Tr}(t) = t'$ and a data state model M'_{DS} for Σ'_{DS}, the *tranformation restriction function* $R_{f_{Tr},t}$ assigns to each transformation expression for t' and M'_{DS} a transformation expression for t and $Restr_{f_{DS}}(M'_{DS})$ defined by the forgetful functor on the data states and concatenation with the mappings for the parameters.

$$\begin{aligned} &R_{f_{Tr},t}\colon TrExpr(M'_{DS}, t') \to TrExpr(Restr_{f_{DS}}(M'_{DS}), t) \\ &\text{with} \quad R_{f_{Tr},t}(t'_{A',B'}(a', b')) = t_{V_{f_{DS}}(A'), V_{f_{DS}}(B')}(a' \circ f_{in(t)}, b' \circ f_{out(t)}) \end{aligned}$$

Proof: The states $V_{f_{DS}}(A')$ and $V_{f_{DS}}(B')$ are contained in $Restr_{f_{DS}}(M'_{DS})$ by definition.

The correct functionality of the parameter assignments is ensured by the sort compatibility of $f_{in(t)}$ and $f_{out(t)}$ as shown in Fig. 2.11. $\square$

$$\begin{tikzcd} in(t)_s \arrow[r, "f_{in(t),s}"] \arrow[dr, "(a' \circ f_{in(t)})_s"'] & in(t')_{f_{DT,S}(s)} \arrow[d, "a'_{f_{DT,S}(s)}"] \\ & A_s = A'_{f_{DT,S}(s)} \end{tikzcd}$$

(a) Input parameters

$$\begin{tikzcd} out(t)_s \arrow[r, "f_{out(t),s}"] \arrow[dr, "(b' \circ f_{out(t)})_s"'] & out(t')_{f_{DT,S}(s)} \arrow[d, "b'_{f_{DT,S}(s)}"] \\ & B_s = B'_{f_{DT,S}(s)} \end{tikzcd}$$

(b) Output parameters

Figure 2.11: Restriction of transformation parameters

Now the restriction for transformation models is obtained by restricting the transformation expressions in each transformation.

Definition and Fact 2.2.9 (Transformation Model Restriction)
Given a transformation signature morphism $f_{Tr}\colon (\Sigma_{Tr}, in, out) \to (\Sigma'_{Tr}, in, out)$ and a Σ'_{Tr}-model M'_{Tr}, the *restriction* of M'_{Tr} is a Σ_{Tr}-model $Restr_{f_{Tr}}(M'_{Tr}) = M_{Tr}$, which contains for each transformation symbol $t \in \Sigma_{Tr}$ the restrictions of all transformation steps of $f_{Tr}(t)_{M'}$.

$$\begin{aligned} &Restr_{f_{Tr}}(M'_{Tr}) = M_{Tr} = (t_M)_{t \in \Sigma_{Tr}} \\ &\text{with} \quad t_M = \left\{ R_{f_{Tr},t}(t'_{A',B'}(a', b')) \mid t'_{A',B'}(a', b') \in f_{Tr}(t)_{M'} \right\} \end{aligned}$$

Proof: Since, transformations are only demanded to be subclasses of the corresponding transformation expressions by Def. 2.1.12, the restriction is obviously a transformation model, since the restrictions of transformation steps are transformation steps due to Definition and Fact 2.2.8. $\square$

By extending this construction to transformation sets, where expressions for transformation symbols, which are not in the image of f_{Tr} are forgotten, we get a graph homomorphism between the DSTSs.

Definition 2.2.10 (Restriction-Induced DSTS-Homomorph.)
Given a data state signature morphism $f_{DS}: \Sigma_{DS} \to \Sigma'_{DS}$, a transformation signature morphism $f_{Tr}: (\Sigma_{Tr}, in, out) \to (\Sigma'_{Tr}, in, out)$, a Σ'-DSTS M'_{DSTS} and the corresponding Σ-DSTS M_{DSTS}, $f_{M'_{DSTS}}: M'_{DSTS} \to M_{DSTS}$ is a graph homomorphism between the DSTSs, given by the forgetful functor $V_{f_{DS}}$ on nodes and the extension of the transformation restriction $R_{f_{Tr}}$ to sets on edges.

$$
\begin{array}{ll}
f_{M'_{DSTS}}: M'_{DSTS} \to M_{DSTS} & \\
\text{with} \quad f_{M'_{DSTS}}(A') = V_{f_{DS}}(A') & \\
\qquad\quad f_{M'_{DSTS}}(T'_{A',B'}) = \{ \quad R_{f_{Tr},t}(t'_{A',B'}(a',b')) \mid & \\
\qquad\qquad\qquad\qquad\qquad\qquad f_{Tr}(t) = t', t'_{A',B'}(a',b') \in T'_{A',B'} \quad \} &
\end{array}
$$

To get the restriction of transformation homomorphisms, which is uniquely determined by the underlying data state homomorphism, we just need to have compatiblity of the homomorphism and restriction constructions.

Definition and Fact 2.2.11 (Transformation Hom. Restriction)
Given a transformation signature morphism $f_{Tr}: (\Sigma_{Tr}, in, out) \to (\Sigma'_{Tr}, in, out)$ and a Σ'_{Tr}-homomorphism $h'_{Tr}: M'_{Tr} \to N'_{Tr}$, the *restriction* of h'_{Tr} is a Σ_{Tr}-homomorphism $Restr_{f_{Tr}}(h'_{Tr}) = h_{Tr}: M_{Tr} \to N_{Tr}$ between the restrictions $M_{Tr} = Restr_{f_{Tr}}(M'_{Tr})$ and $N_{Tr} = Restr_{f_{Tr}}(N'_{Tr})$ assigning to the restrictions of transformation steps in M'_{Tr} the restrictions of the images under h'_{Tr}.

$$
\begin{array}{l}
Restr_{f_{Tr}}(h'_{Tr}: M'_{Tr} \to N'_{Tr}) = h_{Tr}: M_{Tr} \to N_{Tr} \\
\text{with} \quad h_t(R_{f_{Tr},t}(t'_{A',B'}(a',b')) = R_{f_{Tr},t}(h'_{t'}(t'_{A',B'}(a',b'))
\end{array}
$$

The restriction is compatible with composition and identities:

$$
\begin{array}{ll}
 & Restr_{f_{Tr}}(h'_{Tr} \circ g'_{Tr}) = Restr_{f_{Tr}}(h'_{Tr}) \circ Restr_{f_{Tr}}(g'_{Tr}) \\
\text{and} & Restr_{f_{Tr}}(id_{M'_{Tr}}) = id_{Restr_{f_{Tr}}(M'_{Tr})}
\end{array}
$$

Proof: It suffices to show the compatibility of homomorphisms and restrictions, i.e. the commutativity of the diagram in Fig. 2.12, because the transformation restrictions are defined in terms of $R_{f_{Tr},t}$ and, hence, the images of the restriction are well-defined, if they are compatible with $R_{f_{Tr},t}$.

$$
\begin{tikzcd}
t'_{M'} \arrow[r, "h'_{t'}"] \arrow[d, "R_{f_{Tr},t}"'] \arrow[dr, phantom, "(=)"] & t'_{N'} \arrow[d, "R_{f_{Tr},t}"] \\
t_M \arrow[r, "h_t"'] & t_N
\end{tikzcd}
$$

Figure 2.12: Restriction of transformation homomorphisms

$$
\begin{array}{ll}
& R_{f_{Tr},t}(h'_{t'}(t'_{A',B'}(a',b'))) \\
= & \text{(transformation homomorphism, Def. 2.1.16)} \\
& R_{f_{Tr},t}(t'_{h'_{DS}(A'),h'_{DS}(B')}(h'_{A'} \circ a', h'_{B'} \circ b')) \\
= & \text{(restriction of transformation expressions, Def. 2.2.8)} \\
& t_{V_{f_{DS}}(h'_{DS}(A')),V_{f_{DS}}(h'_{DS}(B'))}(h'_{A'} \circ a', h'_{B'} \circ b') \\
= & \text{(restriction of data state homomorphisms, Def. 2.2.6)} \\
& t_{h_{DS}(V_{f_{DS}}(A')),h_{DS}(V_{f_{DS}}(B'))}(h_{V_{f_{DS}}(A')} \circ a', h_{V_{f_{DS}}(B')} \circ b') \\
= & \text{(transformation homomorphism, Def. 2.1.16)} \\
& h_t(t_{V_{f_{DS}}(A'),V_{f_{DS}}(B')}(a',b')) \\
= & \text{(restriction of transformation expressions, Def. 2.2.8)} \\
& h_t(R_{f_{Tr},t}(t'_{A',B'}(a',b')))
\end{array}
$$

Since, transformation homomorphisms are unique w.r.t. the underlying data state homomorphism, the equalitites for the compatibility with composition and identities follow directly from the corresponding compatibilities on the data state layer. □

2.2.4 Process Layer

Since the syntax of process symbols is similar to that of transformation symbols, we also obtain a similar notion of signature morphism.

Definition 2.2.12 (Process Signature Morphism)
Given a data state signature morphism $f_{DS}: \Sigma_{DS} \to \Sigma'_{DS}$ and two corresponfing process signatures (Σ_{Pr}, in, out) and (Σ'_{Pr}, in, out), a *process signature morphism* $f_{Pr}: (\Sigma_{Pr}, in, out) \to (\Sigma'_{Pr}, in, out)$ is a triple $(f_{Pr}, f_{in}, f_{out})$ with a function $f_{Pr}: \Sigma_{Pr} \to \Sigma'_{Pr}$, which assigns to each process symbol $p \in \Sigma_{Pr}$ a process symbol $f_{Pr}(p) \in \Sigma'_{Pr}$, and two families f_{in} and f_{out} of function families $f_{in(p)}: in(p) \to in(f_{Pr}(p))$ and $f_{out(p)}: out(p) \to out(f_{Pr}(p))$ assigning parameters of $f_{Tr}(p)$ with matching type to the parameters of each transformation symbol $p \in \Sigma_{Pr}$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{f_{Pr}: (\Sigma_{Pr}, in, out) \to (\Sigma'_{Pr}, in, out) = (f_{Pr}, f_{in}, f_{out})} \\
\text{with} & f_{Pr}: \Sigma_{Pr} \to \Sigma'_{Pr} \\
& (f_{in(p)}: in(p) \to in(f_{Pr}(p)))_{p \in \Sigma_{Pr}} \\
& f_{in(p)} = (f_{in(p),s}: in(p)_s \to in(f_{Pr}(p))_{f_{DS,S}(s)})_{s \in \Sigma_{DS,S}} \\
& (f_{out(p)}: out(p) \to out(f_{Pr}(t)))_{p \in \Sigma_{Pr}} \\
& f_{out(p)} = (f_{out(p),s}: out(p)_s \to out(f_{Pr}(p))_{f_{DS,S}(s)})_{s \in \Sigma_{DS,S}}
\end{array}
$$

We also obtain a similar notion for the restriction of process expressions.

Definition and Fact 2.2.13 (Process Expression Restriction)
Given a process signature morphism $f_{Pr}: (\Sigma_{Pr}, in, out) \to (\Sigma'_{Pr}, in, out)$, a process symbol $p \in \Sigma_{Pr}$ with $f_{Pr}(p) = p'$ and a data state model M'_{DS} for Σ'_{DS}, the *process restriction function* $R_{f_{Pr},p}$ assigns to each process expression for p' and M'_{DS} a process expression for p and $Restr_{f_{DS}}(M'_{DS})$ defined by the forgetful functor on the data states and concatenation with the mappings for the

parameters.

$$\begin{array}{ll} R_{f_{Pr},p}: PrExpr(M'_{DS}, p') \to PrExpr(Restr_{f_{DS}}(M'_{DS}), p) \\ \text{with} \quad R_{f_{Pr},p}(p'_{A',B'}(a',b')) = p_{V_{f_{DS}}(A'),V_{f_{DS}}(B')}(a' \circ f_{in(p)}, b' \circ f_{out(p)}) \end{array}$$

Proof: Similar to Definition and Fact 2.2.8. □

The restriction of a process model is now constructed similar to transformations, for the processes themselves. For the RSTS, we take the identity, because we have no syntactical hint, how to restrict the structure inside the graphs. The occurence sets are obtained by taking the union of the occurence sets for all process expressions identified by the restriction and then concatenating all of them with the DSTS-homomorphism induced by the data state and transformation restriction.

Definition and Fact 2.2.14 (Process Model Restriction)
Given a process signature morphism $f_{Pr}: (\Sigma_{Pr}, in, out) \to (\Sigma'_{Pr}, in, out)$ and a Σ'_{Pr}-model M'_{Pr}, the *restriction* of M'_{Pr} is a Σ_{Pr}-model $Restr_{f_{Pr}}(M'_{Pr}) = M_{Pr}$, where the RSTSs for the process symbols in Σ_{Pr} are given by the RSTSs for the images in Σ'_{Pr}, the processes are constructed like the transformations on the transformation layer and the occurrences are constructed by concatenating the occurrences of the image process with the induced DSTS-homomorphism.

$$\begin{array}{ll} Restr_{f_{Pr}}(M'_{Pr}) = M_{Pr} \\ \text{with} \quad M_p = M'_{f_{Pr}(p)} \\ \quad p_M = \{ R_{f_{Pr},p}(p'_{A',B'}(a',b')) \mid p'_{A',B'}(a',b') \in f_{Pr}(p)_{M'} \} \\ \quad Occ_{M_p}(p_{A,B}(a,b)) = \{ f_{M'_{DSTS}} \circ occ' \mid f_{Pr}(p) = p', \\ \qquad\qquad occ' \in Occ_{M'_{p'}}(p'_{A',B'}(a',b')) \} \end{array}$$

Proof: We have to show that the restriction is a proper process model, which consists of the processes being correct similar to Definition and Fact 2.2.9, the RSTSs being obviously RSTSs since they are identical, the correct functionality of the occurences shown in Fig. 2.13 and the occurences mapping pathes from initial to final reactive states to pathes from initial to final data state:

$$\begin{array}{ll} & \exists i \in I_{M'_{f_{Pr}(p)}} = I_{M_p} \exists f \in F_{M'_{f_{Pr}(p)}} = F_{M_p}: \\ & \quad occ'(i) = A', occ'(f) = B', \\ & \quad i \xrightarrow{E^*_{M'_{f_{Pr}(p)}} = E^*_{M_p}} f \\ \Longrightarrow & \text{(restriction of DSTSs, Def. 2.2.10)} \\ & \exists i \in I_{M_p} \exists f \in F_{M_p}: \\ & \quad f_{M'_{DSTS}}(occ'(i)) = V_{f_{DS}}(A'), f_{M'_{DSTS}}(occ'(f)) = V_{f_{DS}}(B'), \\ & \quad i \xrightarrow{E^*_{M_p}} f \\ \Longleftrightarrow & \text{(restriction of occurences, Def. 2.2.14)} \\ & \exists i \in I_{M_p} \exists f \in F_{M_p}: \\ & \quad occ(i) = V_{f_{DS}}(A'), occ(f) = V_{f_{DS}}(B'), \\ & \quad i \xrightarrow{E^*_{M_p}} f \end{array}$$

□

$$\begin{tikzcd} G_{M_p} = G_{M'_{f_{Pr}(p)}} \arrow[r, "occ'"] \arrow[dr, "occ"'] & M'_{DSTS} \arrow[d, "f_{M'_{DSTS}}"] \\ & M_{DSTS} \end{tikzcd}$$

Figure 2.13: Restriction of occurences

The restriction of process homomorphisms is defined on the one hand, for graphs, like the restriction of data type homomorphisms, by just forgetting the graph homomorphism for additional symbols, on the other hand, for processes, similar to the restriction of transformation homomorphisms.

Definition and Fact 2.2.15 (Process Homomorphism Restriction)
Given a process signature morphism $f_{Pr}: (\Sigma_{Pr}, in, out) \to (\Sigma'_{Pr}, in, out)$ and a Σ'_{Pr}-homomorphism $h'_{Pr}: M'_{Pr} \to N'_{Pr}$, the *restriction* of h'_{Pr} is a Σ_{Pr}-homomorphism $Restr_{f_{Pr}}(h'_{Pr}) = h_{Pr}: M_{Pr} \to N_{Pr}$ between the restrictions $M_{Pr} = Restr_{f_{Pr}}(M'_{Pr})$ and $N_{Pr} = Restr_{f_{Pr}}(N'_{Pr})$. It consists of the graph homomorphisms for the image process symbols and process expression mappings constructed similar to the transformation restriction.

$$\begin{array}{ll} \multicolumn{2}{l}{Restr_{f_{Pr}}(h'_{Pr}: M'_{Pr} \to N'_{Pr}) = h_{Pr}: M_{Pr} \to N_{Pr}} \\ \text{with} & M_{Pr} = Restr_{f_{Pr}}(M'_{Pr}), N_{Pr} = Restr_{f_{Pr}}(N'_{Pr}) \\ & h_{G_p} = h'_{G_{f_{Pr}(p)}} \\ & h_p(R_{f_{Pr},p}(p'_{A',B'}(a',b'))) = R_{f_{Pr},p}(h'_{p'}(p'_{A',B'}(a',b'))) \end{array}$$

The restriction is compatible with composition and identities:

$$\begin{array}{ll} & Restr_{f_{Pr}}(h'_{Pr} \circ g'_{Pr}) = Restr_{f_{Pr}}(h'_{Pr}) \circ Restr_{f_{Pr}}(g'_{Pr}) \\ \text{and} & Restr_{f_{Pr}}(id_{M'_{Pr}}) = id_{Restr_{f_{Pr}}(M'_{Pr})} \end{array}$$

Proof: We have to show the homomorphism properties of the restriction and the compatibility with composition and identities.

Homomorphism Properties: Showing that the restriction is a proper homomorphism consists of the h_{G_p} being graph homomorphisms preserving initial and final states, the h_p being proper process expression mappings and the existence of an occurrence for the image of each process expression.

- Since the graph homomorphisms are identical to the corresponding ones in h'_{Pr}, preservation of initial and final states is inherited from h'_{Pr}.
- For the process expressions the proof is similar to the one of Definition and Fact 2.2.11.
- We still have to show. that the existence of an occurence for a process expression implies the existence of an occurence for the image

expression (see Fig. 2.14):

$$
\begin{array}{rl}
& \exists occ_M \in Occ_{M_p}(p_{A,B}(a,b)) \\
\Longrightarrow & \text{(restriction of occurences, Def. 2.2.14)} \\
& \exists occ_{M'} \in Occ_{M'_{f_{Pr}(p)}}(p'_{A',B'}(a',b')) \\
& \;\text{with } occ_M = f_{M'_{DSTS}} \circ occ_{M'} \\
& \;\text{for some } p'_{A',B'}(a',b') \\
& \;\text{with } R_{f_{Pr},p}(p'_{A',B'}(a',b')) = p_{A,B}(a,b) \\
\Longrightarrow & (h'_{Pr} \text{ process homomorphism, Def. 2.1.24)} \\
& \exists occ_{N'} \in Occ_{N'_{f_{Pr}(p)}}(h'_p(p'_{A',B'}(a',b'))) \\
\Longrightarrow & \text{(restriction of occurences, Def. 2.2.14)} \\
& \exists f_{N_{DSTS}} \circ occ_{N'} \in Occ_{N_p}(R_{f_{Pr},p}(h'_p(p'_{A',B'}(a',b')))) \\
\Longrightarrow & \text{(restriction of process hom., Def. 2.2.15)} \\
& \exists occ_N \in Occ_{N_p}(h_p(p_{A,B}(a,b)))
\end{array}
$$

$$
\begin{tikzcd}
G_{M_p} = G_{M'_{f_{Pr}(p)}} \arrow[r, "occ_{M'}"] \arrow[rr, bend left, "occ_M"] \arrow[d, "h_{G_p}=h'_{G_{f_{Pr}(p)}}"'] & M'_{DSTS} \arrow[r, "f_{M'_{DSTS}}"] \arrow[d, dashed, "h'_{DSTS}"] & M_{DSTS} \arrow[d, "h_{DSTS}"] \\
G_{N_p} = G_{N'_{f_{Pr}(p)}} \arrow[r, "occ_{N'}"'] \arrow[rr, bend right, "occ_N"'] & N'_{DSTS} \arrow[r, "f_{N_{DSTS}}"'] & N_{DSTS}
\end{tikzcd}
$$

Figure 2.14: Restriction of process homomorphisms

Compatibility with Composition and Identity: Compatibility with composition and identities can be shown for the graph part like for the data type restriction with graphs instead of carrier sets, and for the processes like for transformations.

□

2.2.5 Integration

Now we combine the signature morphisms and restriction constructions on the different layers into integrated notions.

Definition 2.2.16 (Integrated Signature Morphism)
Given two integrated signatures Σ and Σ', an *integrated signature morphism* $f: \Sigma \to \Sigma'$ consists of

- a data type signature morphism $f_{DT}: \Sigma_{DT} \to \Sigma'_{DT}$ as in Def. 2.2.1,
- a data state signature morphism $f_{DS}: \Sigma_{DS} \to \Sigma'_{DS}$ w.r.t. f_{DT} as in Def. 2.2.4,

- a transformation signature morphism $f_{Tr}: (\Sigma_{Tr}, in, out) \rightarrow (\Sigma'_{Tr}, in, out)$ w.r.t. f_{DS} as in Def. 2.2.7 and
- a process signature morphism $f_{Pr}: (\Sigma_{Pr}, in, out) \rightarrow (\Sigma'_{Pr}, in, out)$ w.r.t. f_{DS} as in Def. 2.2.12.

In contrast to the integrated definition of model (Def. 2.1.28), we do not have to demand the compatibility of the layers in the definition of integrated restrictions, since it results from the compatibility in the unrestricted model.

