Modal Logics for Reasoning about Object-based Component Composition

Claus Pahl
School of Computer Applications
Dublin City University
Dublin 9, Ireland

Abstract

Component-oriented development of software supports the adaptability and maintainability of large systems, in particular if requirements change over time and parts of a system have to be modified or replaced. The software architecture in such systems can be described by components and their composition. In order to describe larger architectures, the composition concept becomes crucial.

We will present a formal framework for component composition for object-based software development. The deployment of modal logics for defining components and component composition will allow us to reason about and prove properties of components and compositions.

1 Introduction

In order to achieve flexibility in the development of large scale software systems, we design software architectures that allow its software components to be adapted, composed and exchanged. A software development framework using a component-oriented approach usually consists of a library of (reusable) components and a component framework to describe the software architecture through the composition of components. Composing components, or plugging them together, is the key to a flexible composition framework [LS00].

Object-oriented programming languages are ideal for implementation. Drawbacks in these languages are deficiencies in concepts for composition. Reuse is typically considered too late in the development process. The architecture consisting of components and component compositions cannot be described explicitly. This paper addresses a formal foundation for object-based component composition. Our mathematical framework is modal logics. A formal semantics is a necessity to reason about a language construct. Building upon a formal semantics for object-based components, we present semantics for composition that will allow us to reason about the properties of a composition, e.g. proving the correctness of a composition. Modal logics are in particular suitable for reasoning about state-based systems. They also provide an abstract notation for the formulation of components. The idea of a contract between a service provider and a service user can be formalised usin modal logics.

Following Wegner's classification [Weg90], our approach is object-based. Objects are executable entities that provide services. They encapsulate a local state. Components are abstractions over objects. They are a means to construct object-based systems with explicit architectures. The use of components in software development supports reliability by dividing a system into separate elements which can be developed independently. Components also support portability, interoperability, and maintainability by providing abstraction.

We can distinguish between black-box and white-box components. In contrast to white-box components, black-box components encapsulate implementation details and provide therefore an abstract interface. The black-box approach is preferable since the use of components is simplified. The need to understand the implementation in the process of composition is limited.

Encapsulation (supported by black-box components) and composition are the key issues in a component approach. A composition mechanism shall support extendibility and configurability. Other criteria are:

- Variability: how much variation can be achieved. This is supported through parameterisation of components.
- Adaptability: how easy is it to adapt an existing system to changing requirements. A system designed as a composition of components can easily be adapted by modifying single components.
- Ease of use: this is in particular supported by black-box components since the user of the component is not concerned with implementation details.

The important advantage of using black-box components in composition is that a formal contract is explicitly formed between the components involved, e.g. between a service provider and a service requester. The service requester might specify its requirements using a parameter interface. A component typically has an import interface (what services it requires) and an export interface (what services it provides). Forming interfaces for black-box components and their composition limits the flexibility to a certain extent, but this disadvantage is more than compensated by the advantages. The component approach allows us to realise reuse. Components, specified by an abstract interface, can be stored in reuse libraries. The services they offer can be used by other components. Formality allows us to reason about the composition. Types can be checked. We can prove whether a service provider meets the requirements of a requesting component.

Development techniques such as modularisation, composition or implementation can be classified in two dimensions. *Vertical development* denotes the lowering of the level of abstraction, i.e. transforming an abstract specification into a more concrete implementation. In *horizontal development*, components are manipulated without changing the level of abstraction, i.e. components are modified or a number of components are composed to more complex ones. Both dimensions are connected by the *horizontal composition property*, by which the compatibility between constructs of horizontal development (composition of components) and constructs of vertical development (implementation) is expressed. The *vertical composition property* requires the transitivity of the implementation relation.

Our abstract component language provides a high-level framework treating components as black-boxes. This is accompanied by notions that support state-based development and that can e.g. be connected with a particular state- or object-based programming language.

This paper extends previous work. In [Pah97] we have presented an early version of the formal object model. In [Pah00] we have applied the concepts of refinement and implementation to derive a refinement calculus geared towards a specific specification notation.