Definition and Fact 2.2.17 (Integrated Model Restriction)
Given an integrated signature morphism $f: \Sigma \rightarrow \Sigma'$ and a Σ'-model M' the *restriction* of M' is a Σ-model $Restr_f(M') = M$ consisting of

- the data type restriction $M_{DT} = V_{f_{DT}}(M'_{DT})$ as in Def. 2.2.2,
- the data state restriction $M_{DS} = Restr_{f_{DS}}(M'_{DS})$ as in Def. 2.2.5,
- the transformation restriction $M_{Tr} = Restr_{f_{Tr}}(M'_{Tr})$ as in Def. 2.2.9 and
- the process restriction $M_{Pr} = Restr_{f_{Pr}}(M'_{Pr})$ as in Def. 2.2.14.

Proof: M is a proper Σ-model, since the restrictions on the layers are proper models, as shown in the corresponding subsections. □

For the restriction of homomorphisms, we also integrate the compatibility with composition and identities, which is needed for the functor properties of restriction in Theorem 2.2.19.

Definition and Fact 2.2.18 (Integrated Homomorphism Restriction)
Given an integrated signature morphism $f: \Sigma \rightarrow \Sigma'$ and a Σ'-homomorphism $h': M' \rightarrow N'$ the *restriction* of h' is a Σ-homomorphism $Restr_f(h') = h: M \rightarrow N$, where $M = Restr_f(M')$ and $N = Restr_f(N')$, consisting of

- the restriction $h_{DT}: M_{DT} \rightarrow N_{DT} = V_{f_{DT}}(h'_{DT}: M'_{DT} \rightarrow N'_{DT})$ of the data type homomorphism as in Def. 2.2.3,
- the restriction $h_{DS}: M_{DS} \rightarrow N_{DS} = Restr_{f_{DS}}(h'_{DS}: M'_{DS} \rightarrow N'_{DS})$ of the data state homomorphism as in Def. 2.2.6,
- the restriction $h_{Tr}: M_{Tr} \rightarrow N_{Tr} = Restr_{f_{Tr}}(h'_{Tr}: M'_{Tr} \rightarrow N'_{Tr})$ of the transformation homomorphism as in Def. 2.2.11 and
- the restriction $h_{Pr}: M_{Pr} \rightarrow N_{Pr} = Restr_{f_{Pr}}(h'_{Pr}: M'_{Pr} \rightarrow N'_{Pr})$ of the process homomorphism as in Def. 2.2.15.

The restriction is compatible with composition and identities:

$$\begin{aligned} & Restr_f(h' \circ g') = Restr_f(h') \circ Restr_f(g') \\ \text{and} \quad & Restr_f(id_{M'}) = id_{Restr_f(M')} \end{aligned}$$

Proof: h is a proper Σ-homomorphism, since the homomorphisms on the layers are proper homomorphisms, as shown in the corresponding subsections. □

As the main result of this section, we get the following theorem, interpreting the restriction construction for a signature morphism as a functor from the model category of the second signature to the model category of the first one.

Theorem 2.2.19 (Restriction Functor)
The restriction construction for an integrated signature morphism $f\colon \Sigma \to \Sigma'$ forms a functor $Restr_f\colon Mod(\Sigma') \to Mod(\Sigma)$.

Proof: The necessary properties of a functor are exactly the compatibility with composition and identity, proven in Definition and Fact 2.2.18. □

2.3 Integrated Signature Specification Frame

Specification frames are a concept for the formulation and classification of structural aspects common to many formal specification techniques introduced in [EGR94]. A specification frame defines a category of abstract specifications and a model functor assigning models and forgetful functors to abstract specifications and abstract specification morphisms.

Definition 2.3.1 (Specification Frame [EGR94])
A *specification frame* $SF = (\mathbf{ASpec}, Cat)$ consists of a category **ASpec**, called *category of abstract specifications*, and a functor $Cat\colon \mathbf{ASpec} \to \mathbf{Cat}$, called *model functor*.

With the signature morphisms and the restriction construction, we can extend the class of all integrated signatures Σ to become a category **IntSig** and the assignment of model categories $Mod(\Sigma)$ to become a contravariant functor $Mod\colon \mathbf{IntSig}^{op} \to \mathbf{Cat}$. Hence, we get a specification frame $IntSigSF$.

Theorem 2.3.2 (Integrated Signature Specification Frame)
The category **IntSig** of integrated signatures and integrated signature morphisms, together with the functor $Mod\colon \mathbf{IntSig}^{op} \to \mathbf{Cat}$ defined by

$Mod(\Sigma)$ (the model category as in Theorem 2.1.32)
$Mod(f\colon \Sigma \to \Sigma') = Restr_f\colon Mod(\Sigma') \to Mod(\Sigma)$
(the restriction functor as in Theorem 2.2.19)

forms a specification frame, called *specification frame of integrated signatures* $IntSigSF$.

Proof: We have to show that **IntSig** is a category and Mod a functor.

IntSig Category: For **IntSig** being a category, signature morphisms have to be closed under composition, which is associative, because it consists of functions, for which the composition is associative.

For the algebraic signatures on the first two layers, we have that $dom \circ g_{Op} \circ f_{Op} = (g_S)^* \circ dom \circ f_{Op}$ by the compatibility of g and $(g_S)^* \circ dom \circ f_{Op} = (g_S)^* \circ (f_S)^* \circ dom$ by the compatibility of f, and analogously for cod.

On the other layers there are no compatibility requirements, but just functions, which can be composed straightforward.

Moreover there have to be neutral identities, which is the case, because the morphisms consisting of identities in all parts are compatible with the domains and codomains.

Mod **Functor:** Now showing that Mod is a functor consists of the compatibility with composition and identities.

There are basically two kinds of restriction constructions, on the one hand, for carrier sets, operations and RSTSs, the exact model element for the image symbol is taken, on the other hand, for data states, transformations, processes and occurences, an inner restriction is applied to the model element. The first kind can be seen as a special case of the second one, where the inner restriction is the identity.

Therefore, without loss of generality, we only have to show the properties for the following generic version of the construction:

$$Restr_f(M')_x = \{\, R_{f,x}(X') \mid X' \in M'_{f(x)} \,\}$$

Here, x is a syntactical symbol, and $R_{f,x}$ is the inner restriction. The X' are the elements of the carrier sets, the operation applications (single tuples in the operation graph), resp. the nodes and edges of the RSTSs for the first kind of restriction, for the second kind it are data states, transformation or process expressions or occurences.

Preservation of Composition: We have to show:

$$Restr_{g\circ f} = Restr_f \circ Restr_g$$

For the left hand side we get by applying the generic restriction above:

$$Restr_{g\circ f}(M'')_x \quad = \left\{\, R_{g\circ f,x}(X'') \mid X'' \in M''_{g\circ f(x)} \,\right\}$$

For the right hand side sequential application of the restriction yields:

$$\begin{aligned}
&Restr_f(Restr_g(M''))_x = \\
&\{\, R_{f,x}(X') \mid X' \in Restr_g(M'')_{f(x)} \,\} = \\
&\left\{\, R_{f,x}(X') \mid X' \in \left\{\, R_{g,f(x)}(X'') \mid X'' \in M''_{g(f(x))} \,\right\} \,\right\} = \\
&\left\{\, R_{f,x}(R_{g,f(x)}(X'')) \mid X'' \in M''_{g(f(x))} \,\right\}
\end{aligned}$$

Since $g \circ f(x) = g(f(x))$ by definition of function composition, for both sides to be identical, we get the sufficient but not necessary property, that the inner restriction satisfies:

$$R_{g\circ f,x} = R_{f,x} \circ R_{g,f(x)}$$

For the identities this is obvious. For the data states, the inner restriction is the fogetful functor for partial algebras, which in turn satisfies the property, because its inner restrictions are identities.

For transformation and process expressions, we get by applying the definition and the fact, that partial algebra restriction is a functor:

$$\begin{aligned}
&R_{g\circ f,x}\left(g(f(x))_{A'',B''}(a,b)\right) = \\
&x_{V_{g_{DS}\circ f_{DS}}(A''),V_{g_{DS}\circ f_{DS}}(B'')}(a,b) = \\
&x_{V_{f_{DS}}(V_{g_{DS}}(A'')),V_{f_{DS}}(V_{g_{DS}}(A''))}(a,b) = \\
&R_{f,x}\left(f(x)_{V_{g_{DS}}(A''),V_{g_{DS}}(A'')}(a,b)\right) = \\
&R_{f,x}\left(R_{g,f(x)}\left(g(f(x))_{A'',B''}(a,b)\right)\right)
\end{aligned}$$

Finally, for occurrences the inner restriction is concatenation with $f_{M'_{DSTS}}$, which in turn consists of $V_{f_{DS}}$ and $R_{f_{Tr}}$, whence it satisfies the restriction properties:

$$\begin{aligned}
&R_{g\circ f,p}\left(occ\colon M''_{g(f(p))} \to M''_{DSTS}\right) = \\
&(g\circ f)_{M''_{DSTS}} \circ occ\colon M_p \to M_{DSTS} = \\
&f_{M'_{DSTS}} \circ g_{M''_{DSTS}} \circ occ\colon M_p \to M_{DSTS} = \\
&R_{f,p}\left(g_{M''_{DSTS}} \circ occ\colon M'_{f(p)} \to M'_{DSTS}\right) = \\
&R_{f,p}\left(R_{g,f(p)}\left(occ\colon M''_{g(f(p))} \to M''_{DSTS}\right)\right)
\end{aligned}$$

Preservation of Identities: Here, we have to show:

$$Restr_{id_\Sigma} = id_{Mod(\Sigma)}$$

The left hand side yields, with the generic restriction:

$$Restr_{id_\Sigma}(M)_x = \left\{ R_{id_\Sigma,x}(X) \mid X \in M_{id_\Sigma(x)} \right\}$$

The right hand side with the definition of model identities leads to:

$$id_{Mod(\Sigma)}(M)_x = M_x$$

Now, since $id_\Sigma(x) = x$, we obtain as a sufficient but not necessary property for the equality of both sides:

$$R_{id_\Sigma,x}(X) = X$$

For sorts, operations and RSTSs inner restrictions are identities by definition. For data states, we can again recursively argue, that inner restriction is the partial algebra forgetful functor, whose inner restriction is identity, and, hence, the forgetful functor for $id_{\Sigma_{DS}}$ maps data states identical.

It remains to show that restrictions of transformation and process expressions and occurences, are identities. For the expressions, we again recurse to the partial algebra forgetful functor:

$$\begin{aligned}
&R_{id_\Sigma,x}\left(x_{A,B}(a,b)\right) = \\
&x_{V_{id_{\Sigma,DS}}(A),V_{id_{\Sigma,DS}}(B)}(a,b) = \\
&x_{A,B}(a,b)
\end{aligned}$$

The DSTS homomorphism, with which occurences are concatenated by inner restriction, is the identity, since $V_{id_{\Sigma,DS}}$ and $R_{id_{\Sigma,Tr}}$ are identities:

$$\begin{aligned}
&R_{id_\Sigma,p}(occ) = \\
&id_{\Sigma,M_{DSTS}} \circ occ = \\
&id_{M_{DSTS}} \circ occ = occ
\end{aligned}$$

$$
\begin{tikzcd}
\mathbf{IntSig} & \Sigma \arrow[rr, "f"] & & \Sigma' \\
\mathbf{IntSig}^{op} \arrow[dd, "Mod"'] & \Sigma \arrow[dd, mapsto] & & \Sigma' \arrow[ll] \arrow[dd, mapsto] \\
 & & f \arrow[d, mapsto] & \\
\mathbf{Cat} & Mod(\Sigma) & Restr_f & Mod(\Sigma') \arrow[ll]
\end{tikzcd}
$$

Figure 2.15: Integrated signature specification frame

Hence, **IntSig** is a category, Mod a contra-variant functor and $IntSigSF$ a specification frame. □

The specification $IntSigSF$ encapsulates all notions introduced in this chapter, signatures, model categories, signature morphisms and restriction functors. The relations between these notions are summarized in Figure 2.15.

Chapter 3

Structuring Concepts

In this chapter we will introduce means to structure large integrated signatures and models by *union and amalgamation* (Section 3.1) and *refinement and abstraction* (Section 3.2).

Both constructions are new and have not been considered for the integration framework in [EO01a, EO01b, EOB+02b] so far.

We will not prove the constructions in this chapter formally, but only give informal arguments, that they satisfy the needed properties.

3.1 Union of Signatures and Amalgamation of Models

Union (on the syntactical side) and amalgamation (on the semantical side) are means to compose signatures and models with common parts. The common subpart is given by a signature Σ^0 with morphisms into both signatures Σ^1 and Σ^2 that are to be composed. Then the union is constructed as the pushout object Σ^3 in Fig. 3.1(a).

Two models $M^1 \in Mod(\Sigma^1)$ and $M^2 \in Mod(\Sigma^2)$ can be amalgamated, if the restriction $M^0 \in Mod(\Sigma^0)$ to the common subsignature is the same for both models, leading to a model $M^3 \in Mod(\Sigma^3)$ such that the restriction to Σ^1 is M^1 and the restriction to Σ^2 is M^2 (see Fig. 3.1(b)).

The notion of amalgamation, as it is used for example in [EM85] and [EGR94], normally includes the assumption, that the amalgamation is unique, and, hence, the models of Σ^3 can be decomposed into their restrictions to Σ^1 and Σ^2, such that the original model is retrieved by re-amalgamating the restrictions.

In this sense, our amalgamation is only a "weak" amalgamation, because the uniqueness and decomposition properties are not fullfilled. We will see the reasons for that in the corresponding subsections, namely in Subsection 3.1.2 and 3.1.3. In the following we will just write "amalgamation", when our weak version is meant, and "strong amalgamation", when uniqueness and decomposition are fullfilled.

$$\begin{tikzcd}
 & \Sigma^0 \arrow[dl, "f^1"'] \arrow[dr, "f^2"] & \\
\Sigma^1 \arrow[dr, "g^1"'] \arrow[rr, phantom, "(PO)"] & & \Sigma^2 \arrow[dl, "g^2"] \\
 & \Sigma^3 &
\end{tikzcd}$$

(a) Union

$$\begin{tikzcd}
 & M^0 & \\
M^1 \arrow[ur, mapsto, "Restr_{f^1}"] \arrow[rr, phantom, "(=)"] & & M^2 \arrow[ul, mapsto, "Restr_{f^2}"'] \\
 & M^3 \arrow[ul, mapsto, "Restr_{g^1}"] \arrow[ur, mapsto, "Restr_{g^2}"'] &
\end{tikzcd}$$

(b) Amalgamation

Figure 3.1: Union and amalgamation

Moreover, we only get amalgamations for signature morphisms, which are injective on transformation and process symbols, because the inner restrictions could otherwise lead to incompatibilities.

Again, we will first look at the layers in detail and then, in Subsection 3.1.5, combine them to the integrated notions.

3.1.1 Data Type Layer

The union of data type signatures depicted in Fig. 3.2, which is a pushout of algebraic signatures, can be constructed by pushouts in **Set** for the sort and operation symbol sets.

$$\begin{tikzcd}
 & & \Sigma^1_{DT,Op} \arrow[d, shift right, "dom"'] \arrow[d, shift left, "cod"] \arrow[drr, "g^1_{DT,Op}"] & & \\
\Sigma^0_{DT,Op} \arrow[urr, "f^1_{DT,Op}"] \arrow[d, shift right, "dom"'] \arrow[d, shift left, "cod"] \arrow[drr, "f^2_{DT,Op}"] & & (\Sigma^1_{DT,S})^* \arrow[drr, "(g^1_{DT,S})^*"] & & \Sigma^3_{DT,Op} \arrow[d, shift right, "dom"'] \arrow[d, shift left, "cod"] \\
(\Sigma^0_{DT,S})^* \arrow[urr, "(f^1_{DT,S})^*"] \arrow[drr, "(f^2_{DT,S})^*"'] & & \Sigma^2_{DT,Op} \arrow[urr, "g^2_{DT,Op}"] \arrow[d, shift right, "dom"'] \arrow[d, shift left, "cod"] & & (\Sigma^3_{DT,S})^* \\
 & & (\Sigma^2_{DT,S})^* \arrow[urr, "(g^2_{DT,S})^*"'] & &
\end{tikzcd}$$

Figure 3.2: Union of data type signatures

Definition 3.1.1 (Union of Data Type Signatures)
Given data type signatures Σ^0_{DT}, Σ^1_{DT} and Σ^2_{DT} with data type signature mor-

phisms $f^1_{DT}: \Sigma^0_{DT} \to \Sigma^1_{DT}$ and $f^2_{DT}: \Sigma^0_{DT} \to \Sigma^2_{DT}$, the *union* of Σ^1_{DT} and Σ^2_{DT} over Σ^0_{DT} is a pushout $\Sigma^1_{DT} \cup_{\Sigma^0_{DT}} \Sigma^2_{DT} = \Sigma^3_{DT}$ in the category **AlgSig** constructed by pushouts of the symbol sets:

$$\begin{array}{l} \Sigma^3_{DT,S} = \Sigma^1_{DT,S} \cup_{\Sigma^0_{DT,S}} \Sigma^2_{DT,S} \\ \Sigma^3_{DT,Op} = \Sigma^1_{DT,Op} \cup_{\Sigma^0_{DT,Op}} \Sigma^2_{DT,Op} \end{array}$$

The morphisms $g^i_{DT}: \Sigma^i_{DT} \to \Sigma^3_{DT}$ for $i \in \{1,2\}$ consist of the functions $g^i_{DT,S}$ and $g^i_{DT,Op}$ given by the pushout constructions for the symbol sets. The domain and codomain functions are given by the translation of the corresponding domain and codomain functions in Σ^1_{DT} and Σ^2_{DT}.

$$\begin{array}{l} dom(g^i_{DT,Op}(op^i)) = (g^i_{DT,S})^*(dom(op^i)) \\ cod(g^i_{DT,Op}(op^i)) = (g^i_{DT,S})^*(cod(op^i)) \end{array}$$

An amalgamated data type is constructed by taking for a symbol x in Σ^3_{DT} the model elements of M^1_{DT} and/or M^2_{DT} for symbols, which are mapped to x. This construction works, because, if there are multiple symbols, which are mapped to x, then the pushout property ensures, that they are the images of symbols in Σ^0_{DT}, for which the model elements in the restrictions have to be identical.

On the other hand this immediately ensures, that the restrictions of the amalgamation to Σ^1_{DT} resp. Σ^2_{DT} again yield M^1_{DT} resp. M^2_{DT}. On this layer we obtain a strong amalgamation, where the restriction to Σ^1 and Σ^2 leads to a decomposition of models, because the model elements are kept identical by the restriction as well as the amalgamation.

Definition 3.1.2 (Amalgamation of Data Type Models)
Given a union of data type signatures, a Σ^1_{DT}-model M^1_{DT} and a Σ^2_{DT}-model M^2_{DT}, such that $V_{f^1_{DT}}(M^1_{DT}) = M^0_{DT} = V_{f^2_{DT}}(M^2_{DT})$, the *amalgamation* of M^1_{DT} and M^2_{DT} over M^0_{DT} is a Σ^3_{DT}-model $M^1_{DT} +_{M^0_{DT}} M^2_{DT} = M^3_{DT}$, which is constructed by taking the carrier sets and operations of M^1_{DT} and M^2_{DT} respectively for symbols who are assigned to corresponding symbols of Σ^1_{DT} and Σ^2_{DT}.

$$\begin{array}{l} M^1_{DT} +_{M^0_{DT}} M^2_{DT} = M^3_{DT} \\ \text{with} \quad M^3_{s^3} = \begin{cases} M^1_{s^1} & \text{if } s^3 = g^1_{DT,S}(s^1) \\ M^2_{s^2} & \text{if } s^3 = g^2_{DT,S}(s^2) \end{cases} \\ \qquad\quad op^3_{M^3} = \begin{cases} op^1_{M^1} & \text{if } op^3 = g^1_{DT,Op}(op^1) \\ op^2_{M^2} & \text{if } op^3 = g^2_{DT,Op}(op^2) \end{cases} \end{array}$$

We can also apply the amalgamtion construction to homomophisms of data types, leading to amalgamated homomorphisms between the amalgamations of the corresponding sources and targets.

Definition 3.1.3 (Amalgamation of Data Type Hom.)
Given a union of data type signatures, a Σ^1_{DT}-homomorphism $h^1_{DT}: M^1_{DT} \to N^1_{DT}$ and a Σ^2_{DT}-homomorphism $h^2_{DT}: M^2_{DT} \to N^2_{DT}$, such that $V_{f^1_{DT}}(h^1_{DT}) = h^0_{DT} = V_{f^2_{DT}}(h^2_{DT})$, the *amalgamation* of h^1_{DT} and h^2_{DT} over h^0_{DT} is a Σ^3_{DT}-homomorphism $h^1_{DT} +_{h^0_{DT}} h^2_{DT} = h^3_{DT}$, which is constructed by taking the sort

functions of h^1_{DT} and h^2_{DT} respectively for sort symbols who are assigned to corresponding symbols of Σ^1_{DT} and Σ^2_{DT}.

$$\begin{array}{ll} & h^1_{DT} +_{h^0_{DT}} h^2_{DT} = h^3_{DT}\colon M^3_{DT} \to N^3_{DT} \\ \text{with} & M^3_{DT} = M^1_{DT} +_{M^0_{DT}} M^2_{DT}, N^3_{DT} = N^1_{DT} +_{N^0_{DT}} N^2_{DT} \\ & h^3_{s^3} = \begin{cases} h^1_{s^1} & \text{if } s^3 = g^1_{DT,S}(s^1) \\ h^2_{s^2} & \text{if } s^3 = g^2_{DT,S}(s^2) \end{cases} \end{array}$$

3.1.2 Data State Layer

On the data state layer, there is just another algebraic signature, so we can take the same construction as above. Since pushouts are only unique up to isomorphism, we have to take care, that the pushout signature on the data type layer is a subsignature of the pushout signature on the data state layer.

Definition 3.1.4 (Union of Data State Signatures)
Given a union of data type signatures and corresponding data state signatures Σ^0_{DS}, Σ^1_{DS} and Σ^2_{DS} with morphisms $f^1_{DS}\colon \Sigma^0_{DS} \to \Sigma^1_{DS}$ and $f^2_{DS}\colon \Sigma^0_{DS} \to \Sigma^2_{DS}$, the *union* is again a pushout $\Sigma^1_{DS} \cup_{\Sigma^0_{DS}} \Sigma^2_{DS} = \Sigma^3_{DT}$ in the category **AlgSig**, where we select a pushout, such that $\Sigma^3_{DT} \subseteq \Sigma^3_{DS}$.

Here is one of the points, where the weakness of our amalgamation is rooted. The data state classes M^1_{DS} and M^2_{DS} can both contain several data states with the same restriction in M^0_{DS}. Now we construct the maximal model where every pair of fitting data states from M^1_{DS} and M^2_{DS} is amalgamated, but a model where there are only a few of these amalgamations would yield the same restrictions, as long as there is at least one data state in M^3_{DS} for each data state in M^1_{DS} or M^2_{DS}.

Definition 3.1.5 (Amalgamation of Data State Models)
Given a union of data state signatures, a Σ^1_{DS}-model M^1_{DS} and a Σ^2_{DS}-model M^2_{DS}, such that $Restr_{f^1_{DS}}(M^1_{DS}) = M^0_{DS} = Restr_{f^2_{DS}}(M^2_{DS})$, the *amalgamation* of M^1_{DS} and M^2_{DS} over M^0_{DS} is a Σ^3_{DS}-model $M^1_{DS} +_{M^0_{DS}} M^2_{DS} = M^3_{DS}$, which consists of all possible amalgamations of the data states from M^1_{DS} and M^2_{DS}.

$$\begin{array}{lll} & M^1_{DS} +_{M^0_{DS}} M^2_{DS} = M^3_{DS} & \\ \text{with} & M^3_{DS} = \{ & A^1 +_{A^0} A^2 \mid A^1 \in M^1_{DS}, A^2 \in M^2_{DS}, \\ & & A^0 = V_{f^1_{DS}}(A^1) = V_{f^2_{DS}}(A^2) \in M^0_{DS} \quad \} \end{array}$$

Again this construction can be extended to homomorphisms, since the construction of the data state models ensures that every data state in M^3_{DS} is constructed from data states of both sides, for which the homomorphism images can be amalgamated, which yields the homomorphism image also being included in M^3_{DS}.

Definition 3.1.6 (Amalgamation of Data State Hom.)
Given a union of data state signatures, a Σ^1_{DS}-homomorphism $h^1_{DS}\colon M^1_{DS} \to N^1_{DS}$ and a Σ^2_{DS}-homomorphism $h^2_{DS}\colon M^2_{DS} \to N^2_{DS}$, so that $Restr_{f^1_{DS}}(h^1_{DS}) =$

$h^0_{DS} = Restr_{f^2_{DS}}(h^2_{DS})$, the *amalgamation* of h^1_{DS} and h^2_{DS} over h^0_{DS} is a Σ^3_{DS}-homomorphism $h^1_{DS} +_{h^0_{DS}} h^2_{DS} = h^3_{DS}$, which maps the amalgamations of data states to the amalgamations of the images under the homomorphisms.

$$\begin{array}{l} h^1_{DS} +_{h^0_{DS}} h^2_{DS} = h^3_{DS}\colon M^3_{DS} \to N^3_{DS} \\ \text{with} \quad M^3_{DS} = M^1_{DS} +_{M^0_{DS}} M^2_{DS}, N^3_{DS} = N^1_{DS} +_{N^0_{DS}} N^2_{DS} \\ \qquad\quad h^3_{DS}(A^1 +_{A^0} A^2) = h^1_{DS}(A^1) +_{h^0_{DS}(A^0)} h^2_{DS}(A^2) \end{array}$$

3.1.3 Transformation Layer

The union of transformation signatures is constructed by a **Set**-pushout of the transformation symbols, where the parameter variables are given by by pushouts for common transformation symbols and by just translating the sorts of the variables for transformation symbols, which stem from just one of the signatures.

Definition 3.1.7 (Union of Transformation Signatures)
Given a union of data state signatures and corresponding transformation signatures (Σ^0_{Tr}, in, out), (Σ^1_{Tr}, in, out) and (Σ^2_{Tr}, in, out) with morphisms f^1_{Tr} and f^2_{Tr}, the *union* (Σ^3_{Tr}, in, out) is constructed by a pushout $\Sigma^3_{Tr} = \Sigma^1_{Tr} \cup_{\Sigma^0_{Tr}} \Sigma^2_{Tr}$ in the category **Set** for the transformation symbols, where the parameters are also obtained by pushouts $in(t^3) = in(t^1) \cup_{in(t^0)} in(t^2)$ for transformation symbols in the common part. For transformation symbols not in the common part, the sorts of the parameters are just translated according to the data state layer.

Transformations are the second reason for the weakness of our amalgamation, because again the existence of several transformation steps on both sides, which are identified by the restriction, leads to several possible Σ^3_{Tr}-models, having the same restriction. We also choose the maximal model, where all possible amalgamated transformations are included.

At this point we need the injectivity of transformation signature morphisms, because otherwise we would have to amalgamate transformation steps for all transformation symbols identified by the signature morphism, but we could not guarantee their compatibility.

Definition 3.1.8 (Amalgamation of Transformations)
Given a union of transformation signatures with a transformation symbol t^0 in the common part and two transformations $t^1_{M^1}$ and $t^2_{M^2}$ for the image transformation symbols $t^1 = f^1_{Tr}(t^0)$ and $t^2 = f^2_{Tr}(t^0)$, such that the restriction to Σ^0 is the same transformation $R_{f^1_{Tr},t^0}(t^1_{M^1}) = t^0_{M^0} = R_{f^2_{Tr},t^0}(t^2_{M^2})$, the *amalgamated transformation* $t^1_{M^1} +_{t^0_{M^0}} t^2_{M^2} = t^3_{M^3}$ for the common transformation symbol $g^1_{Tr}(t^1) = t^3 = g^2_{Tr}(t^2)$ in Σ^3_{Tr} is obtained by combining all transformation steps that agree on the common part of the transformed data states and

on the common parameters.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{For } t^3 = g^1_{Tr}(t^1) = g^2_{Tr}(t^2):} \\
\multicolumn{2}{l}{t^3_{M^3} = t^1_{M^1} +_{t^0_{M^0}} t^2_{M^2} =} \\
\{t^3_{A^3,B^3}(a^3,b^3) \mid & t^0_{A^0,B^0}(a^0,b^0) \in t^0_{M^0}, \\
& t^1_{A^1,B^1}(a^1,b^1) \in t^1_{M^1}, t^2_{A^2,B^2}(a^2,b^2) \in t^2_{M^2}, \\
& A^0 = V_{f^1_{DS}}(A^1) = V_{f^2_{DS}}(A^2), A^3 = A^1 +_{A^0} A^2, \\
& B^0 = V_{f^1_{DS}}(B^1) = V_{f^2_{DS}}(B^2), B^3 = B^1 +_{B^0} B^2, \\
& a^0 = a^1 \circ f^1_{in(t^0)} = a^2 \circ f^2_{in(t^0)}, \\
& a^3(x^3) = \begin{cases} a^1(x^1) & \text{if } x^3 = g^1_{in(t^1)}(x^1) \\ a^2(x^2) & \text{if } x^3 = g^2_{in(t^2)}(x^2) \end{cases}, \\
& b^0 = b^1 \circ f^1_{out(t^0)} = b^2 \circ f^2_{out(t^0)}, \\
& b^3(x^3) = \begin{cases} b^1(x^1) & \text{if } x^3 = g^1_{out(t^1)}(x^1) \\ b^2(x^2) & \text{if } x^3 = g^2_{out(t^2)}(x^2) \end{cases} \qquad \}
\end{array}
$$

The lifting of transformations, which stem from only one of the sides, is done loosely, which means that in the added part of the data states all possible changes are allowed to occur. This is necessary to get the amalgamation of process occurrences in the next subsection.

Definition 3.1.9 (Lifting of Transformations)
Given a union of transformation signatures with a transformation symbol t^1 in Σ^1_{Tr}, which is not common, i. e. it is not image of a transformation symbol in the common part, and a corresponding transformation $t^1_{M^1}$, the *lifted transformation* $t^3_{M^3}$ for the image transformation symbol t^3 in Σ^3_{Tr} is obtained by taking all transformation steps for which there exists a transformation step in $t^1_{M^1}$ with the forgetful functor images of the data states and the same parameters.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{For } t^3 = g^1_{Tr}(t^1) \text{ and } \neg\exists t^2 \colon t^3 = g^2_{Tr}(t^2):} \\
\multicolumn{2}{l}{t^3_{M^3} = (t^1_{M^1})^{\#} =} \\
\{t^3_{A^3,B^3}(a^1,b^1) \mid & t^1_{A^1,B^1}(a^1,b^1) \in t^1_{M^1}, \\
& A^1 = V_{g^1_{DS}}(A^3), B^1 = V_{g^1_{DS}}(B^3) \quad \}
\end{array}
$$

Now, the amalgamation of transformation models is just constructed by the amalgamation of common transformations and the lifting of transformations, which stem from only one of the sides.

Definition 3.1.10 (Amalgamation of Transformation Models)
Given a union of transformation signatures, a Σ^1_{Tr}-model M^1_{Tr} and a Σ^2_{Tr}-model M^2_{Tr}, such that $Restr_{f^1_{Tr}}(M^1_{Tr}) = M^0_{Tr} = Restr_{f^2_{Tr}}(M^2_{Tr})$, the *amalgamation* of M^1_{Tr} and M^2_{Tr} over M^0_{Tr} is a Σ^3_{Tr}-model $M^1_{Tr} +_{M^0_{Tr}} M^2_{Tr} = M^3_{Tr}$, which consists of amalgamated transformations for the common transformation symbols and lifted transformations for the transformation symbols from only one of M^1_{Tr} and M^2_{Tr}.

$$
\begin{array}{l}
M^3_{Tr} = (t^3_{M^3})_{t^3 \in \Sigma^3_{Tr}} \\
\text{with} \quad t^3_{M^3} = \begin{cases} t^1_{M^1} +_{t^0_{M^0}} t^2_{M^2} & \text{if } t^3 = g^1_{Tr}(t^1) = g^2_{Tr}(t^2) \\ (t^1_{M^1})^{\#} & \text{if } t^3 = g^1_{Tr}(t^1) \wedge \neg\exists t^2 \colon t^3 = g^2_{Tr}(t^2) \\ (t^2_{M^2})^{\#} & \text{if } t^3 = g^2_{Tr}(t^2) \wedge \neg\exists t^1 \colon t^3 = g^1_{Tr}(t^1) \end{cases}
\end{array}
$$

Again, this construction can be extended to homomorphisms, where the existence of the homomorphism, uniquely determined by the homomorphism on the data state layer, is guaranteed, because for amalgamated transformations all homomorphism images are also amalgamated and, hence, exist in the amalgamation.

For the transformations from one of the sides, the lifting construction, which includes all transformation steps being restricted to transformation steps on the corresponding side into the amalgamation, also does this for the homomorphism images.

Definition 3.1.11 (Amalgamation of Transformation Homomorph.)
Given a union of transformation signatures, a Σ^1_{Tr}-homomorphism $h^1_{Tr}\colon M^1_{Tr} \to N^1_{Tr}$ and a Σ^2_{Tr}-homomorphism $h^2_{Tr}\colon M^2_{Tr} \to N^2_{Tr}$, so that $Restr_{f^1_{Tr}}(h^1_{Tr}) = h^0_{Tr} = Restr_{f^2_{Tr}}(h^2_{Tr})$, the *amalgamation* of h^1_{Tr} and h^2_{Tr} over h^0_{Tr} is a Σ^3_{Tr}-homomorphism $h^1_{Tr} +_{h^0_{Tr}} h^2_{Tr} = h^3_{Tr}$, which maps the amalgamations of transformation steps to the amalgamations of the images under the homomorphisms.

$$\begin{array}{ll} h^1_{Tr} +_{h^0_{Tr}} h^2_{Tr} = h^3_{Tr}\colon M^3_{Tr} \to N^3_{Tr} & \\ \text{with} & M^3_{Tr} = M^1_{Tr} +_{M^0_{Tr}} M^2_{Tr}, N^3_{Tr} = N^1_{Tr} +_{N^0_{Tr}} N^2_{Tr} \\ & h^3_{t^3}(t^1_{A^1,B^1}(a^1,b^1) +_{t^0_{A^0,B^0}(a^0,b^0)} t^2_{A^2,B^2}(a^2,b^2)) = \\ & h^1_{t^1}(t^1_{A^1,B^1}(a^1,b^1)) +_{h^0_{t^0}(t^0_{A^0,B^0}(a^0,b^0))} h^2_{t^2}(t^2_{A^2,B^2}(a^2,b^2)) \end{array}$$

3.1.4 Process Layer

The union of process signatures is constructed like the union of transformation signatures, since they are structured identically.

Definition 3.1.12 (Union of Process Signatures)
Given a union of data state signatures and corresponding process signatures (Σ^0_{Pr}, in, out), (Σ^1_{Pr}, in, out) and (Σ^2_{Pr}, in, out) with morphisms f^1_{Pr} and f^2_{Pr}, the *union* (Σ^3_{Pr}, in, out) is constructed by a pushout $\Sigma^3_{Pr} = \Sigma^1_{Pr} \cup_{\Sigma^0_{Pr}} \Sigma^2_{Pr}$ in the category **Set** for the process symbols, where the parameters are also obtained by pushouts $in(p^3) = in(p^1) \cup_{in(p^0)} in(p^2)$ for process symbols in the common part. For process symbols not in the common part, the sorts of the parameters are just translated according to the data state layer.

Amalgamation of process expressions, amalgamation of common processes and lifting of processes is done exactly as for transformations. The RSTSs, which are identical for common process symbols, since the restriction is the same, are just included identically into the amalgamated process model.

For occurences we again choose a loose amalgamation, where all graph morphism, which have the corresponding restrictions, are included into the occurence sets. More precisely, in the case of common process symbols for each pair of occurences, whose restrictions to the common part are the same, all occurence morphisms, whose restrictions are these occurences are included. For process symbols from one of the sides the occurences are lifted.

Definition 3.1.13 (Amalgamation of Process Models)
Given a union of process signatures, a Σ^1_{Pr}-model M^1_{Pr} and a Σ^2_{Pr}-model M^2_{Pr}, such that $Restr_{f^1_{Pr}}(M^1_{Pr}) = M^0_{Pr} = Restr_{f^2_{Pr}}(M^2_{Pr})$, the *amalgamation* of M^1_{Pr} and M^2_{Pr} over M^0_{Pr} is a Σ^3_{Pr}-model $M^1_{Pr} +_{M^0_{Pr}} M^2_{Pr} = M^3_{Pr}$, which consists of the RSTSs for the corresponding symbols in M^1_{Pr} and M^2_{Pr}, processes constructed similar to transformations and occurrence sets, which are constructed such that the restriction to the sides yields the original sets (see Fig. 3.3).

$$
\begin{array}{l}
M^3_{Pr} = (M^3_{p^3}, p^3_{M^3}, Occ_{M^3_{p^3}})_{p^3 \in \Sigma^3_{Tr}} \\
\text{with} \quad M^3_{p^3} = \begin{cases} M^1_{p^1} & \text{if } p^3 = g^1_{Pr}(p^1) \\ M^2_{p^2} & \text{if } p^3 = g^2_{Pr}(p^2) \end{cases} \\
\qquad\quad p^3_{M^3} = \begin{cases} p^1_{M^1} +_{p^0_{M^0}} p^2_{M^2} & \text{if } p^3 = g^1_{Pr}(p^1) = g^2_{Pr}(p^2) \\ (p^1_{M^1})^{\#} & \text{if } p^3 = g^1_{Pr}(p^1) \wedge \neg\exists p^2 \colon p^3 = g^2_{Pr}(p^2) \\ (p^2_{M^2})^{\#} & \text{if } p^3 = g^2_{Pr}(p^2) \wedge \neg\exists p^1 \colon p^3 = g^1_{Pr}(p^1) \end{cases} \\
\qquad\quad Occ_{M^3_{p^3}}(p^3_{A,B}(a,b)) = \begin{cases}
\{\, occ^3 \colon G_{M^3_{p^3}} \to M^3_{DSTS} \mid \\
\quad \exists occ^1 \in Occ_{M^1_{p^1}}(R_{g^1_{Pr},p^1}(p^3_{A,B}(a,b))), \\
\quad \exists occ^2 \in Occ_{M^2_{p^2}}(R_{g^2_{Pr},p^2}(p^3_{A,B}(a,b))) \colon \\
\quad f^1_{M^1_{DSTS}} \circ occ^1 = f^2_{M^2_{DSTS}} \circ occ^2, \\
\quad occ^1 = g^1_{M^3_{DSTS}} \circ occ^3, \\
\quad occ^2 = g^2_{M^3_{DSTS}} \circ occ^3 \,\} \\
\qquad \text{if } p^3 = g^1_{Pr}(p^1) = g^2_{Pr}(p^2) \\
\{\, occ^3 \colon G_{M^3_{p^3}} \to M^3_{DSTS} \mid \\
\quad \exists occ^1 \in Occ_{M^1_{p^1}}(R_{g^1_{Pr},p^1}(p^3_{A,B}(a,b))) \colon \\
\quad occ^1 = g^1_{M^3_{DSTS}} \circ occ^3 \,\} \\
\qquad \text{if } p^3 = g^1_{Pr}(p^1) \wedge \neg\exists p^2 \colon p^3 = g^2_{Pr}(p^2) \\
\{\, occ^3 \colon G_{M^3_{p^3}} \to M^3_{DSTS} \mid \\
\quad \exists occ^2 \in Occ_{M^2_{p^2}}(R_{g^2_{Pr},p^2}(p^3_{A,B}(a,b))) \colon \\
\quad occ^2 = g^2_{M^3_{DSTS}} \circ occ^3 \,\} \\
\qquad \text{if } p^3 = g^2_{Pr}(p^2) \wedge \neg\exists p^1 \colon p^3 = g^1_{Pr}(p^1)
\end{cases}
\end{array}
$$

$$
\begin{tikzcd}
G_{M^1_{p^1}} \arrow[r, equal] \arrow[d, "occ^1"'] \arrow[dr, phantom, "(=)"] & G_{M^3_{p^3}} \arrow[r, equal] \arrow[d, "occ^3"] \arrow[dr, phantom, "(=)"] & G_{M^2_{p^2}} \arrow[d, "occ^2"] \\
M^1_{DSTS} \arrow[dr, "f^1_{M^1_{DSTS}}"'] & M^3_{DSTS} \arrow[l, "g^1_{M^3_{DSTS}}"'] \arrow[r, "g^2_{M^3_{DSTS}}"] \arrow[d, phantom, "(=)"] & M^2_{DSTS} \arrow[dl, "f^2_{M^2_{DSTS}}"] \\
& M^0_{DSTS} &
\end{tikzcd}
$$

Figure 3.3: Amalgamation of process occurrences

Definition 3.1.14 (Amalgamation of Process Homomorphisms)
Given a union of process signatures, a Σ^1_{Pr}-homomorphism $h^1_{Pr} \colon M^1_{Pr} \to N^1_{Pr}$ and a Σ^2_{Pr}-homomorphism $h^2_{Pr} \colon M^2_{Pr} \to N^2_{Pr}$, so that $Restr_{f^1_{Pr}}(h^1_{Pr}) = h^0_{Pr} =$

$Restr_{f^2_{Pr}}(h^2_{Pr})$, the *amalgamation* of h^1_{Pr} and h^2_{Pr} over h^0_{Pr} is a Σ^3_{Pr}-homomorphism $h^1_{Pr} +_{h^0_{Pr}} h^2_{Pr} = h^3_{Pr}$, which contains the graph homomorphisms of h^1_{Pr} and h^2_{Pr} and mappings of the amalgamations of process expressions to the amalgamations of the images under the homomorphisms.

$$
\begin{array}{ll}
h^1_{Pr} +_{h^0_{Pr}} h^2_{Pr} = h^3_{Pr}: M^3_{Pr} \to N^3_{Pr} & \\
\text{with} & M^3_{Pr} = M^1_{Pr} +_{M^0_{Pr}} M^2_{Pr}, N^3_{Pr} = N^1_{Pr} +_{N^0_{Pr}} N^2_{Pr} \\
& h^3_{G_{p^3}} = \begin{cases} h^1_{G_{p^1}} & \text{if } p^3 = g^1_{Pr}(p^1) \\ h^2_{G_{p^2}} & \text{if } p^3 = g^2_{Pr}(p^2) \end{cases} \\
& h^3_{p^3}(p^1_{A^1,B^1}(a^1,b^1) +_{p^0_{A^0,B^0}(a^0,b^0)} p^2_{A^2,B^2}(a^2,b^2)) = \\
& h^1_{p^1}(p^1_{A^1,B^1}(a^1,b^1)) +_{h^0_{p^0}(p^0_{A^0,B^0}(a^0,b^0))} h^2_{p^2}(p^2_{A^2,B^2}(a^2,b^2))
\end{array}
$$

3.1.5 Integration

This time we do not define the union of integrated signatures in terms of the underlying layers, but characterize it as a pushout in **IntSig**.