In Section 2 we will present a formal object model. The foundations for specifying components will be laid in Section 3. In Section 4 we define two relations which will essentially capture the notion of component composition. These relations will help us to reason about a composition. Components will then be defined as parameterised specifications (see Section 5). Finally, we investigate the composition of components in Section 6. We conclude with some summarising remarks and related work. Proofs are omitted in the main part of the paper itself. They are presented in the Appendix.

2 A Simple Notion of Objects

Objects are the basic building blocks of our approach. Objects are state-based entities providing services in form of operations. A local state is encapsulated. We present objects in a typed framework. An early version of this object model has been presented in more detail in [Pah97]. There, we focused on the object notion and showed how this model of objects can also be used to model concepts of imperative programming languages.

The abstract syntax of an object shall be captured by a signature. We assume a set of sorts $Sort$ and a set of identifiers Id for variables and operation names. For each **object signature** Σ, element of the class of object signatures Sig, we define

- a mapping $sorts : Sig \to \mathcal{P}Sort$ with $state \in sorts(\Sigma)$,
- a mapping $fctn : Sig \to \mathcal{P}Id$ with a signature $sig(f) = s_1 \times \ldots \times s_n \to s$ for each function $f \in fctn(\Sigma)$ with $s_1, \ldots, s_n, s \in (sorts(\Sigma) - \{state\})$
- the state sort $state$ as $fctn(\Sigma) \to sig(fctn(\Sigma))$
- a mapping $proc : Sig \to \mathcal{P}Id$ with a signature $sig(p) = state \times s_1 \times \ldots \times s_m \to state \times s$ for each procedure $p \in proc(\Sigma)$.

The sorts $sorts(\Sigma)$ include basic data sorts $s_1, s_2, \ldots$ representing data types and the distinguished sort $state$ to represent the object state. A state is a dynamic (i.e. modifiable) mapping from function identifiers to functions. A variable (in the sense of programming languages) is modelled as a nullary function. There are two forms of operations: functions $fctn(\Sigma)$ and procedures $proc(\Sigma)$. Functions do not change the state, procedures might do. Procedures might modify function definitions. The operations constitute the services provided by the object.

Σ-terms are permissable syntactic entities for a given signature Σ. Let $X = (X_s)_{s \in sorts(\Sigma)}$ be a $sorts(\Sigma)$-sorted set of free variables. A **Σ-term** of data sort s over X_s is thus every $x \in X_s$, every $F(t_1, \ldots, t_n)$ with $f : s_1 \times \ldots \times s_n \to s$, and every $\pi_2(p(st, t_1, \ldots, t_n))$ with $p : state \times s_1 \times \ldots \times s_n \to state \times s$ and t_i Σ-term of data sort $s_i \in S$, $i = 0, \ldots, n$; $st : state$[1]. **Σ-terms** of state sort $state$ over X_s are identifiers st to denote the state, and every $\pi_1(p(st, a_1, \ldots, a_m))$ for $p \in proc(\Sigma)$ with $p : state \times s_1 \times \ldots \times s_m \to state \times s$, $st : state$ and $a_i : s_i$. With $T(\Sigma, X)_s$ we denote the set of Σ-terms of sort s including variables.

A **state signature morphism** σ is a structure-preserving mapping between object signatures where $\sigma_{Sort}(state)$ is the identity. Signature morphisms will be needed later on to express how components in a composition can be adapted syntactically.

A structure is called a **Σ-object** for an object signature Σ, if it has

- a carrier set S including an undefinedness symbol $\perp$ for each sort s,
- a function of type $S_1 \times \ldots \times S_n \to S$ for each function with signature $s_1 \times \ldots \times s_n \to s$,

[1]The symbols π_1 and π_2 denote projections.

- a carrier set $State$ for sort $state$ containing total assignments $\mathcal{P}Id \to F$ if F is the set of functions that match the signatures of functions,
- a function of type $(State \times S_1 \times \ldots \times S_m) \to (State \times S)$ for each procedure symbol in $proc(\Sigma)$ with the corresponding signature.

This is object-based according to Wegner's classification [Weg90]. We have not modelled an object identity (which is considered optional by Wegner). Carrier sets for basic sorts and functions for function symbols form an algebra.