Definition 3.1.15 (Union of Integrated Signatures)
The *union* of integrated signatures Σ^0, Σ^1 and Σ^2 with signature morphisms $f^1: \Sigma^0 \to \Sigma^1$ and $f^2: \Sigma^0 \to \Sigma^2$ is a pushout object $\Sigma^1 \cup_{\Sigma^0} \Sigma^2 = \Sigma^3$ with morphisms $g^1: \Sigma^1 \to \Sigma^3$ and $g^2: \Sigma^2 \to \Sigma^3$, which can be constructed by corresponding unions on the layers.

The amalgamation of integrated models and model homomorphisms can then again be defined layer-wise.

Definition 3.1.16 (Amalgamation of Integrated Models)
Given a union of integrated signatures and integrated models $M^1 \in Mod(\Sigma^1)$ and $M^2 \in Mod(\Sigma^2)$, such that $Restr_{f^1}(M^1) = M^0 = Restr_{f^2}(M^2)$, the *amalgamation* of M^1 and M^2 over M^0 is a Σ^3-model $M^1 +_{M^0} M^2 = M^3$, which is constructed by

- the data type amalgamation $M^3_{DT} = M^1_{DT} +_{M^0_{DT}} M^2_{DT}$ as in Def. 3.1.2,
- the data state amalgamation $M^3_{DS} = M^1_{DS} +_{M^0_{DS}} M^2_{DS}$ as in Def. 3.1.5,
- the transformation model amalgamation $M^3_{Tr} = M^1_{Tr} +_{M^0_{Tr}} M^2_{Tr}$ as in Def. 3.1.10 and
- the process model amalgamation $M^3_{Pr} = M^1_{Pr} +_{M^0_{Pr}} M^2_{Pr}$ as in Def. 3.1.13.

Definition 3.1.17 (Amalgamation of Integrated Homomorphisms)
Given a union of integrated signatures and integrated model homomorphisms $h^1: M^1 \to N^1$ and $h_2: M^2 \to N^2$, such that $Restr_{f^1}(h^1) = h^0 = Restr_{f^2}(h^2)$, the *amalgamation* of h^1 and h^2 over h^0 is a Σ^3-homomorphism $h^1 +_{h^0} h^2 = h^3$, which is constructed by

- the data type homomorphism amalgamation $h^3_{DT} = h^1_{DT} +_{h^0_{DT}} h^2_{DT}$ as in Def. 3.1.3,

- the data state homomorphism amalgamation $h^3_{DS} = h^1_{DS} +_{h^0_{DS}} h^2_{DS}$ as in Def. 3.1.6,
- the transformation homomorphism amalgamation $h^3_{Tr} = h^1_{Tr} +_{h^0_{Tr}} h^2_{Tr}$ as in Def. 3.1.11 and
- the process homomorphism amalgamation $h^3_{Pr} = h^1_{Pr} +_{h^0_{Pr}} h^2_{Pr}$ as in Def. 3.1.14.

The essential property of amalgamations of integrated models is summarized in the following weak amalgamation theorem.

Theorem 3.1.18 (Weak Amalgamation Theorem)
Given a union of integrated signatures and integrated models $M^1 \in Mod(\Sigma^1)$ and $M^2 \in Mod(\Sigma^2)$, such that $Restr_{f^1}(M^1) = M^0 = Restr_{f^2}(M^2)$, the amalgamation fullfills the following *amalgamation property*:

$$Restr_{g^1}(M^1 +_{M^0} M^2) = M^1 \text{ and } Restr_{g^2}(M^1 +_{M^0} M^2) = M^2$$

The theorem is weaker then the notions of amalgamation in [EM85, EGR94], because we are not able to guarantee the decomposition property of (strong) amalgamations, $M^3 = Restr_{g^1}(M^3) +_{M^0} Restr_{g^2}(M^3)$, since M^3 could be a model, where not all of the data states and transformation steps of the (maximal) model $Restr_{g^1}(M^3) +_{M^0} Restr_{g^2}(M^3)$ are present, as already mentioned in Subsections 3.1.2 and 3.1.3.

3.2 Signature Refinements and Abstraction of Models

Signature refinements are a more general class of signature morphisms and are again interpreted by forgetful functors on model categories in the opposite direction, which we will call abstractions of models and homomorphisms.

3.2.1 Data Type Layer

Refinements are allowed to map operation symbols not only to other operation sybols, but also to terms with corresponding variables. Therefore, we first have to define terms for algebraic signatures, which are build over families of variables.

Definition 3.2.1 (Terms for an Algebraic Signature)
Given an algebraic signature Σ and a family $X = (X_s)_{s\in\Sigma_S}$ of variables, the family $T_\Sigma(X) = (T_\Sigma(X)_s)_{s\in\Sigma_S\cup\{\lambda\}}$ of *terms* over Σ is recursively defined by:

$$\begin{array}{l} X_s \subseteq T_\Sigma(X)_s \\ op \in \Sigma_{Op}, dom(op) = s_1 \dots s_n, cod(op) = s, \\ t_1 \in T_\Sigma(X)_{s_1}, \dots, t_n \in T_\Sigma(X)_{s_n} \implies op(t_1, \dots, t_n) \in T_\Sigma(X)_s \end{array}$$

Note, that this definition subsumes the special cases of constant symbols, which are initially included in the corresponding term set, and predicate symbols, constituting terms in $T_\Sigma(X)_\lambda$, which can not be used in the construction of other terms.

We will denote by $Var(t)$ the family of variables used in the term t.

Normally, e. g. in [EM85], an algebra of terms with trivial operations is constructed, which can be used to characterize the evaluation as (initial) homomorphisms. Since evaluation will be partial and our homomorphisms are not, we do not apply this categorical characterization here.

For sorts we allow refinements to map static sorts to dynamic ones. This feature is useful, if changes to a static sort become necessary during system development.

The terms assigned to static operation symbols, however, may not contain dynamic operation symbols, which leads to the restriction that no operation symbols are allowed for static sorts refined by dynamic ones.

Note, that the terms are not required to use all parameters given to the operation. This can be used, when the result of an operation in the refined system model does not depend on all its parameters.

Definition 3.2.2 (Data Type Signature Refinement)
Given two data type signatures Σ_{DT} and Σ'_{DT} and a data state signature Σ'_{DS} w. r. t. Σ'_{DT}, a *data type signature refinement* $r_{DT}: \Sigma_{DT} \rightsquigarrow (\Sigma'_{DT}, \Sigma'_{DS})$ consists of a function $r_{DT,S}: \Sigma_{DT,S} \to \Sigma'_{DS,S}$ and a function $r_{DT,Op}: \Sigma_{DT,Op} \to T_{\Sigma'_{DT}}(X)$, where the variables contained in a term $r_{DT,Op}(op)$ are canonically named representants for the parameters in $dom(op)$ and the sort of the term is $cod(op)$.

$$\begin{array}{ll} \multicolumn{2}{l}{r_{DT}: \Sigma_{DT} \rightsquigarrow (\Sigma'_{DT}, \Sigma'_{DS})} \\ \text{with} & r_{DT,S}: \Sigma_{DT,S} \to \Sigma'_{DS,S} \\ & r_{DT,Op}: \Sigma_{DT,Op} \to T_{\Sigma'_{DT}}(X) \\ & r_{DT,Op}(op) \in T_{\Sigma'_{DT}}(X)_{r_{DT,S}(cod(op))} \\ & Var(r_{DT,Op}(op)) \subseteq \{x_1: r_{DT,S}(s_1), \ldots, x_n: r_{DT,S}(s_n)\} \\ & \text{for } dom(op) = s_1 \ldots s_n \end{array}$$

To interpret terms we introduce the evaluation of terms for a given partial algebra, which is a partial function from the terms of a sort symbol into the corresponding carrier set.

Definition 3.2.3 (Evaluation of Terms)
Given an algebraic signature Σ, variables X, a Σ-algebra A and an assignment $asg: X \to A$, the *evaluation* of terms is a family $\overline{asg}: T_\Sigma(X) \to A$ of functions defined recursively by:

$$\begin{array}{l} \overline{asg}_s(x) = asg_s(x) \text{ for } x \in X_s \\ \overline{asg}_s(op(t_1, \ldots, t_n)) = op_A(\overline{asg}_{s_1}(t_1), \ldots, \overline{asg}_{s_n}(t_n)) \\ \ \text{for } dom(op) = s_1 \ldots s_n, cod(op) = s \end{array}$$

Similar to the domain $Dom(op_A)$ of an operation we denote by $Dom(t, A)$ the set of all assignments, for which the term t is defined.

On the model side the assignment of a dynamic sort symbol to a static sort symbol is interpreted by taking the union of the carrier sets in all data states as abstracted carrier sets. Since the static sorts are included in the data states we can subsume the statically refined sort symbols as special cases, where constructing the union over all data states, does not change anything and results in the carrier sets of the data type.

For operation symbols the application of the operation in the abstracted model is interpreted by the evaluation of the term, where the parameters given to the operation are assigned to the corresponding variables.

Definition 3.2.4 (Data Type Model Abstraction)
Given a data type signature refinement $r_{DT}: \Sigma_{DT} \rightsquigarrow (\Sigma'_{DT}, \Sigma'_{DS})$, a Σ'_{DT}-model M'_{DT} and a corresponding Σ'_{DS}-model M'_{DS}, the *abstraction* of (M'_{DT}, M'_{DS}) is a Σ_{DT}-model $Abstr_{r_{DT}}(M'_{DT}, M'_{DS}) = M_{DT}$, where the carrier sets are given by the union of the corresponding carrier sets in M'_{DS} and the operations by the evaluation of the correponding terms in M'_{DT}.

$$
\begin{aligned}
& Abstr_{r_{DT}}(M'_{DT}, M'_{DS}) = M_{DT} \\
& \text{with} \quad M_s = \textstyle\bigcup_{A' \in M'_{DS}} A'_{r_{DT,S}(s)} \\
& \qquad\quad op_M(m_1, \ldots, m_n) = \overline{asg}(r_{DT,Op}(op)) \\
& \qquad\quad asg: Var(r_{DT,Op}(op)) \rightarrow M'_{DT} = \{x_1 \mapsto m_1, \ldots, x_n \mapsto m_n\}
\end{aligned}
$$

Strictly speaking, the data state model is allowed to be a class, not only a set, of data states, and the union of a class of sets will not be a proper carrier set. We could avoid this problem by assuming a universe set U_s for each dynamic sort s, which shall be abstracted to a static one, such that the carrier sets A_s for all data states are subsets of U_s.

For abstracted data type homomorphisms we get a construction similar to restriction for the statically refined sorts, while for sort symbols refined by dynamic ones, we get an inclusion of the carrier sets, because the requirement, that dynamic sorts and their images under a homomorphism have to be identical, ensures that an abstracted sort in the second model includes at least all elements of the corresponding abstracted sort in the first model.

Again, we just take the union of all homomorphism functions between the data state carrier sets, because this subsumes the data type homomorphisms as special cases.

Definition 3.2.5 (Data Type Homomorph. Abstraction)
Given a data type signature refinement $r_{DT}: \Sigma_{DT} \rightsquigarrow (\Sigma'_{DT}, \Sigma'_{DS})$ and a Σ'_{DS}-homomorphism $h'_{DS}: M'_{DS} \rightarrow N'_{DS}$, the *abstraction* of h'_{DS} is a Σ_{DT}-homomorphism $Abstr_{f_{DT}}(h'_{DS}) = h_{DT}: M_{DT} \rightarrow N_{DT}$, where the functions h_s are given by the unions of the corresponding functions for all data states in M'_{DS}.

$$
\begin{aligned}
& Abstr_{f_{DT}}(h'_{DS}: M'_{DS} \rightarrow N'_{DS}) = h_{DT}: M_{DT} \rightarrow N_{DT} \\
& \text{where} \quad h_s = \textstyle\bigcup_{A' \in M'_{DS}} h_{A', r_{DT,S}(S)}
\end{aligned}
$$

The abstraction is compatible with composition and identities:

$$
\begin{aligned}
& \qquad\quad Abstr_{f_{DT}}(h'_{DT} \circ g'_{DT}) = Abstr_{f_{DT}}(h'_{DT}) \circ Abstr_{f_{DT}}(g'_{DT}) \\
& \text{and} \quad Abstr_{f_{DT}}(id_{M'_{DT}}) = id_{Abstr_{f_{DT}}(M'_{DT})}
\end{aligned}
$$

3.2.2 Data State Layer

On the data state layer we again allow the assignment of terms to operation symbols. For sort symbols, however, we do not have the possibility to generalize the morphism notion, because there are no sorts other than the ones of Σ'_{DS} in Σ'.

Definition 3.2.6 (Data State Signature Refinement)
Given a data type signature refinement $r_{DT}\colon \Sigma_{DT} \rightsquigarrow (\Sigma'_{DT}, \Sigma'_{DS})$ and a data state signature Σ_{DS} w.r.t. Σ_{DT}, a *data state signature refinement* $r_{DS}\colon \Sigma_{DS} \rightsquigarrow \Sigma'_{DS}$ consists of a function $r_{DS,S}\colon \Sigma_{DS,S} \to \Sigma'_{DS,S}$ and a function $r_{DS,Op}\colon \Sigma_{DS,Op} \to T_{\Sigma'_{DS}}(X)$, such that r_{DS} is an extension of r_{DT}.

$$
\begin{array}{ll}
\multicolumn{2}{l}{r_{DS}\colon \Sigma_{DS} \rightsquigarrow \Sigma'_{DS}} \\
\text{with} & r_{DS,S}\colon \Sigma_{DS,S} \to \Sigma'_{DS,S} \\
& r_{DS,Op}\colon \Sigma_{DS,Op} \to T_{\Sigma'_{DS}}(X) \\
& r_{DS,Op}(op) \in T_{\Sigma'_{DS}}(X)_{r_{DS,S}(cod(op))} \\
& Var(r_{DS,Op}(op)) \subseteq \{x_1\colon r_{DS,S}(s_1), \ldots, x_n\colon r_{DS,S}(s_n)\} \\
& \text{for } dom(op) = s_1 \ldots s_n \\
& r_{DT} \subseteq r_{DS}
\end{array}
$$

For the abstraction of data states we cannot take the abstraction construction of the data type layer, because it would unify the sorts of all data states, even for sorts which are still dynamic in the abstracted model. Instead, we have to distinguish between static and dynamic sorts, and take the union over all data states only for sorts that are static in the abstracted model, but just the sort of the particular data state for sorts that stay dynamic.

Note that, if a dynamic sort symbol is assigned to a static and a dynamic one, this results in two different sorts in the abstracted model, one that is static and contains all elements included in any data state, and one that is dynamic and contains only the elements of the corresponding data state in the concrete model.

For dynamic operations whose dynamic domain or codomain is abstracted to a static one, the abstracted domains and codomains include the original ones, there are just additional elements from other data states which are unreachable for codomains and for which the operation is undefined in the domain case.

Definition 3.2.7 (Data State Abstraction)
Given a data state signature refinement $r_{DS}\colon \Sigma_{DS} \rightsquigarrow \Sigma'_{DS}$ and a Σ'_{DS}-model M'_{DS}, the *abstraction* $A_{r_{DS}}(A') = A$ of a data state $A' \in M'_{DS}$ is a data state for Σ_{DS} w.r.t. the abstracted data type M_{DT} constructed by taking the union of the carrier sets in all data states for static sorts and just the carrier sets of A' for dynamic sorts.

$$
\begin{array}{ll}
\multicolumn{2}{l}{A_{r_{DS}}(A') = A} \\
\text{with} & A_s = \bigcup_{X' \in M'_{DS}} X'_{r_{DT,S}(s)} \text{ for } s \in \Sigma_{DT,S} \\
& A_s = A'_{r_{DS,S}} \text{ for } s \in \Sigma_{DS,S} \setminus \Sigma_{DT,S} \\
& op_A(a_1, \ldots, a_n) = \overline{asg}(r_{DS,Op}(op)) \\
& asg\colon Var(r_{DS,Op}(op)) \to A' = \{x_1 \mapsto m_1, \ldots, x_n \mapsto m_n\}
\end{array}
$$

Using this definition we can now abstract the whole data state model similar to the restriction by applying the above abstraction to all data states.

Definition 3.2.8 (Data State Model Abstraction)
Given a data state signature refinement $r_{DS}: \Sigma_{DS} \rightsquigarrow \Sigma'_{DS}$ and a Σ'_{DS}-model M'_{DS}, the *abstraction* of M'_{DS} is a Σ_{DS}-model $Abstr_{r_{DS}}(M'_{DS}) = M_{DS}$ consisting of all abstractions for the data states in M'_{DS}.

$$\begin{array}{l} Abstr_{r_{DS}}(M'_{DS}) = M_{DS} \\ \text{with} \quad M_{DS} = \{\, A_{r_{DS}}(A') \mid A' \in M'_{DS} \,\} \end{array}$$

For data state homomorphisms there are no specialities, since the abstracted dynamic parts are also dynamic in the concrete model, which is why they meet all requirements for data state homomorphisms.

Definition 3.2.9 (Data State Homomorph. Abstraction)
Given a data state signature refinement $r_{DS}: \Sigma_{DS} \rightsquigarrow \Sigma'_{DS}$ and a Σ'_{DS}-homomorphism $h'_{DS}: M'_{DS} \to N'_{DS}$, the *abstraction* of h'_{DS} is a Σ_{DS}-homomorphism $Abstr_{r_{DS}}(h'_{DS}) = h_{DS}: M_{DS} \to N_{DS}$ between $M_{DS} = Abstr_{f_{DS}}(M'_{DS})$ and $N_{DS} = Abstr_{f_{DS}}(N'_{DS})$ assigning to the abstraction $A_{r_{DS}}(A')$ of a data state A' the forgetful functor image of $h'_{DS}(A')$.

$$\begin{array}{ll} \multicolumn{2}{l}{Abstr_{r_{DS}}(h'_{DS}: M'_{DS} \to N'_{DS}) = h_{DS}: M_{DS} \to N_{DS}} \\ \text{with} & M_{DS} = Abstr_{r_{DS}}(M'_{DS}), N_{DS} = Abstr_{r_{DS}}(N'_{DS}) \\ & h_{DS}(Abstr_{r_{DS}}(A')) = Abstr_{r_{DS}}(h'_{DS}(A')) \end{array}$$

The restriction is compatible with composition and identities:

$$\begin{array}{ll} & Abstr_{r_{DS}}(h'_{DS} \circ g'_{DS}) = Abstr_{r_{DS}}(h'_{DS}) \circ Abstr_{r_{DS}}(g'_{DS}) \\ \text{and} & Abstr_{r_{DS}}(id_{M'_{DS}}) = id_{Abstr_{r_{DS}}(M'_{DS})} \end{array}$$

3.2.3 Transformation Layer

On the transformation layer transformation symbols can not only be refined by transformation symbols, but also by process symbols. This allows to add an inner process structure to specify how the effect of the abstract transformation is reached in terms of more concrete transformations.

Definition 3.2.10 (Transformation Signature Refinement)
Given a data state signature refinement $f_{DS}: \Sigma_{DS} \rightsquigarrow \Sigma'_{DS}$ and corresponding transformation and process signatures (Σ_{Tr}, in, out), (Σ'_{Tr}, in, out) and (Σ'_{Pr}, in, out), a *transformation signature refinement* $f_{Tr}: (\Sigma_{Tr}, in, out) \rightsquigarrow (\Sigma'_{Tr}, \Sigma'_{Pr}, in, out)$ is a triple $(f_{Tr}, f_{in}, f_{out})$ with a function $f_{Tr}: \Sigma_{Tr} \to \Sigma'_{Tr}$, which assigns to each transformation symbol $t \in \Sigma_{Tr}$ a transformation or a process symbol $r_{Tr}(t) \in \Sigma'_{Tr} \cup \Sigma'_{Pr}$, and two families f_{in} and f_{out} of function families $f_{in(t)}: in(t) \to in(f_{Tr}(t))$ and $f_{out(t)}: out(t) \to out(f_{Tr}(t))$ assigning parameters of $f_{Tr}(t)$ with matching type to the parameters of each transformation symbol

$t \in \Sigma_{Tr}$.

$$
\begin{array}{ll}
f_{Tr}\colon (\Sigma_{Tr}, in, out) \rightsquigarrow (\Sigma'_{Tr}, \Sigma'_{Pr}, in, out) = (f_{Tr}, f_{in}, f_{out}) & \\
\text{with} \quad f_{Tr}\colon \Sigma_{Tr} \to \Sigma'_{Tr} \cup \Sigma'_{Pr} & \\
\qquad (f_{in(t)}\colon in(t) \to in(f_{Tr}(t)))_{t \in \Sigma_{Tr}} & \\
\qquad f_{in(t)} = (f_{in(t),s}\colon in(t)_s \to in(f_{Tr}(t))_{f_{DS,S}(s)})_{s \in \Sigma_{DS,S}} & \\
\qquad (f_{out(t)}\colon out(t) \to out(f_{Tr}(t)))_{t \in \Sigma_{Tr}} & \\
\qquad f_{out(t)} = (f_{out(t),s}\colon out(t)_s \to out(f_{Tr}(t))_{f_{DS,S}(s)})_{s \in \Sigma_{DS,S}} &
\end{array}
$$

The (inner) abstraction of transformation steps is obtained by generalizing the restriction construction to using abstractions of the source and target data state, instead of just restrictions.

Definition 3.2.11 (Transformation Expr. Abstraction)
Given a transformation signature refinement $r_{Tr}\colon (\Sigma_{Tr}, in, out) \rightsquigarrow (\Sigma'_{Tr}, \Sigma'_{Pr}, in, out)$, a transformation symbol $t \in \Sigma_{Tr}$ with $r_{Tr}(t) = t'$ and a data state model M'_{DS} for Σ'_{DS}, the *tranformation abstraction function* $A_{r_{Tr},t}$ assigns to each transformation or process expression for t' and M'_{DS} a transformation expression for t and $Abstr_{f_{DS}}(M'_{DS})$ defined by the abstraction on the data states and concatenation with the mappings for the parameters.

$$
\begin{array}{l}
A_{r_{Tr},t}\colon TrExpr(M'_{DS}, t') \to TrExpr(Abstr_{r_{DS}}(M'_{DS}), t) \text{ resp.} \\
A_{r_{Tr},t}\colon PrExpr(M'_{DS}, t') \to TrExpr(Abstr_{r_{DS}}(M'_{DS}), t) \\
\text{with} \quad A_{r_{Tr},t}(t'_{A',B'}(a', b')) = t_{A_{r_{DS}}(A'), A_{r_{DS}}(B')}(a' \circ r_{in(t)}, b' \circ r_{out(t)})
\end{array}
$$

Abstracted transformations are constructed very similar to restricted transformations, where, if a process symbol is assigned to a transformation symbol, the process structure consisting of the RSTS and the occurences is just forgotten.

Definition 3.2.12 (Transformation Model Abstraction)
Given a transformation signature refinement $r_{Tr}\colon (\Sigma_{Tr}, in, out) \rightsquigarrow (\Sigma'_{Tr}, \Sigma'_{Pr}, in, out)$, a Σ'_{Tr}-model M'_{Tr} and a Σ'_{Pr}-model M'_{Pr}, the *abstraction* of (M'_{Tr}, M'_{Pr}) is a Σ_{Tr}-model $Abstr_{r_{Tr}}(M'_{Tr}, M'_{Pr}) = M_{Tr}$, which contains for each transformation symbol $t \in \Sigma_{Tr}$ the abstractions of all transformation steps of $r_{Tr}(t)_{M'}$.

$$
\begin{array}{l}
Abstr_{r_{Tr}}(M'_{Tr}, M'_{Pr}) = M_{Tr} = (t_M)_{t \in \Sigma_{Tr}} \\
\text{with} \quad t_M = \{\, A_{r_{Tr},t}(t'_{A',B'}(a', b')) \mid t'_{A',B'}(a', b') \in r_{Tr}(t)_{M'} \,\}
\end{array}
$$

The abstraction, like the restriction, induces a graph homomorphism between the DSTSs, where transformation sets containing transformation steps generated by abstraction of process expressions are unreachable, since there are no corresponding transformation steps in the concrete model.

Definition 3.2.13 (Abstraction-Induced DSTS-Homomorph.)
Given a data state signature refinement $r_{DS}\colon \Sigma_{DS} \rightsquigarrow \Sigma'_{DS}$, a transformation signature refinement $r_{Tr}\colon (\Sigma_{Tr}, in, out) \rightsquigarrow (\Sigma'_{Tr}, \Sigma'_{Pr}, in, out)$ and a Σ'-DSTS M'_{DSTS} and the corresponding Σ-DSTS M_{DSTS}, the graph homomorphism

$r_{M'_{DSTS}}: M'_{DSTS} \to M_{DSTS}$ is given by the data state abstraction $A_{r_{DS}}$ on nodes and the extension of the transformation abstraction $A_{r_{Tr}}$ to sets on edges.

$$\begin{array}{ll} r_{M'_{DSTS}}: M'_{DSTS} \to M_{DSTS} & \\ \text{with} \quad r_{M'_{DSTS}}(A') = A_{r_{DS}}(A') & \\ \qquad r_{M'_{DSTS}}(T'_{A',B'}) = \{ \quad A_{r_{Tr},t}(t'_{A',B'}(a',b')) \mid & \\ \qquad\qquad r_{Tr}(t) = t', t'_{A',B'}(a',b') \in T'_{A',B'} \quad \} & \end{array}$$

Abstracted transformation homomorphisms are again similar to restricted transformation homomorphisms, because the corresponding functions between the expression sets exist for transformations as well as for processes.

Definition 3.2.14 (Transformation Hom. Abstraction)
Given a transformation signature refinement $r_{Tr}: (\Sigma_{Tr}, in, out) \rightsquigarrow (\Sigma'_{Tr}, \Sigma'_{Pr}, in, out)$, a Σ'_{Tr}-homomorphism $h'_{Tr}: M'_{Tr} \to N'_{Tr}$ and a Σ'_{Pr}-homomorphism $h'_{Pr}: M'_{Pr} \to N'_{Pr}$, the *abstraction* of (h'_{Tr}, h'_{Pr}) is a Σ_{Tr}-trafo-homomorphism $Abstr_{r_{Tr}}(h'_{Tr}, h'_{Pr}) = h_{Tr}: M_{Tr} \to N_{Tr}$ between $M_{Tr} = Abstr_{r_{Tr}}(M'_{Tr}, M'_{Pr})$ and $N_{Tr} = Abstr_{r_{Tr}}(N'_{Tr}, N'_{Pr})$ assigning to the restrictions of transformation steps in M'_{Tr} the restrictions of the images under h'_{Tr}.

$$\begin{array}{l} Abstr_{r_{Tr}}(h'_{Tr}, h'_{Pr}) = h_{Tr}: M_{Tr} \to N_{Tr} \\ \text{with} \quad h_t(A_{r_{Tr},t}(t'_{A',B'}(a',b'))) = A_{r_{Tr},t}(h'_{t'}(t'_{A',B'}(a',b'))) \end{array}$$

The restriction is compatible with composition and identities:

$$\begin{array}{ll} & Abstr_{r_{Tr}}(h'_{Tr} \circ g'_{Tr}, h'_{Pr} \circ g'_{Pr}) = \\ & Abstr_{r_{Tr}}(h'_{Tr}, h'_{Pr}) \circ Abstr_{r_{Tr}}(g'_{Tr}, g'_{Pr}) \\ \text{and} & Abstr_{r_{Tr}}(id_{M'_{Tr}}, id_{M'_{Pr}}) = id_{Abstr_{r_{Tr}}(M'_{Tr}, M'_{Pr})} \end{array}$$

3.2.4 Process Layer

On the process layer refinements are exactly the same as morphisms.

Definition 3.2.15 (Process Signature Refinement)
Given a data state signature refinement $r_{DS}: \Sigma_{DS} \rightsquigarrow \Sigma'_{DS}$ and two corresponding process signatures (Σ_{Pr}, in, out) and (Σ'_{Pr}, in, out), a *process signature refinement* $r_{Pr}: (\Sigma_{Pr}, in, out) \rightsquigarrow (\Sigma'_{Pr}, in, out)$ is a triple $(r_{Pr}, r_{in}, r_{out})$ with a function $r_{Pr}: \Sigma_{Pr} \to \Sigma'_{Pr}$, which assigns to each process symbol $p \in \Sigma_{Pr}$ a process symbol $r_{Pr}(p) \in \Sigma'_{Pr}$, and two families r_{in} and r_{out} of function families $r_{in(p)}$ and $r_{out(p)}$ assigning parameters of $r_{Pr}(p)$ with matching type to the parameters of each process symbol $p \in \Sigma_{Pr}$.

$$\begin{array}{ll} r_{Pr}: (\Sigma_{Pr}, in, out) \rightsquigarrow (\Sigma'_{Pr}, in, out) = (r_{Pr}, r_{in}, r_{out}) & \\ \text{with} \quad r_{Pr}: \Sigma_{Pr} \to \Sigma'_{Pr} & \\ \qquad (r_{in(p)}: in(p) \to in(r_{Pr}(p)))_{p \in \Sigma_{Pr}} & \\ \qquad r_{in(p)} = (r_{in(p),s}: in(p)_s \to in(r_{Pr}(p))_{f_{DS,S}(s)})_{s \in \Sigma_{DS,S}} & \\ \qquad (r_{out(p)}: out(p) \to out(r_{Pr}(t)))_{p \in \Sigma_{Pr}} & \\ \qquad r_{out(p)} = (r_{out(p),s}: out(p)_s \to out(r_{Pr}(p))_{f_{DS,S}(s)})_{s \in \Sigma_{DS,S}} & \end{array}$$

Since there are no differences between process morphisms and refinements, the abstraction of process expressions and process models is similar to the restriction with the use of data state abstraction instead of restriction.

Definition 3.2.16 (Process Expression Abstraction)
Given a process signature refinement $r_{Pr}: (\Sigma_{Pr}, in, out) \rightsquigarrow (\Sigma'_{Pr}, in, out)$, a process symbol $p \in \Sigma_{Pr}$ with $r_{Pr}(p) = p'$ and a data state model M'_{DS} for Σ'_{DS}, the *process abstraction function* $A_{r_{Pr},p}$ assigns to each process expression for p' and M'_{DS} a process expression for p and $Abstr_{r_{DS}}(M'_{DS})$ defined by the abstraction on the data states and concatenation with the mappings for the parameters.

$$\begin{array}{ll} & A_{r_{Pr},p}: PrExpr(M'_{DS}, p') \to PrExpr(Restr_{r_{DS}}(M'_{DS}), p) \\ \text{with} & A_{r_{Pr},p}(p'_{A',B'}(a', b')) = p_{A_{r_{DS}}(A'), A_{r_{DS}}(B')}(a' \circ r_{in(p)}, b' \circ r_{out(p)}) \end{array}$$

Now the abstraction of process models is just the same as the restriction with restriction of process expressions replaced by abstraction.

Definition 3.2.17 (Process Model Abstraction)
Given a process signature refinement $r_{Pr}: (\Sigma_{Pr}, in, out) \rightsquigarrow (\Sigma'_{Pr}, in, out)$ and a Σ'_{Pr}-model M'_{Pr}, the *abstrsction* of M'_{Pr} is a Σ_{Pr}-model $Abstr_{f_{Pr}}(M'_{Pr}) = M_{Pr}$, where the RSTSs for the process symbols in Σ_{Pr} are given by the RSTSs for the images in Σ'_{Pr}, the processes are constructed like the transformations on the transformation layer and the occurrences are constructed by concatenating the occurrences of the image process with the induced DSTS-homomorphism.

$$\begin{array}{ll} & Abstr_{r_{Pr}}(M'_{Pr}) = M_{Pr} \\ \text{with} & M_p = M'_{r_{Pr}(p)} \\ & p_M = \{ A_{r_{Pr},p}(p'_{A',B'}(a', b')) \mid p'_{A',B'}(a', b') \in r_{Pr}(p)_{M'} \} \\ & Occ_{M_p}(p_{A,B}(a, b)) = \{ \; r_{M'_{DSTS}} \circ occ' \mid r_{Pr}(p) = p', \\ & \qquad occ' \in Occ_{M'_{p'}}(p'_{A',B'}(a', b')) \quad \} \end{array}$$

For process homomorphisms, again the notion of abstraction is similar to restriction, using abstraction instead of restriction for all notions imported from lower layers.

Definition 3.2.18 (Process Homomorphism Abstraction)
Given a process signature refinement $r_{Pr}: (\Sigma_{Pr}, in, out) \rightsquigarrow (\Sigma'_{Pr}, in, out)$ and a Σ'_{Pr}-homomorphism $h'_{Pr}: M'_{Pr} \to N'_{Pr}$, the *abstraction* of h'_{Pr} is a Σ_{Pr}-homomorphism $Abstr_{r_{Pr}}(h'_{Pr}) = h_{Pr}: M_{Pr} \to N_{Pr}$ between the abstractions $M_{Pr} = Abstr_{r_{Pr}}(M'_{Pr})$ and $N_{Pr} = Abstr_{r_{Pr}}(N'_{Pr})$. It consists of the graph homomorphisms for the image process symbols and process expression mappings constructed similar to the transformation restriction.

$$\begin{array}{ll} & Abstr_{r_{Pr}}(h'_{Pr}: M'_{Pr} \to N'_{Pr}) = h_{Pr}: M_{Pr} \to N_{Pr} \\ \text{with} & M_{Pr} = Abstr_{r_{Pr}}(M'_{Pr}), N_{Pr} = Abstr_{r_{Pr}}(N'_{Pr}) \\ & h_{G_p} = h'_{G_{r_{Pr}(p)}} \\ & h_p(A_{r_{Pr},p}(p'_{A',B'}(a', b')) = A_{r_{Pr},p}(h'_{p'}(p'_{A',B'}(a', b')) \end{array}$$

The abstraction is compatible with composition and identities:

$$\begin{array}{ll} & Abstr_{r_{Pr}}(h'_{Pr} \circ g'_{Pr}) = Abstr_{r_{Pr}}(h'_{Pr}) \circ Abstr_{r_{Pr}}(g'_{Pr}) \\ \text{and} & Abstr_{r_{Pr}}(id_{M'_{Pr}}) = id_{Abstr_{r_{Pr}}(M'_{Pr})} \end{array}$$

3.2.5 Integration

In this subsection we, once more, just combine the notions of the different layers into integrated ones, where this seems a little more necessary in this case because of the cross-layer relationships in the data type and transformation refinements and abstractions.

For the syntax we get integrated signature refinements consisting, not surprisingly, of refinements on the different layers. Note, that refinements subsume plain morphisms as special cases.

Definition 3.2.19 (Integrated Signature Refinement)
Given two integrated signatures Σ and Σ', an *integrated signature refinement* $r: \Sigma \rightsquigarrow \Sigma'$ consists of

- a data type signature refinement $r_{DT}: \Sigma_{DT} \rightsquigarrow (\Sigma'_{DT}, \Sigma'_{DS})$ as in Def. 3.2.2,
- a data state signature refinement $r_{DS}: \Sigma_{DS} \rightsquigarrow \Sigma'_{DS}$ w. r. t. r_{DT} as in Def. 3.2.6,
- a transformation signature refinement $r_{Tr}: (\Sigma_{Tr}, in, out) \rightsquigarrow (\Sigma'_{Tr}, \Sigma'_{Pr}, in, out)$ w. r. t. r_{DS} as in Def. 3.2.10 and
- a process signature refinement $r_{Pr}: (\Sigma_{Pr}, in, out) \rightsquigarrow (\Sigma'_{Pr}, in, out)$ w. r. t. r_{DS} as in Def. 3.2.15.

On the semantical side we again combine the notions presented in the previous subsections.

Definition 3.2.20 (Integrated Model Abstraction)
Given an integrated signature morphism $r: \Sigma \rightsquigarrow \Sigma'$ and a Σ'-model M' the *abstraction* of M' is a Σ-model $Abstr_r(M') = M$ consisting of

- the data type abstraction $M_{DT} = Abstr_{r_{DT}}(M'_{DT}, M'_{DS})$ as in Def. 3.2.4,
- the data state abstraction $M_{DS} = Abstr_{r_{DS}}(M'_{DS})$ as in Def. 3.2.8,
- the transformation abstraction $M_{Tr} = Abstr_{r_{Tr}}(M'_{Tr}, M'_{Pr})$ as in Def. 3.2.12 and
- the process abstraction $M_{Pr} = Abstr_{r_{Pr}}(M'_{Pr})$ as in Def. 3.2.17.

Definition 3.2.21 (Integrated Homomorphism Abstraction)
Given an integrated signature refinement $r: \Sigma \rightsquigarrow \Sigma'$ and a Σ'-homomorphism $h': M' \to N'$ the *abstraction* of h' is a Σ-homomorphism $Abstr_r(h') = h: M \to N$, where $M = Abstr_r(M')$ and $N = Abstr_r(N')$, consisting of

- the abstraction $h_{DT}: M_{DT} \to N_{DT} = Abstr_{r_{DT}}(h'_{DS}: M'_{DS} \to N'_{DS})$ of the data type homomorphism as in Def. 3.2.5,
- the abstraction $h_{DS}: M_{DS} \to N_{DS} = Abstr_{r_{DS}}(h'_{DS}: M'_{DS} \to N'_{DS})$ of the data state homomorphism as in Def. 3.2.9,

- the abstraction $h_{Tr}: M_{Tr} \to N_{Tr} = Abstr_{r_{Tr}}(h'_{Tr}: M'_{Tr} \to N'_{Tr}, h'_{Pr}: M'_{Pr} \to N'_{Pr})$ of the transformation homomorphism as in Def. 3.2.14 and
- the abstraction $h_{Pr}: M_{Pr} \to N_{Pr} = Abstr_{r_{Pr}}(h'_{Pr}: M'_{Pr} \to N'_{Pr})$ of the process homomorphism as in Def. 3.2.18.

The abstraction is compatible with composition and identities:

$$\begin{array}{ll} & Abstr_r(h' \circ g') = Abstr_r(h') \circ Abstr_r(g') \\ \text{and} & Abstr_r(id_{M'}) = id_{Abstr_r(M')} \end{array}$$

Like the restriction construction the abstraction construction also constitutes functors between the model categories.

Theorem 3.2.22 (Abstraction Functor)
The abstraction construction for an integrated signature refinement $r: \Sigma \to \Sigma'$ forms a functor $Abstr_r: Mod(\Sigma') \to Mod(\Sigma)$.

Summarizing this section, we get another specification frame consisting of the category of integrated signatures and refinements and a model functor assigning abstractions to refinements.

Definition 3.2.23 (Integrated Refinement Specification Frame)
The category **IntRef** of integrated signatures and integrated signature refinements, together with the functor $Mod: \mathbf{IntRef}^{op} \to \mathbf{Cat}$ defined by

$$\begin{array}{l} Mod(\Sigma) \quad \text{(the model category as in Theorem 2.1.32)} \\ Mod(r: \Sigma \to \Sigma') = Abstr_r: Mod(\Sigma') \to Mod(\Sigma) \\ \qquad\qquad \text{(the abstraction functor as in Theorem 3.2.22)} \end{array}$$

forms a specification frame, called *specification frame of integrated refinements* $IntRefSF$.

Chapter 4

Constraints

In this chapter we introduce constraints as means to specify properties of integrated models. In Section 4.1 we look at the notion of a *logic of constraints* for a specification frame and define integrated specifications, by combining integrated signatures with sets of constraints. Then, in Section 4.2, we will introduce some examples of logics of constraints for integrated signatures. Finally, in Section 4.3, we introduce the concept of high-level constraints.

4.1 Logics of Constraints and Specifications

In Subsection 4.1.1 we will conceptually introduce the notion of logics of constraints, which will be used in Subsection 4.1.2 to define specifications and specifcation morphisms and examine the effects on the semantics.

4.1.1 Logics of Constraints

Logics of constraints were introduced in [EM90] and [EGR94] as categorical concepts for specifying properties of models. A constraint functor assigns a class of constraints to each signature and a satisfaction relation is defined between models and constraints of the signature.

For signature morphisms the constraint functor yields a translation function between the corresponding constraint classes, where the satisfaction relation needs to be compatible with these translations on one hand and the restriction of models on the other hand.

Definition 4.1.1 (Logic of Constraints)
A *logic of constraints* $LC = (Constr, \models)$ for the specification frame $IntSigSF = (\mathbf{IntSig}, Mod)$ of integrated signatures is given by a functor

$$Constr\colon \mathbf{IntSig} \to \mathbf{Class}$$

into the category **Class** of classes, where the translation functions $Constr(f)$

for signature morphisms f are denoted by $f^{\#}$, and for each integrated signature $\Sigma \in \mathbf{IntSig}$ a relation

$$\models_{\Sigma} \subseteq Mod(\Sigma) \times Constr(\Sigma) \ ,$$

called *satisfaction relation*, such that for all morphisms $f: \Sigma \to \Sigma'$, all models $M' \in Mod(\Sigma')$ and all constraints $\phi \in Constr(\Sigma)$, the *satisfaction condition*

$$M' \models_{\Sigma'} f^{\#}(\phi) \iff Restr_f(M) \models_{\Sigma} \phi$$

holds.

The satisfaction relation can be extended to classes of constraints, where a model M satisfies a class of constraints Φ, if and only if it satisfies all constraints in Φ:

$$M \models \Phi \iff \forall \phi \in \Phi : M \models \phi$$

Logics of constraints can be combined to sums of logics of constraints, which are again logics of constraints, by taking for each signature the disjoint union of the constraint classes as new constraint class and as new satisfaction relation a combination of the original relations.

Definition and Fact 4.1.2 (Sum of Logics of Constraints)
Given two logics of constraints $LC_1 = (Constr_1, \models_1)$ and $LC_2 = (Constr_2, \models_2)$ for integrated signatures, the sum $LC_1 + LC_2 = (Constr_3, models_3)$ is a logic of constraint, where the constraint functor is given for signatures by

$$Constr_3(\Sigma) = Constr_1(\Sigma) \uplus Constr_2(\Sigma)$$

and for signature morphisms correspondingly by

$$Constr_3(f) = Constr_1(f) \uplus Constr_2(f) \ .$$

The combined satisfaction relation is constructed by

$$M \models_{3,\Sigma} \phi \iff \begin{cases} M \models_{1,\Sigma} \phi & \text{if } \phi \in Constr_1(\Sigma) \\ M \models_{2,\Sigma} \phi & \text{if } \phi \in Constr_2(\Sigma) \ . \end{cases}$$

Proof: We have to show the satisfaction condition:

For all $f: \Sigma \to \Sigma'$, $\phi \in Constr_3(\Sigma)$ and $M' \in Mod(\Sigma')$ we have:

$$\begin{array}{ll} & M' \models_{3,\Sigma'} Constr_3(f)(\phi) \\ \iff & \text{(sum of logic of constraints, Def. 4.1.2)} \\ & M' \models_{1,\Sigma'} Constr_1(f)(\phi) \text{ if } \phi \in Constr_1(\Sigma) \\ & M' \models_{2,\Sigma'} Constr_2(f)(\phi) \text{ if } \phi \in Constr_2(\Sigma) \\ \Longrightarrow & (LC_1 \text{ and } LC_2 \text{ logics of constraints, Def. 4.1.1)} \\ & Restr_f(M') \models_{1,\Sigma} \phi \text{ if } \phi \in Constr_1(\Sigma) \\ & Restr_f(M') \models_{2,\Sigma} \phi \text{ if } \phi \in Constr_2(\Sigma) \\ \iff & \text{(sum of logic of constraints, Def. 4.1.2)} \\ & Restr_f(M') \models_{3,\Sigma} \phi \end{array}$$

□

4.1.2 Specifications and Specification Morphisms

We get integrated specifications by adding to a signature Σ a set Φ of constraints for that signature, which models should satisfy.

Definition 4.1.3 (Integrated Specification)
Given a logic of constraints for integrated signatures $LC = (Constr, \models)$, an integrated specification (Σ, Φ) consists of an integrated signature $\Sigma \in \mathbf{IntSig}$ and a class of constraints $\Phi \subseteq Constr(\Sigma)$.

Now, the model category of an integrated specification is just the category of all models of the signature satisfying the constraints.

Definition 4.1.4 (Model Category for a Specification)
Given an integrated specification (Σ, Φ), the model category $Mod(\Sigma, \Phi)$ is the full subcategory of $Mod(\Sigma)$, where all models satisfy Φ.

$$Mod(\Sigma, \Phi) = \{\, M \in Mod(\Sigma) \mid M \models \Phi \,\}$$

We also want to relate different specifications by specification morphisms, which are signature morphisms being compatible with the constraints, in the sense that the translated constraints of the first specification are implied by the constraints of the second one.

Definition 4.1.5 (Specification Morphisms)
Given two integrated specifications (Σ, Φ) and (Σ', Φ'), a specification morphism $f\colon (\Sigma, \Phi) \to (\Sigma', \Phi')$ is a signature morphism $f\colon \Sigma \to \Sigma'$, such that all models satisfying Φ' also satisfy the translated constraints $f^{\#}(\Phi)$.

$$\forall M' \in Mod(\Sigma')\colon M' \models_{\Sigma'} \Phi' \implies M' \models_{\Sigma'} f^{\#}(\Phi)$$

For specification morphisms the restriction construction respects the constraints in the sense, that the restriction of a model of the second specification is a model of the first one.

Fact 4.1.6 (Restriction for Specification Morphisms)
Given a specification morphism $f\colon (\Sigma, \Phi) \to (\Sigma', \Phi')$, the restriction construction for the signature morphism f also defines a functor $Restr_f\colon Mod(\Sigma', \Phi') \to Mod(\Sigma, \Phi)$ between the model categories of the specifications.

Proof: Since the restriction is already a functor for model categories of signatures, we only have to show, that it respects the constraints, i. e. that $M' \models_{\Sigma'} \Phi'$ implies $Restr_f(M') \models_{\Sigma} \Phi$.
For all $f\colon (\Sigma, \Phi) \to (Sigma', \Phi')$ and $M' \in Mod(\Sigma')$ we have:

$$\begin{array}{ll} & M' \models_{\Sigma'} \Phi' \\ \Longrightarrow & (f \text{ specification morphism, Def. 4.1.5}) \\ & M' \models_{\Sigma'} f^{\#}(\Phi) \\ \Longleftrightarrow & (\text{satisfaction condition, Def. 4.1.1}) \\ & Restr_f(M') \models_{\Sigma} \Phi \end{array}$$

□

The notions of specifications and specification morphisms on the syntactical and models of specifications and restrictions for specification morphisms on the semantical side can again be combined into a specification frame, the specification frame of integrated specifications $IntSpecSF$.

Definition 4.1.7 (Integrated Specification Specification Frame)
The category **IntSpec** of integrated specifications and integrated speification morphisms, together with the functor $Mod\colon \mathbf{IntSpec}^{op} \to \mathbf{Cat}$ defined by

$$
\begin{array}{ll}
Mod(\Sigma, \Phi) & \text{(the model category as in Definition 4.1.4)} \\
Mod(f\colon (\Sigma, \Phi) \to (\Sigma', \Phi)) = Restr_f\colon Mod(\Sigma', \Phi') \to Mod(\Sigma, \Phi) & \\
& \text{(the restriction functor as in Fact 4.1.6)}
\end{array}
$$

forms a specification frame, called *specification frame of integrated specifications* $IntSpecSF$.

4.2 Constraints for Integrated Signatures

In this section we will shortly introduce some logics of constraints for integrated signatures. In Subsection 4.2.1 we will see examples of partial algebra constraints and how to use to constrain the data type and the data state layer. Then, in Subsection 4.2.2, we will see some ways to specify the effects of transformations and processes. Since these notions are only meant to serve as conceptual examples, and are not the main focus of this report, we ommit the exact definitions of translations of constraints and the proofs for the satisfaction condition here.

4.2.1 Data Type and Data State Constraints

Since our formal framework is built on partial algebras as data types and data states, constraints for partial algebras can also be used for the framework. We will later formulate this for arbitrary logics of constraints over the specification frame ($\mathbf{AlgSig}, PAlg$) of algebraic signatures and partial algebras, which is why we first introduce some examples of such logics.

All these examples of "low-level" constraints, in contrast to the "high-level" ones in the next section, have in common that their satisfaction is defined in terms of the satisfaction for variable assignments.

As a first, rather simple example terms can be required to be defined in a partial algebra. These constraints, as all constraints defined over variables and terms, can be translated along morphisms and refinements by translating the sorts of the variables according to the sort mapping and the terms according to the operation mappings.

Definition 4.2.1 (Definedness Constraints)
The logic of *definedness constraints* for partial algebras $LC_{Def} = (Constr_{Def}, \models_{Def})$ is given by the constraints

$$Constr_{Def}(\Sigma) = \{\, (X, t \text{ defined}) \mid X = (X_s)_{s \in \Sigma_S}, t \in T_\Sigma(X) \,\}$$

and the satisfaction relation

$$\begin{array}{l} A \models_{Def} (X, t \text{ defined}) \iff \forall asg\colon X \to A\colon A, asg \models_{Def} (X, t \text{ defined}) \\ \text{where} \quad A, asg \models_{Def} (X, t \text{ defined}) \iff asg \in Dom(t, A) \ . \end{array}$$

Our second example shall be equations, where, similar to the case of homomorphisms, we have different choices depending on how the definedness of terms is treated. We choose a version, where the equality of term evaluations is only required for assignments, for which both terms are defined. We can retrieve the well-known notion of existential equations by taking the union of such an equation and definedness constraints for both sides of the equation.

Definition 4.2.2 (Equational Constraints)
The logic of *equational constraints* for partial algebras $LC_{Equ} = (Constr_{Equ}, \models_{Equ})$ is given by the constraints

$$Constr_{Equ}(\Sigma) = \{\, (X, l = r) \mid X = (X_s)_{s \in \Sigma_S}, l, r \in T_\Sigma(X) \,\}$$

and the satisfaction relation

$$\begin{array}{l} A \models_{Equ} (X, l = r) \iff \forall asg\colon X \to A\colon A, asg \models_{Equ} (X, l = r) \\ \text{where} \quad A, asg \models_{Equ} (X, l = r) \iff [\quad asg \in Dom(l, A) \cap Dom(r, A) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \implies \overline{asg}(l) = \overline{asg}(r) \quad] \ . \end{array}$$

As a last example we will define the logic of conditional equations, which consist of a set of definedness and equational constraints as premise and one definedness or equational constraint as conclusion, where these are formulated over the same variable family. A conditional equation is satisfied, if if for all assignments, for which the premise holds, also the conclusion is satisfied.

Definition 4.2.3 (Condtional Equational Constraints)
The logic of *conditional equational constraints* for partial algebras $LC_{CEqu} = (Constr_{CEqu}, \models_{CEqu})$ is given by the constraints

$$\begin{array}{ll} Constr_{CEqu}(\Sigma) = \{ & (X, p_1, \ldots, p_n \to c) \mid X = (X_s)_{s \in \Sigma_S}, \\ & (X, p_1), \ldots, (X, c) \in Constr_{Def+Equ}(\Sigma)\} \end{array}$$

and the satisfaction relation

$$\begin{array}{l} A \models_{CEqu} (X, p_1, \ldots, p_n \to c) \iff \\ \forall asg\colon X \to A\colon A, asg \models_{CEqu} (X, p_1, \ldots, p_n \to c) \\ \text{where} \quad A, asg \models_{CEqu} (X, p_1, \ldots, p_n \to c) \iff \\ \qquad [\quad A, asg \models_{Def+Equ} (X, p_1) \ \wedge \ldots \wedge \ A, asg \models_{Def+Equ} (X, p_n) \\ \qquad\quad \implies A, asg \models_{Def+Equ} (X, c) \quad] \ . \end{array}$$

Now, constraints for the data type and data state layer can be constructed from arbitrary partial algebra constraints, by just requiring, that the partial algebra constraint is satisfied for the data type or all data states respectively. Since the data type is included in all data states, we only need to define this for data states, which subsumes the notion for the data type as special case.

In practical use, however, it makes sense to seperate these cases in the notation of specifications, because of the conceptual difference. Similar to the wide-spread practice of grouping equations with the operation symbols this does not affect the semantics, but makes specifications eaiser to grasp.

Definition 4.2.4 (Data Type and Data State Constraints)
Given a logic of partial algebra constraints $LC_{PA} = (Constr_{PA}, \models_{PA})$, there is an induced logic of *data type and data state constraints* over integrated signature $LC_{DS}(LC_{PA}) = (Constr_{DS}, \models_{DS})$ consisting of the constraints

$$Constr_{DS}(\Sigma) = Constr_{PA}(\Sigma_{DS})$$

and the satisfaction relation

$$M \models_{DS} c \iff \forall A \in M_{DS}\colon A \models_{PA} c \ .$$

4.2.2 Transformation and Process Constraints

For the transformation and process signatures, we will introduce a method here, where we construct a “pre-post-signature” containing all dynamic operation symbols twice, once for the source, once for the target data states.

Definition 4.2.5 (Pre-Post-Signature)
Given a data state signature Σ_{DS} w.r.t. a data type signature Σ_{DT}, the *pre-post-signature* Σ_{PP} consists of the sort symbols of Σ_{DS}, the operation symbols of Σ_{DT} and for each operation symbol of $\Sigma_{DS} \setminus \Sigma_{DT}$ a pre- and a post-operation symbol.

$$\begin{aligned}
&\Sigma_{PP,S} = \Sigma_{DS,S} \\
&\Sigma_{PP,Op} = \Sigma_{DT,Op} \cup \{\, op@pre, op@post \mid op \in \Sigma_{DS,Op} \setminus \Sigma_{DT,Op} \,\} \\
&dom(op@pre) = dom(op@post) = dom(op) \\
&cod(op@pre) = cod(op@post) = cod(op)
\end{aligned}$$

These pre-post-signatures are interpreted by contructing a partial algebra combining the source and the target data state. Elements of the dynamic sorts, which are contained in both data states are included only once in the combined algebra, while the dynamic operations of both data states are disjointly included. This allows to also formulate invariants or operation interpretations in the target data state determined by interpretations in the source data state.

Definition 4.2.6 (Pre-Post-Algebra)
Given a transformation or process expression $x_{A,B}(a, b)$, the *pre-post-algebra* $A + B$ is a Σ_{PP}-algebra, where the sorts are the unions of the sorts of A and

B, the pre-operations are the operations of A and the post-operations are the operations of B.

$$\begin{array}{l}
(A+B)_s = A_s \cup B_s \\
op_{A+B} = op_A = op_B = op_M \text{ for } op \in \Sigma_{DT,Op} \\
op@pre_{A+B} = op_A \\
op@post_{A+B} = op_B \text{ for } op \in \Sigma_{DS,Op} \setminus \Sigma_{DT,Op}
\end{array}$$

Note, that for static sorts the union of $A_s = M_s$ and $B_s = M_s$ is again just M_s, and that the operations have the correct functionalities, since $A_{dom(op)} \subseteq A_{dom(op)} \cup B_{dom(op)}$ and similarly for *cod* and B.

Now, pre-post-constraints for a transformation or process symbol are arbitrary partial algebra constraints over the pre-post-signature with the disjoint union of the input and output parameters contained in the variables. The additional variables are, as for partial algebras implicitly universally quantified. A transformation step satisfies such a constraint, if the combined algebra satisfies the constraint for all assignments extending the parameter assignments, and transformations satisfy it, if all transformation steps satisfy it.

Definition 4.2.7 (Pre-Post-Constraint)
Given a logic of partial algebra constraints $LC_{PA} = (Constr_{PA}, \models_{PA})$, where the constraints (X, c) are formulated over variables X, there is an induced logic of ***pre-post-constraints*** over integrated signature $LC_{PP}(LC_{PA}) = (Constr_{PP}, \models_{PP})$ consisting of the constraints

$$\begin{array}{ll}
Constr_{PP}(\Sigma) = \{ & (s, X, c) \mid s \in \Sigma_{Tr} \cup \Sigma_{Pr}, X = (X_s)_{s \in \Sigma_{DS,S}}, \\
& in(s) \subseteq X, out(s) \subseteq X, (X, c) \in Constr_{PA}(\Sigma_{PP}) \ \}
\end{array}$$

and the satisfaction relation

$$\begin{array}{l}
M \models_{PP} (s, X, c) \iff \forall s_{A,B}(a, b) \in s_M \colon s_{A,B}(a, b) \models_{PP} (X, c) \\
\text{where} \quad s_{A,B}(a, b) \models_{PP} (X, c) \iff \\
\qquad \forall asg \colon X \to A + B \colon [\quad a \subseteq asg, b \subseteq asg \\
\qquad\qquad\qquad \implies A + B, asg \models_{PA} (X, c) \quad] \ .
\end{array}$$

Pre-post-constraints are able to determine elements which must be present in the input and output data states respectively, namely the values of dynamic terms for pre- or post-operation symbols, for which definedness constraints have to be satisfied.

In contrast to that we have, until now, no way of specifying elements, that are added or deleted, i. e. that must not be present in the input or output data state, since the pre-post-algebra contains the union of the carrier sets and the information, whether elements stem from both or only one of the states, is lost. In Subsection 4.3.3 we will see how to use attributed graph transformation for the purpose of, among other things, specifying the deletion and insertion of elements.

Also, pre-post-constraints for process symbols do not determine the data states reached inside the process in any way, but can only formulate preconditions, postconditions and invariants, that have to hold in the initial and final data state, but not necessarily in between. They can not specify the structure of the RSTSs or the transformations used by the processes, either. We will use CCS specifications for the purpose of specifying the structure of processes in Subsection 4.3.2.

4.3 High-Level Constraints

In this section we introduce the notion of high-level constraints, which are high-level structures of some modelling technique used to constrain an integrated signature. In Subsection 4.3.1 we will introduce the concept of instantiating our formal framwework with a modelling technique, and see how to use such an instatiation as logic of constraints for integrated signatures. After that, we will give two example instantiations, namely for the CCS process algebra in Subsection 4.3.2 and for attributed graph transformation in Subsection 4.3.3.

4.3.1 Instantiations as Constraints

For modelling techniques, that are to be integrated into our framework, we just assume that they define a class $\mathcal{HL}$ of high-level structures.

An instantiation of the formal framework to such a modelling technique is then given by the assignment of an integrated signature and a subcategory of the models of the signature to a structure $HL \in \mathcal{HL}$.

In the previous versions of the framework in [EO01a, EOB+02b] instantiations were considered to assign only single models to a high-level structure. We extend this to categories of models in order to be able to integrate techniques with loose semantics.

Moreover, we can decouple integrated techniques, such as attributed graph transformation or algebraic high-level nets: In the original version of the framework, the data type algebra is fixed and included in the high-level structures for such techniques and the interpretation just chooses this algebra. In contrast to that, we would like to be able to leave the data type unspecified, and assign a category of models for all possible algebras that could be taken as the data type.

Definition 4.3.1 (Interpretation of High-Level Structures)
Given a class $\mathcal{HL}$ of high-level structures, an *interpretation* (Sig, Mod) of this class in the framework of integrated signatures is given by two functions

$$Sig\colon \mathcal{HL} \to \mathbf{IntSig} \ ,$$

assigning an integrated signature $Sig(HL)$ to each high-level structure HL, and

$$Mod\colon \mathcal{HL} \to \mathbf{Cat} \ ,$$

assigning a category $Mod(HL) \subseteq Mod(Sig(HL))$ to each high-level structure HL.

This very generic notion allows the integration of many kinds of modelling-techniques. Since we only require a (syntactical) class of structures, it is possible to use the framework to assign a formal semantics to semi-formal modelling techniques. Of course the models assigned to a structure of the modelling technique should somehow be related to the "normal" semantics of the technique, formal or semi-formal.

The idea of high-level constraints was introduced in [EOB+02b]. A high-level constraint w.r.t. an integrated signature Σ is a high-level structure HL with a signature morphism from $Mod(HL)$ to Σ, that is satisfied for a model M, if the restriction of M is a model of HL.

Definition and Fact 4.3.2 (Logic of High-Level Constraints)
Given a class $\mathcal{HL}$ of high-level structures and an interpretation (Sig, Mod) for this class, the logic of *high-level constraints* $LC_{\mathcal{HL}} = (Constr_{\mathcal{HL}}, \models)$ for this interpretation is given by the constraints

$$Constr_{\mathcal{HL}}(\Sigma) = \{\, (HL, m) \mid HL \in \mathcal{HL}, m\colon Sig(HL) \to \Sigma \,\}\ ,$$

the translations of constraints

$$f^{\#}(HL, m\colon Sig(HL) \to \Sigma) = (HL, f \circ m\colon Sig(HL) \to \Sigma') \text{ for } f\colon \Sigma \to \Sigma'$$

and the satisfaction relation

$$M \models (HL, m) \iff Restr_m(M) \in Mod(HL)\ .$$

Proof: We have to show the satisfaction condition:

For all $f\colon \Sigma \to \Sigma'$, $(HL, m) \in Constr(\Sigma)$ and $M' \in Mod(\Sigma')$ we have:

$$\begin{array}{ll}
 & M' \models f^{\#}(HL, m) \\
\iff & \text{(translation of high-level constraints, Def. 4.3.2)} \\
 & M' \models (HL, f \circ m) \\
\iff & \text{(satisfaction relation for high-level constraints, Def. 4.3.2)} \\
 & Restr_{f \circ m}(M') \in Mod(HL) \\
\iff & \text{(restriction of integrated models, Def. 2.2.17)} \\
 & Restr_m(Restr_f(M')) \in Mod(HL) \\
\iff & \text{(satisfaction relation for high-level constraints, Def. 4.3.2)} \\
 & Restr_f(M') \models (HL, m)
\end{array}$$

□

We are able to translate high-level constraints along signature refinements, if we allow the morphism m in a constraint (HL, m) to be a refinement instead of a plain morphism and modify the satisfaction relation to use the abstraction functor $Abstr_m$ instead of $Restr_m$.

In order to specify a system by heterogeneous modelling techniques, one has to give an interpretation of each of the modelling techniques in the formal framework. Then the sum of the corresponding logics of high-level constraints can be used to specify the system, whose structure is given by an integrated signature.

The morphism in a high-level constraint determines which part of the whole system is specified by the constraint.

Moreover, these logics of high-level constraints can be combined with some low-level constraints, which are for example useful for describing the interaction of different high-level constraints.

4.3.2 CCS Process Algebra

In this subsection we will integrate the "Calculus of Communicating Systems" (CCS) [Mil89], which is a process algebra developed by R. Milner, into our framework. A sketch of this integration for the original version of the framework was given in [EO01a].

Process algebras are means to reason about the behaviour of concurrent systems. These systems consist of agents, which perform actions. Two agents can synchronize on certain actions. In the CCS the approach is to have complementary actions, where an agent willing to perform an input action synchronizes with an agent willing to perform the complementary output action.

In [Mil89] an infinite set $\mathcal{A}$ of names is assumed, input actions a are taken from this set, and output actions, denoted by $\overline{a}$, are taken from the corresponding set of co-names $\overline{\mathcal{A}}$. When synchronizing an action and its complementary action, the name is forgotten and the synchronized action becomes τ, which is the name for an arbitrary internal action, because the CCS is designed to deal with external commnunications an does not need any information about the names of internal actions.

In our framework, however, whole systems including their internal behaviour are modelled. Therefore we still need the names of synchronized actions and choose an approach, where we have for each name an input action a_{in}, an output action a_{out} and an internal action a_{int}. When we mean either of the three, we will write a_x.

For each agent of the system, there is an agent constant. For these constants there has to be a defining equation $A \stackrel{def}{=} E$ with an agent expression E, where agent expressions are built from action constants using action prefixing $a_x.E$ for an action a_x, summation $\sum_{i \in I} E_i$ over an I-indexed set of expressions, composition $E_1|E_2$, restriction $E \setminus L$ for a set L of names and relabelling $E[f]$ for a function $f\colon \mathcal{A} \to \mathcal{A}$. This also includes $0 = \sum_{i \in \emptyset} E_i$, which specifies an agent, which is not able to perform any action.

In many cases CCS agents are specified recursively and are not meant to ever finish, but a state where they can not perform any action is considered a "deadlock", which is in general not wanted. To use CCS agents to specify processes of our framework, however, they need to have final states. At least an agent, which is associated to a process has to have final states. Here, we will just use deadlocks as final states, making them wanted in case the agent has completed its task. In a refined version, one would probably want a way to distinguish between succesfull completion and unexpected deadlock, such as it is included e.g. in LOTOS.

The semantics of CCS is defined in terms of labelled transition systems $LTS =$

$(S, T, \rightarrow)$, where S is a set of states, T a set of transition labels and $\rightarrow$ a family of relations $\xrightarrow{t} \subseteq S \times S$ for all $t \in T$.

In the case of CCS the states are agent expressions, the transition labels are the actions and the transition relations are defined by the inference rules in Fig. 4.1.

$$\textbf{Act}\ \frac{}{a_x.E \xrightarrow{a_x} E} \qquad \textbf{Sum}_j\ \frac{E_j \xrightarrow{a_x} E'_j}{\sum_{i \in I} E_i \xrightarrow{a_x} E'_j}\ (j \in I)$$

$$\textbf{Com}_1\ \frac{E \xrightarrow{a_x} E'}{E|F \xrightarrow{a_x} E'|F} \qquad \textbf{Com}_2\ \frac{F \xrightarrow{a_x} F'}{E|F \xrightarrow{a_x} E|F'}$$

$$\textbf{Com}_3\ \frac{E \xrightarrow{a_{in}} E' \quad F \xrightarrow{a_{out}} F'}{E|F \xrightarrow{a_{int}} E'|F'} \qquad \textbf{Res}\ \frac{E \xrightarrow{a_x} E'}{E \setminus L \xrightarrow{a_x} E' \setminus L}\ (a \notin L)$$

$$\textbf{Rel}\ \frac{E \xrightarrow{a_x} E'}{E[f] \xrightarrow{f(a)_x} E'[f]} \qquad \textbf{Con}\ \frac{E \xrightarrow{a_x} E'}{A \xrightarrow{a_x} E'}\ (A \stackrel{def}{=} E)$$

Figure 4.1: Inference rules for the semantics of CCS

In the value-passing CCS calculus agent constants and actions can be parameterized by values. In [Mil89] these values are not given any sort, but since we have sorts in our framework, we allow them for CCS as well.

Agent expressions are modified, so that they input actions $a_{in}(x_1, \ldots, x_n)$ are parameterized with variables, to which they assign the values of the parameters of the corresponding output actions, when the transition occurs, and output actions $a_{out}(t_1, \ldots, t_n)$ are equipped with terms, whose evaluations are conversely assigned to the variables of the corresponding input action. Parameterized constants are equipped with variables on the left-hand side of defining equations, expressing that these agent gets the variables when initialized, and with terms, if they are used on the right-hand side, determining the values that are given to the recursively used agent.

The terms used in output actions and recursive calls of agent constants may only contain variables, which were defined by the parameters of the constant or by previous input actions.

Additionally a sort *bool* with some operations returning it is assumed and expressions can contain conditionals **if** b **then** E expressing that the agent can behave like the expression E if the boolean expression b, also containing only previously defined variables, evaluates to $true$.

We assume a CCS specification, that is to be interpreted in the framework to consist of a signature for the values, that are to be passed by actions, a signature for the action names with sorts for each action, that determine the sorts of the values passed by that action, and a set of defining equations, where one or more of the agents are marked to be interpreted as the processes.

The integrated signature for such a specification integrates the value signature on the data type layer, the action names as transformation symbols, where the value sorts are interpreted as input parameters, and the marked agent constants as process symbols with the parameters of the constant as input parameters.

The integrated models of such a system are given by an arbitrary total data type for the value signature, one single data state, since there are no dynamic sorts and operations, and transformations having transformation steps on the one data state for all possible parameter assignments.

On the process layer the interpretation of the CCS semantics is done, where the RSTS is the part of the LTS of CCS reachable from the agent constant, interpreted as graph. Processes are then given for all possible parameter assignments by occurrences mapping the edges of the LTS to transformation sets containg just the one transformation step corresponding to the action and the given parameters.

We are not able to formulate the synchronization of such a process, which still contains unsynchronized input and output actions, with other processes in the framework now, because we did not include a possibility to let high-level constraints interact with each other. They are totally unaware of the environment, in which the signature of the constraint is embedded by the morphism of the constraint.

4.3.3 Attributed Graph Transformation

In this subsection we will adopt the interpretation of attributed graph transformation [ELO95] introduced in [EO01a] and refined in [Ehr01] to our modified version of the framework.

First we will shortly introduce the basic notions of attributed graphs and graph transformation rules and rule applications, which will be interpreted as data states, transformations and transformation steps, respectively. For this, we first need the notion of graph signatures, which define the structure of attributed graphs.

Definition 4.3.3 (Attributed Graph Signature)
An *attributed graph signature* is a triple

$$(ASig, GSig, Attr)$$

consisting of

- an *attribute signature* $ASig = (S_A, Op_A, dom, cod)$, which is just an ordinary algebraic signature,
- a *graphical signature* $GSig = (S_G, Op_G, dom, cod)$, where S_G is a set of symbols for sorts of *graphical elements* and Op_G a set of symbols for *connections* with $dom\colon Op_G \to S_G$ and $cod\colon Op_G \to S_G$, i.e. all connections are unary, and

- an *attributation signature* $Attr = (Op_{Attr}, dom, op)$, which is a set Op_{Attr} of *attributations* with $dom: Op_{Attr} \to S_G$ and $cod: Op_{Attr} \to S_A$.

Now an attributed graph contains a (total) algebra for the attribute signature. For the graphical signature a graph consisting of sets of graphical elements and connections, being total functions between graphical element sets, is included. The attributation signature is interpreted by total assignments of elements of the attribute algebra to the elements of the graph.

Each graphical sort, which is in the domain of a connection symbol can be seen as a kind of edge, where edges not only relate two nodes, but can connect an arbitrary number of nodes and edges.

Definition 4.3.4 (Attributed Graph)
An *attributed graph* for a given attributed graph signature $(ASig, GSig, Attr)$ is a triple

$$G = (G_A, G_G, G_{Attr})$$

consisting of

- an (total) attribute algebra $G_A \in Alg(ASig)$,
- a graph, which is a (total) algebra $G_G \in Alg(GSig)$ and
- a family $G_{Attr} = (attr_G: G_{G,dom(attr)} \to G_{A,cod(attr)}$ of (total) attributations.

To relate attributed graphs we introduce attributed graph homomorphisms, which express the relations between the attribute as well as the graph part of an attributed graph.

Definition 4.3.5 (Attributed Graph Homomorphism)
Given two attributed graphs G and H for the same attributed graph signature, an *attributed graph homomorphism* is a pair

$$h: G \to H = (h_A: G_A \to H_A, h_G: G_G \to H_G)$$

consisting of algebra homomorphisms, which are compatible with the attributations, i. e. $\forall attr \in Attr: h_{A,cod(attr)} \circ attr_G = attr_H \circ h_{G,dom(attr)}$ (see Fig. 4.2).

$$\begin{tikzcd}
G_{G,dom(attr)} \arrow[r, "h_{G,dom(attr)}"] \arrow[d, "attr_G"'] \arrow[dr, phantom, "(=)"] & H_{G,dom(attr)} \arrow[d, "attr_H"] \\
G_{A,cod(attr)} \arrow[r, "h_{A,cod(attr)}"'] & H_{A,cod(attr)}
\end{tikzcd}$$

Figure 4.2: Attributed graph homomorphisms

Graph transformation rules are means to specify the manipulation of the graph structure and the attributation of attributed graphs. They use the double-pushout approach to transformation of structures, where a rule is given by a span $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$.

In order to be generic w. r. t. the attribute algebra, we assume a rule to have an associated family of variables X over the attribute signature and the attributed graphs of the rule to contain the term algebra with variables $T_{ASig}(X)$ as attribute algebra.

Definition 4.3.6 (Graph Transformation Rule)
Given an attributed graph signature $(ASig, GSig, Attr)$, a *graph transformation rule* is a tuple

$$r = (X, L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$$

consisting of

- a family $X = (X_s)_{s \in S_A}$ of variables over the attribute signature,
- three attributed graphs L, K and R with $L_A = K_A = R_A = T_{ASig}(X)$ and
- two injective attributed graph homomorphisms l and r with $l_A = r_A = id_{T_{ASig}(X)}$.

A rule can be applied to a graph G, if for a given match $m_L : L \to G$ the pushout complement C w. r. t. l exists. Then the result H of the application is constructed as the pushout of C and R over K. This application has the effect, that the elements of $L \setminus l(K)$ are deleted and the elements of $R \setminus r(K)$ are added to the graph.

For the attribute part, a rule can be applied to a graph with an arbitrary attribute algebra G_A, where the matches for the attribute part are given by an evaluation $\overline{asg}$ w. r. t. an assignment $asg : X \to G_A$ and the application of the rule leaves the algebra unchanged (up to isomorphism).

Definition 4.3.7 (Rule Application)
Given a graph transformation rule r and an attributed graph homomorphism $m_L : L \to G$ from the left-hand side of the rule into an attributed graph G, the rule can be *applied* to G w. r. t. m_L, if the pushout complement C, called *context graph*, in the left part of Fig. 4.3 exists. The result of the application is the pushout graph H in the right part of Fig. 4.3. This derivation step is written as $G \overset{r,m_L}{\Longrightarrow} H$.

In the literature a transformation system is normally defined as a set of rules with a start graph. Since on the one hand we want to interpret graph transformation exclusively on the transformation layer and have no way of interpreting the start graph here, and on the other hand a start graph would fix the attribute algebra, because our rules cannot change it, we do not include a start graph here.

$$\begin{tikzcd} L \arrow[d, "m_L"'] \arrow[dr, phantom, "(PO)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "m_K"] \arrow[dr, phantom, "(PO)"] & R \arrow[d, "m_R"] \\ G & C \arrow[l, "l'"] \arrow[r, "r'"'] & H \end{tikzcd}$$

Figure 4.3: Application of a rule

Thus, a graph transformation system is just an attributed graph signature with a set of rules over that signature.

Definition 4.3.8 (Graph Transformation System)
A graph transformation system GTS is given by an attributed graph signature $(ASig, GSig, Attr)$ and a set $Rules$ of graph transformation rules w.r.t. this signature. We denote by $\mathcal{GTS}$ the class of all graph transformation systems.

Now attributed graph transformation systems can be interpreted in our framework. The integrated signature for a graph transformation system contains the attribute signature on the data type layer and the union of the attribute, graphical and attributation signature on the data state layer. On the transformation layer there is a transformation symbol or each rule.

The variables for these transformation symbols shall be used to represent the match morphisms. Hence we choose for the input parameters a variable of the corresponding sort for each graphical element of L and all variables of X used in attributations of the left hand side. For the output parameters we include a variable for each graphical element of R, which is not already present in L, and the variables of X used in the attributations of R and not contained in the input parameters.

This construction of the parameters is chosen, because this way the input parameters concern the elements, which are determined by the graph to be transformed, while the output parameters are used for elements whose matches can be freely chosen.

Definition 4.3.9 (Integrated Signature of a GTS)
Given a graph transformation system $GTS \in \mathcal{GTS}$ consisting of the attributed graph signature $(ASig, GSig, Attr)$ and the set $Rules$ of graph transformation rules, the integrated signature $Sig(GTS)$ consists of

- the data type signature $Sig(GTS)_{DT} = ASig$,
- the data state signature $Sig(GTS)_{DS} = ASig \uplus GSig \uplus Attr$ and
- the transformation signature $(Sig(GTS)_{Tr}, in, out)$, where $Sig(GTS)_{Tr}$ contains a transformation symbol r for each rule $r = (X, L \xleftarrow{l} K \xrightarrow{r} R) \in Rules$ with

$$\begin{array}{ll} in(r)_{gs} = L_{G,gs} & \text{for each } gs \in S_G, \\ in(r)_{as} = Var(L_{Attr})_{as} & \text{for each } as \in S_A, \\ out(r)_{gs} = R_{G,gs} \setminus r_{G,gs}(K_{G,gs}) & \text{for each } gs \in S_G \text{ and} \\ out(r)_{as} = Var(R_{Attr})_{as} \setminus Var(L_{Attr})_{as} & \text{for each } as \in S_A. \end{array}$$

Here $Var(G_{Attr})$ denotes the variables occuring in the attributations:

$$Var(G_{Attr}) = \bigcup_{attr \in Op_{Attr}} \bigcup_{x \in G_{G,dom(attr)}} Var(attr_G(x))$$

The process signature of $Sig(GTS)$ is empty.

The integrated models for a graph transformation system are all models, where the data type is a total algebra, the data states are also total and the transformation steps correspond to rule applications. We require the data state model to contain all possible graphs and attributations to guarantee that a transformation step exists for each corresponding rule application.

Since pushouts and pushout complements are only unique up to isomorphism, rule applications can normally perform arbitrary renamings on the graphical elements. In order to be compatible to instantiations, which rely on the identity of elements of the data states, who are meant to be identical, we restrict this to allowing only transformations where no renamings occur, i. e. the homomorphisms l' and r' in Fig. 4.3 are inclusions instead of injections, which are allowed in general.

Definition 4.3.10 (Integrated Models of a GTS)
Given a graph transformation system GTS and its corresponding integrated signature $Sig(GTS)$, the model category $Mod(GTS) \subseteq Mod(Sig(GTS))$ of GTS consists of all models M for which

- the data type M_{DT} is a total algebra,
- the data state model M_{DS} consists of all total algebras extending the data type, i. e.

$$M_{DS} = \{ A \in Alg(Sig(GTS)_{DS}) \mid A|_{Sig(GTS)_{DT}} = M_{DT} \} \ ,$$

- the transformations r_M contain all transformation expressions $r_{A,B}(a, b)$, for which $A \overset{r,m_L}{\Longrightarrow} B$ with m_L given by the variable assignments in a and b and the node mapping of a, where l' and r' in the double-pushout are inclusions, and the node mapping of b is the same as in m_R.

Chapter 5

Example

In this chapter we will see a more exhaustive example of a system specified with an architecture consisting of unions and refinements of integrated specifications. The modelled system will be a cryptographic infrastructure with a centralized trustcenter, from which public keys of participants can be obtained.

We will outline the step-wise development of the system by refinement, where we start with an abstract specification in Section 5.1, which just states the basic requirements of a system with hosts exchanging messages.

This specification is refined by introducing encryption of the messages with an asymmetric cryptographic algorithm in Section 5.2, where an infrastructure with a trustcenter, from which the keys of other hosts are obtained, is specified by a CCS high-level constraint. The algorithm, however, is left generic and only specified by abstract requirements.

In Section 5.3 the structure of the communication net is introduced by making the hosts a dynamic sort with links between them and specifying the behaviour of the communication medium with attributed graph transformation.

The cryptographic algorithm is refined in Section 5.4 by the use of the RSA algorithm, which is specified by an algebraic specification.

Possible changes to the net structure are specified with the use of attributed graph transformation in Section 5.5. The union of this specification with the specification from Section 5.4 yields the specification of the system as far as we want to go into detail.

5.1 Basic Requirements

The signature of the basic requirements specification *Requ* in Figure 5.1 provides an abstract view of the communication system. There are two static sorts *host*, representing the communicating hosts, and *content*, representing the possible content of messages to exchange. The operation *received* is meant to assign to each host the last received message. Finally, the transformation *message* is supposed to mean the transmission of a message with the content in the *cont*

data type:	
sorts:	*host*, *content*
data state:	
opns:	$received\colon host \to content$
trafos:	$message(from, to\colon host, cont\colon content; \lambda)$

Figure 5.1: Signature of *Requ*

parameter from the host denoted by the *from* parameter to the host denoted by the *to* parameter.

pre-post:	For *message*:
	$received@post(to)$ defined
	$received@post(to) = cont$

Figure 5.2: Constraint for *Requ*

The only constraint is the pre-post constraint for *message* in Figure 5.2, stating that after a *message* transformation step the value of *received* for the receiving host should be defined and the same as the *cont* parameter of the transformation step.

5.2 Abstract Asymmetric Encryption

data type:	
sorts:	*host*, *plain*, *cipher*, *pubkey*, *seckey*, *keypair*
opns:	$pub\colon keypair \to pubkey$
	$sec\colon keypair \to seckey$
	$enc\colon pubkey\ plain \to cipher$
	$dec\colon seckey\ cipher \to plain$
data state:	
opns:	$kp\colon host \to keypair$
	$received\colon host \to plain$
trafos:	$send(from, to\colon host, pln\colon plain; \lambda)$
	$recv(from, to\colon host, pln\colon plain; \lambda)$
	$rqu(from, to\colon host; \lambda)$
	$rpl(from, to\colon host, pk\colon pubkey; \lambda)$
	$crpsend(from, to\colon host, cph\colon cipher; \lambda)$
	$crprecv(from, to\colon host, cph\colon cipher; \lambda)$
procs:	$message(from, to\colon host, cont\colon plain; \lambda)$

Figure 5.3: Signature of *AbstrEncr*

Now we want to introduce the encryption of messages during transmission. In the signature of the refined specification *AbstrEncr*, shown in Figure 5.3, the sort *host* is preserved, while the sort *content* is renamed to *plain* to distinguish plaintexts from the newly introduced ciphertexts in sort *cipher*. The codomain of the dynamic operation *received* is altered accordingly. The transformation *message* is refined by a process with the same name.

Thus, the signature refinement $enc\colon Requ \rightsquigarrow AbstrEncr$ is given by $enc(host) = host$, $enc(content) = plain$, $enc(received) = received$ and $enc(message) = message$.

The static sorts *keypair* with assigned public keys in sort *pubkey* and secret keys in sort *seckey* and the static operations *enc* and *dec* for encryption and decryption are newly introduced. The new dynamic operation *kp* shall assign keypairs to hosts. It is dynamic in order to allow changes to the net in subsequent refinement steps. The various transformations will serve as atomic actions for the CCS constraint below.

$$
\begin{array}{ll}
\text{condequ:} & \text{For all } kp\colon keypair,\ m\colon plain\text{:} \\
& dec(sec(kp), enc(pub(kp), m)) \text{ defined} \\
& dec(sec(kp), enc(pub(kp), m)) = m \\
& \text{For all } h\colon host\text{:} \\
& kp(h) \text{ defined} \\
\text{pre-post:} & \text{For } recv\text{:} \\
& received@post(to) \text{ defined} \\
& received@post(to) = pln \\
\text{CCS:} & \begin{array}[t]{ll}
enc() = & send_{in}(from, to, pln). \\
& rqu_{out}(from, to). \\
& rpl_{in}(from, to, pk). \\
& crpsend_{out}(from, to, enc(pk, pln)). \\
& enc() \\
trust() = & rqu_{in}(from, to). \\
& rpl_{out}(from, to, pub(kp(to))). \\
& trust() \\
medium() = & crpsend_{in}(from, to, cph). \\
& crprecv_{out}(from, to, cph). \\
& medium() \\
dec() = & crprecv_{in}(from, to, cph). \\
& recv_{out}(from, to, dec(sec(to), cph)). \\
& dec() \\
crpmedium() = & (\ enc() \mid trust() \mid \\
& \quad medium() \mid dec()\ )\backslash \\
& \{rqu, rpl, crpsend, crprecv\} \\
\mathbf{message}(from, to, cont) = & (\ send_{out}(from, to, cont). \\
& \quad recv_{in}(from, to, cont).0 \mid \\
& \quad crpmedium()\ )\backslash \\
& \{send, recv\}
\end{array}
\end{array}
$$

Figure 5.4: Constraints for *AbstrEncr*

The first constraint in Figure 5.4 is an equation stating the requirement that decryption is the inverse of encryption for the keypairs. Note that the definedness constraint for $dec(sec(kp), enc(pub(kp), m))$ implies definedness of all of the parts of the term, especially $pub(kp)$, $sec(kp)$ and $enc(pub(kp), m)$. Moreover, each host is required to have an associated keypair.

The second constraint is a pre-post constraint on the transformation *recv*, stating that the plain text in *pln* shall be stored in $received(to)$ after a *recv* transformation step. This is corresponding to the pre-post constraint on *message* in the basic specification. Together with the following CCS constraint the translation of the *message* constraint is implied.

Finally, the CCS constraint describes how the transmission of messages is realized by encryption and decryption agents *enc* and *dec*, a medium *medium* and a trustcenter *trust*.

The encryption agent begins with a *send* input action. It then requests the public key of the recipient using a *rqu* output action, gets the answer in a *rpl* input action and, finally, sends the encrypted message to the medium via a *crpsend* output action.

The decryption agent is triggered by a *crprecv* input action and just decrypts the message and forwards it with a *recv* output action. It is assumed that the $dec()$ agent acts on behalf of the receiving host and therefore has direct access to the secret key.

The medium just gets an encrypted message with a *crpsend* input action and forwards it to the receiver with a *crprecv* output action. The trustcenter answers to *rqu* input actions with the public key of the receiver in a *rpl* output message, so that the encryption agent does not have to know the public keys of all hosts.

These four agents are parallelized to form the encrypted medium *crpmedium*, which is then used to specify the *message* process by parallelization with a control agent doing the *send* action of the sender the *recv* action of the receiver and then terminating.

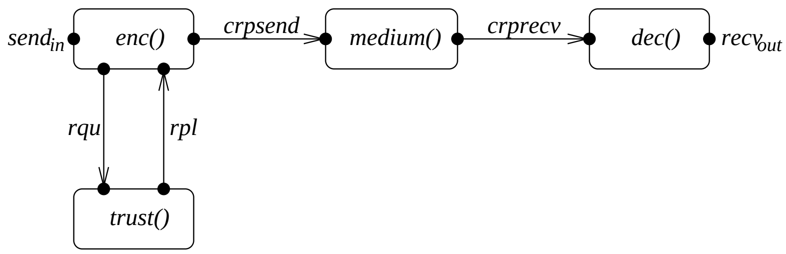

Figure 5.5: Agents in the CCS constraint for *AbstrEncr*

The interconnections of the agents in *crpmedium* are depicted in Figure 5.5.

The signature refinement $enc\colon Requ \rightsquigarrow AbstrEncr$ is a proper specification refinement, because the three constraints above imply the translation of the pre-post constraint for *message*, which is identical to the original constraint (except the translation of the sort of the *cont* parameter), since pre-post constraints can also be attached to processes.

5.3 Net Structure and Communication

In this section we want to refine the static hosts by a dynamic net structure and specify the communication between hosts, i. e. what happens between the *crpsend* and the *crprecv* transformations in the *message* process or, more precisely, in the *medium* agent.

We capture this formally by a refinement of the subsignature *AbstrNet* of *AbstrEncr*, where *AbstrNet* consists of the sorts *host* and *cipher* and the transformations *crpsend* and *crprecv*.

data type:	
sorts:	*cipher*
data state:	
sorts:	*host*, *envelope*, *link*,
	sender, *receiver*, *location*
opns:	$cntnt\colon envelope \to cipher$
	$lnksrc\colon link \to host$
	$lnktrg\colon link \to host$
	$sndsrc\colon sender \to host$
	$sndtrg\colon sender \to envelope$
	$rcvsrc\colon receiver \to envelope$
	$rcvtrg\colon receiver \to host$
	$locsrc\colon location \to envelope$
	$loctrg\colon location \to host$
trafos:	$crpsend(from, to\colon host, cph\colon cipher; env\colon envelope,$
	$\quad snd\colon sender, rcv\colon receiver, loc\colon location)$
	$crprecv(from, to\colon host, cph\colon cipher, env\colon envelope,$
	$\quad snd\colon sender, rcv\colon receiver, loc\colon location; \lambda)$
	$pass(curr, next\colon host, cph\colon cipher, env\colon envelope,$
	$\quad lnk\colon link, loc\colon location; loc'\colon location)$

Figure 5.6: Signature of *DynNet*

In the refined signature *DynNet* in Figure 5.6, which is supposed to be the signature of a set of attributed graph transformation rules, the sort *host* becomes a dynamic sort of nodes, which are connected by edges of sort *link*. In this section we assume, that the hosts form a ring, which is ensured by the net structure changes in Section 5.5 and equational constraints below.

Moreover, a second node sort *envelope* is introduced, which represents messages being in transit between two hosts, where the ciphertext being transmitted is kept in the *cntnt*-attribute. An envelope is connected to the sender and the receiver of the message by edges of sort *sender* and *receiver* respectively, and to the current location of the message by edges of sort *location*.

For the transformation symbols *crpsend* and *crprecv* additional parameters for the new graphical elements are added to the signatures. The passing of a message is represented by the transformation *pass*.

The signature refinement $dyn\colon AbstrNet \rightsquigarrow DynNet$ is given by $dyn(host) =$

host, $dyn(cipher) = cipher$, $dyn(crpsend) = crpsend$ and $dyn(crprecv) = crprecv$. It is not a signature morphism because of *host* becoming dynamic in $DynNet$.

condequ: For all $l1, l2\colon link$:
$lnksrc(l1) = lnksrc(l2) \to l1 = l2$
$lnktrg(l1) = lnktrg(l2) \to l1 = l2$

Figure 5.7: Equational Constraints for $DynNet$

The two equational constraints in Figure 5.7 ensure that for each host there is at most one incoming and at most one outgoing link. Note, that the following attributed graph transformation constraint implies that all of the dynamic operations are total functions. We are not able to formulate by equational constraints the requirements, that there exists an incoming and an outgoing link for each host and that they form just one ring, unconnected hosts or disjoint rings of hosts are still possible.

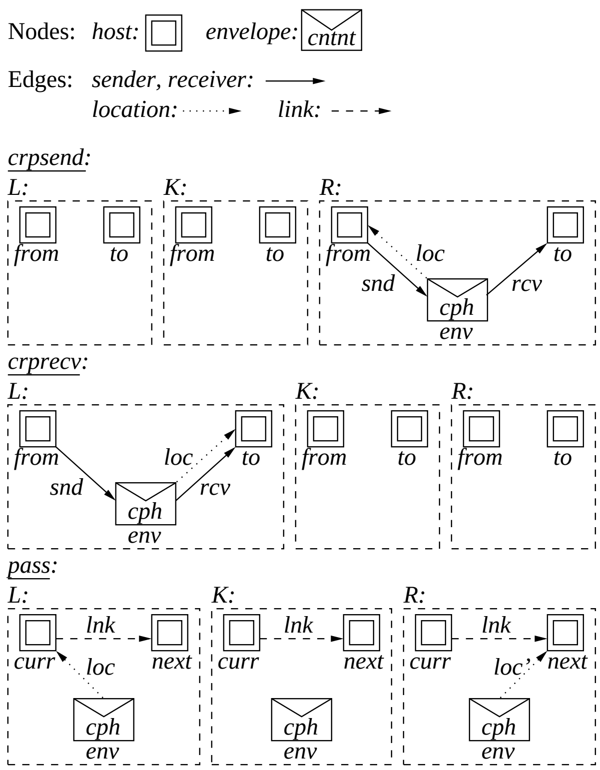

Figure 5.8: AGT constraint for $DynNet$

The attributed graph transformation rules in Figure 5.8 specify the transformations *crpsend*, *crprecv* and *pass*. The transmission of messages is realized by envelopes, which are created at the location of the sending host by applications of the *crpsend*-rule and can be consumed by applications of the *crprecv*-rule, if the envelope is located at the receiving host. The transmission itself is achieved by subsequent applications of the *pass*-rule, which moves the envelope along one direct link between hosts.

5.4 RSA Algorithm

Now, we will refine the encryption of messages to use the well-known RSA algorithm. This refinement is mostly independent of the refinement of the net structure in the last section. We choose to embed the refinement of the net structure into this refinement. Hence, we get the specification architecture in Figure 5.9.

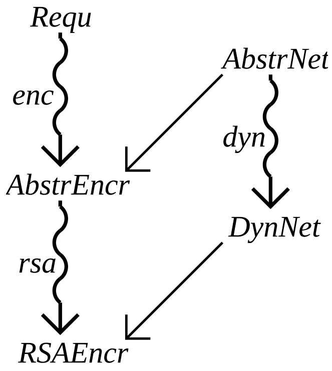

Figure 5.9: Architecture of the specifications

The RSA algorithm is based on exponentiation modulo a product n of large prime numbers p and q. Its security depends on the fact, that factorization is not efficiently possible for large prime factors, which are needed to calculate the inverse for the exponentiation. The public exponent e has to be relatively prime to $\phi(n)$, where ϕ is Euler's phi function given for a product of two primes by $\phi(pq) = (p-1)(q-1)$. The private exponent d has to satisfy $gcd(ed, \phi(n)) = 1$ to be the inverse for exponentiation modulo n. Now encryption of a natural number m into a ciphertext c is done by $c = m^e \bmod n$, and decryption by $m = c^d \bmod n$.

Since these are operations on natural numbers, we first need the specification of the needed operations in Figure 5.10. The constant $zero$, the successor operation $succ$, addition add, multiplication mul and exponentation pow are defined as usual for algebraic specifications. Moreover, we need a predicate lt for "less than", a predecessor $pred$, the remainder mod, the greatest common divisor gcd, which is specified by the equations underlying Euclid's algorithm, and a predicate $prime$.

The signature of $RSAEncr$ in Figure 5.11 is a combination of $DynNet$ and $AbstrEncr$ on the data state, transformation and process layers. On the data type layer additional selector operations for keypairs, public and secret keys are added, which represent the parameters of the algorithm.

$NAT =$	
sorts:	nat
opns:	$zero\colon \to nat$
condequ:	$zero$ defined
opns:	$succ\colon nat \to nat$
condequ:	For all $n\colon nat$:
	$succ(n)$ defined
opns:	$add\colon nat\, nat \to nat$
condequ:	For all $n1, n2\colon nat$:
	$add(n1, n2)$ defined
	$add(n1, zero) = n1$
	$add(n1, succ(n2)) = succ(add(n1, n2))$
opns:	$mul\colon nat\, nat \to nat$
condequ:	For all $n1, n2\colon nat$:
	$mul(n1, n2)$ defined
	$mul(n1, zero) = zero$
	$mul(n1, succ(n2)) = add(mul(n1, n2), n1)$
opns:	$pow\colon nat\, nat \to nat$
condequ:	For all $n1, n2\colon nat$:
	$pow(n1, n2)$ defined
	$pow(n1, zero) = succ(zero)$
	$pow(n1, succ(n2)) = mul(pow(n1, n2), n1)$
opns:	$lt\colon nat\, nat \to \lambda$
condequ:	For all $n1, n2, n3\colon nat$:
	$lt(n1, succ(n1))$
	$lt(n1, n2), lt(n2, n3) \to lt(n1, n3)$
opns:	$pred\colon nat \to nat$
condequ:	For all $n\colon nat$:
	$lt(zero, n) \to pred(n)$ defined
	$pred(succ(n)) = n$
opns:	$mod\colon nat\, nat \to nat$
condequ:	For all $n1, n2\colon nat$:
	$lt(zero, n2) \to mod(n1, n2)$ defined
	$lt(n1, n2) \to mod(n1, n2) = n1$
	$mod(add(n1, n2), n2) = mod(n1, n2)$
opns:	$gcd\colon nat\, nat \to nat$
condequ:	For all $n1, n2\colon nat$:
	$gcd(n1, n2)$ defined
	$gcd(n1, zero) = n1$
	$gcd(n1, n2) = gcd(n2, mod(n1, n2))$
opns:	$prime\colon nat$
condequ:	For all $n1, n2\colon nat$:
	$prime(n1) \to gcd(n1, n2) = succ(zero)$

Figure 5.10: Specification for natural numbers NAT

data type:	$NAT+$
sorts:	$plain$, $pubkey$, $seckey$, $keypair$
opns:	pub: $keypair \to pubkey$
	sec: $keypair \to seckey$
	$minn$: $\to nat$
	p: $keypair \to nat$
	q: $keypair \to nat$
	n: $keypair \to nat$
	e: $keypair \to nat$
	d: $keypair \to nat$
	$pubn$: $pubkey \to nat$
	$pube$: $pubkey \to nat$
	$secn$: $seckey \to nat$
	$secd$: $seckey \to nat$
	$encode$: $plain \to nat$
	$decode$: $nat \to plain$
data state:	
sorts:	$host$, $envelope$, $link$, $sender$, $receiver$, $location$
opns:	kp: $host \to keypair$
	$received$: $host \to plain$
	$cntnt$: $envelope \to cipher$
	$lnksrc$: $link \to host$
	$lnktrg$: $link \to host$
	$sndsrc$: $sender \to host$
	$sndtrg$: $sender \to envelope$
	$rcvsrc$: $receiver \to envelope$
	$rcvtrg$: $receiver \to host$
	$locsrc$: $location \to envelope$
	$loctrg$: $location \to host$
trafos:	$send(from, to: host, pln: plain; \lambda)$
	$recv(from, to: host, pln: plain; \lambda)$
	$rqu(from, to: host; \lambda)$
	$rpl(from, to: host, pk: pubkey; \lambda)$
	$crpsend(from, to: host, cph: cipher; env: envelope, snd: sender, rcv: receiver, loc: location)$
	$crprecv(from, to: host, cph: cipher, env: envelope, snd: sender, rcv: receiver, loc: location; \lambda)$
	$pass(curr, next: host, cph: cipher, env: envelope, lnk: link, loc: location; loc': location)$
procs:	$message(from, to: host, cont: plain; \lambda)$

Figure 5.11: Signature of $RSAEncr$

Since the RSA algorithm operates modulo n, only numbers less than n can be encrypted. We give a constant minimum for this number, represented by $minn$, and an encoding and a decoding operation, which shall translate plaintexts to numbers less than $minn$.

The *enc* and *dec* operations of *AbstrEncr* are not included, because the refinement $rsa\colon AbstrEncr \rightsquigarrow RSAEncr$ assigns encryption and decryption to the operations of the RSA algorithm. This refinement is given by

$$\begin{aligned}&enc(pk,m) \mapsto mod(pow(encode(m),pube(pk)),pubn(pk)) \text{ and}\\ &dec(sk,c) \mapsto decode(mod(pow(c,secd(sk)),secn(sk))) \ ,\end{aligned}$$

where the sort *cipher* is mapped to *nat*.

condequ:	$minn$ defined
	For all $m\colon plain$:
	$decode(encode(m))$ defined
	$decode(encode(m)) = m$
	$lt(encode(m),minn)$
	For all $kp\colon keypair, pk\colon pubkey, sk\colon seckey$:
	$p(kp),q(kp),n(kp),e(kp),d(kp),pub(kp),sec(kp)$ defined
	$pubn(pk),pube(pk),secn(sk),secd(sk)$ defined
	$pubn(pub(kp)) = n(kp)$
	$pube(pub(kp)) = e(kp)$
	$secn(sec(kp)) = n(kp)$
	$secd(sec(kp)) = d(kp)$
	$prime(p(kp)),prime(q(kp))$
	$n(kp) = mul(p(kp),q(kp))$
	$lt(minn,succ(n(kp)))$
	$gcd(e(kp),mul(pred(p(kp)),pred(q(kp)))) = succ(zero)$
	$gcd(mul(e(kp),d(kp)),mul(pred(p(kp)),pred(q(kp)))) =$ $succ(zero)$
	For all $h\colon host$:
	$kp(h)$ defined
pre-post:	For $recv$:
	$received@post(to)$ defined
	$received@post(to) = pln$

Figure 5.12: Constraints for *RSAEncr*

Now, the first four constraints for *RSAEncr* in Figure 5.12 ensure that *minn* is defined and that encoding and decoding with *encode* and *decode* works properly and results in natural numbers less than *minn*. Next, all components of keypairs, public and secret keys are required to be defined and the components of public and secret keys are specified to be the corresponding components of the keypair. The next six constraints ensure the properties given above for keypair components. At last, the equational constraint requiring all hosts to have associated keypairs and the pre-post constraint specifying the effect of the *recv* transformation on the dynamic *received* operation, which were both already present in *AbstrEncr*, are repeated.

Moreover, the CCS constraint from Section 5.2 and the AGT constraint from Section 5.3 have to be translated via the refinement rsa and the inclusion into $RSAEncr$, where we can leave the CCS specification and the AGT rules as they are and just have to compose the refinements and morphisms respectively.

5.5 Net Structure Changes

In this section we add the possibility to change the dynamic net structure from Section 5.3. This is formalized in an extension $ChangeNet$ of the specification $DynNet$. The whole system specified so far can be achieved by the union of $RSAEncr$ and $ChangeNet$ over $DynNet$ as depicted in Figure 5.13. We will not give the specification of the union explicitly here, because it is just a combination of the specifications $RSAEncr$ and $ChangeNet$.

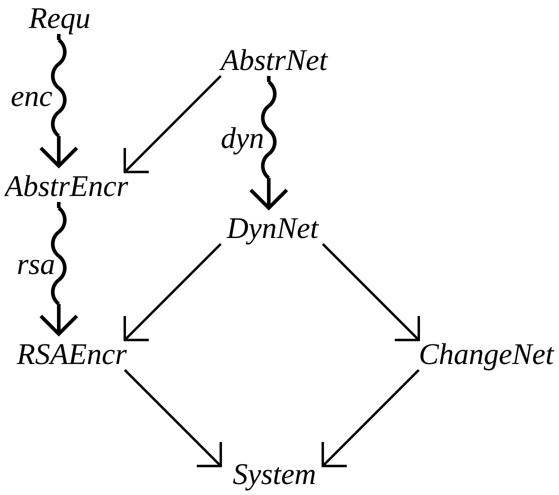

Figure 5.13: Architecture of the specifications

The enhanced signature of $ChangeNet$ in Figure 5.14 has two additional transformation symbols *addhost*, which is supposed to insert new hosts into the ring, and *delhost*, which shall remove hosts from the ring. These are again meant to represent AGT rules and therefore include parameters for all hosts and links affected by the change.

The constraints of $ChangeNet$ are the same as in $DynNet$ with the additional rules in Figure 5.15. These rules ensure that the ring structure is preserved, when a host is added or deleted.

Note, that the last host cannot be removed, because the *delhost* rule is not applicable, if no host remains after the deletion. Likewise, there has to be at least one host with a link to itself in the ring initially, because the *addhost* rule needs a link to be replaced by the new host.

data type:	
sorts:	$cipher$
data state:	
sorts:	$host$, $envelope$, $link$,
	$sender$, $receiver$, $location$
opns:	$cntnt\colon envelope \to cipher$
	$lnksrc\colon link \to host$
	$lnktrg\colon link \to host$
	$sndsrc\colon sender \to host$
	$sndtrg\colon sender \to envelope$
	$rcvsrc\colon receiver \to envelope$
	$rcvtrg\colon receiver \to host$
	$locsrc\colon location \to envelope$
	$loctrg\colon location \to host$
trafos:	$crpsend(from, to\colon host, cph\colon cipher; env\colon envelope,$
	$\quad snd\colon sender, rcv\colon receiver, loc\colon location)$
	$crprecv(from, to\colon host, cph\colon cipher, env\colon envelope,$
	$\quad snd\colon sender, rcv\colon receiver, loc\colon location; \lambda)$
	$pass(curr, next\colon host, cph\colon cipher, env\colon envelope,$
	$\quad lnk\colon link, loc\colon location; loc'\colon location)$
	$addhost(h1, h2\colon host, lnk\colon link; new\colon host, l1, l2\colon link)$
	$delhost(h1, h2, host\colon host, l1, l2\colon link; lnk\colon link)$

Figure 5.14: Signature of $ChangeNet$

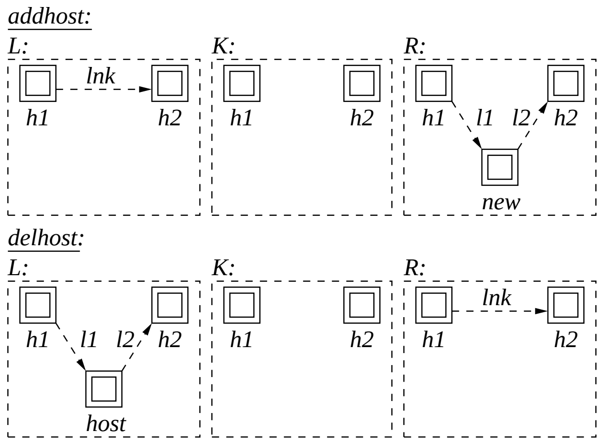

Figure 5.15: AGT constraint for $ChangeNet$

Chapter 6

Conclusion

In this report we have given a formal framework for the integration of data type, state transformation and process aspects of systems, in which heterogeneous modelling techniques can be used to constrain different parts of systems.

We will now give some remarks on related work and future perspectives.

Our framework, as the one in [EO01a] it is based on, is strongly related to the transformation systems of Große-Rhode in [GR01]. Transformation systems contain a transition graph, similar to our RSTSs, that are interpreted in a data space consisting of partial algebras as nodes and action sets as labels, similar to our DSTS. The differences to our framework are, that a system is described by a single transition graph and the data space is not constrained in any way, and, thus, contains all partial algebras and all possible action sets. Moreover, no static data type is distinguished.

For transformation systems an amalgamation of transition graphs is defined, which uses synchronization relations to determine, which states should be identified and which transitions synchronized. Thus, this amalgamation allows the structure of transition graphs to be constructed out of smaller transition graphs. This construction could be adopted to our framework in future work.

The concept of refinements could be used as transformations in the sense of [EOB+02a, EOB+02b] to define a transformation semantics, where transformation semantics is a concept for a purely syntactic semantics of specifications defined by all possible transformations. By defining an extension of refinements along inclusions this would yield a refinement-based component concept as an instantiation of the transformation-based component concept in [EOB+02a], which would be an alternative to the transformation-based component concept for integrated specifications given in [EOB+02a].

Finally, a further point of future work could be the definition of constructive morphisms like in [EO01b], which would yield the possiblity of parameterized integrated specifications, where a model of the body specification is constructed from the parameter specification.

These constructive morphisms would give the possibility to define components in the sense of [EO01b], which were inspired by the algebraic specification modules of [EM90], where the semantics of a component is a construction along a constructive import morphism, followed by a restriction or abstraction along an export morphism or refinement.

Bibliography

[EGR94] H. Ehrig and M. Große-Rhode. Functorial theory of parameterized specifications in a general specification framework. *Theor. Comp. Science*, 135:221–266, 1994.

[Ehr01] H. Ehrig, editor. *From Algebraic Module Specifications to Component Concepts and Integrated Modelling Techniques*. Technical Report 2001/21, Technische Universität Berlin, Fakultät Elektrotechnik und Informatik, 2001.

[ELO95] H. Ehrig, M. Löwe, and F. Orejas. Dynamic abstract data types based on algebraic graph transformation. Number 906 in LNCS, pages 236–254, 1995.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theor. Comp. Science*. Springer, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theor. Comp. Science*. Springer, 1990.

[EO98] H. Ehrig and F. Orejas. Integration paradigm for data type and process specification techniques. *Bull. EATCS 65, Formal Spec. Column, Part 5*, 1998.

[EO01a] H. Ehrig and F. Orejas. A conceptual and formal framework for the integration of data type and process modeling techniques. In *Proc. GT-VMT '01*, number 50,3 in ENTCS. Elsevier, 2001.

[EO01b] H. Ehrig and F. Orejas. A generic component concept for integrated data txpe and process modeling techniques. Technical Report 2001/12, Technische Universität Berlin, Fachbereich Informatik, 2001.

[EOB+02a] H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. A generic component framework for system modeling. In *Proc. FASE '02*, number 2306 in LNCS, pages 33–48. Springer, 2002.

[EOB+02b] H. Ehrig, F. Orejas, B. Braatz, M. Klein, and M. Piirainen. A transformation-based component framework for a generic integrated modeling technique. *Transactions of the SDPS*, 2002.

[GR01] M. Große-Rhode. Semantic integration of heterogeneous formal specifications via transformation systems. Technical Report 2001/13, Technische Universität Berlin, Fachbereich Informatik, 2001.

[Mil89] R. Milner. *Communication and Concurrency.* Int. Series in Comp. Science. Prentice Hall, 1989.