A state ST of sort $state$ can be modified by procedures. We need a substitution mechanism on states. The expression $substitute(ST, x \mapsto v)$ substitutes in the mapping ST the former binding for an identifier x by a binding of x to the value v.

$$substitute(ST, x \mapsto v)(z) = \begin{cases} ST(z) & for\ z \neq x \\ v & for\ z = x \end{cases}$$

So far, we have defined syntactical constructs (signatures and signature morphisms) and semantical constructs (objects and state substitutions). Now, we relate syntax and semantics. A mapping v from identifiers to semantical entities is called a **valuation**; its inductively defined extension v^* for arbitrary terms is called an **interpretation**. Each term depends on the current state. Let $ST : state$ be a state.

$$v(ST, x) := ST(x)$$

The value of an identifier x is stored in ST – remember that the state associates function symbols and functions.

$$\begin{array}{lcl} v^*(ST, f(a_1, \ldots, a_n)) & := & ST(f)(v^*(ST, a_1), \ldots, v^*(ST, a_n)) \\ v^*(ST, p(ST, a_1, \ldots, a_m)) & := & [\![p]\!](ST, v^*(ST, a_1), \ldots, v^*(ST, a_m)) \end{array}$$

$[\![p]\!]$ denotes the function that implements the procedure p.

We introduce a **definedness** predicate D for terms t: $D(ST, t) = true$, if $v^*(ST, t) \neq \bot$ for some state ST. Termination of operations will be expressed using the definedness predicate.

Example 2.1 *An object signature for a stack object includes the following operation signatures:*

$$\begin{array}{ll} push: & state \times s \to state \\ pop: & state \to state \\ top: & \to s \end{array}$$

with sorts state for the stack and s for elements. push and pop are procedures, top is a function.

A stack object consists of a) a carrier set S for elements of sort s which are stored on the stack, b) the state of the object (here the identifier top mapped to a function implementing it) and c) functions implementing pop and push.

In the next section, we will introduce a simple state transition logic allowing us to specify functions and procedures in an abstract way.

3 Component Specification

Modal logics are logics with a notion of state (or time). Modal logics allow us to specify and reason about states and state transitions. Dynamic logic is a modal logic which makes terms of the underlying object language explicit in the logic. We present a simplified dynamic logic which allows the abstract specification of objects by describing the behaviour of their operations. Abstraction of implementation details is the main requirement for a component notion.

Equations and the so-called modal box-operator will form the two basic constructs of the formula language. A Σ-**equation** has the form $t =_s t'$ with $t, t' \in T(\Sigma, X)_s$ for a data sort s. A Σ_{state}-**equation** has the form $t =_{state} t'$ with $t, t' \in T(\Sigma, X)_{state}$. The set of **well-formed formulas** WFF(Σ) is the smallest set with the following properties:

- every Σ-equation and every Σ_{state}-equation is in WFF(Σ),
- if $\phi, \psi \in$ WFF(Σ), then $\phi \rightarrow \psi, \phi \wedge \psi, \phi \vee \psi, \neg\phi \in$ WFF(Σ),
- if P a procedural term and $\phi \in$ WFF(Σ), then $[P]\ \phi \in$ WFF(Σ).

The box operator $[P]\ \phi$ distinguishes this simplified dynamic logic from a standard first-order logic. Assuming that P is a procedure application, its meaning is that, if P terminates, ϕ shall be a property of the state in which the procedure P terminates.

A notion of satisfaction relates formulas and Σ-objects, i.e. it says when a formula is true (or holds in an object). Assume a Σ-object A, a state $ST \in State$ and a Σ-formula ϕ. A **satisfies** ϕ in state ST, or $A, ST \models \phi$, is defined by

- $A, ST \models t =_s t'$ iff $v^*(ST, t) = v^*(ST, t')$ for a data sort s
- $A, ST \models t =_{state} t'$ iff $v^*(\pi_1(v^*(ST, t)), f(a_1, .., a_{n_c})) = v^*(\pi_1(v^*(ST, t')), f(a_1, .., a_{n_c}))$ for any term $f(a_1, .., a_{n_c})$
- $A, ST \models \phi \rightarrow \psi$ iff $\neg\phi \vee \psi$ (with the usual definitions for $\neg$ and $\vee$, omitted here)
- $A, ST \models [\ P\ ]\ \phi$ iff $D(P) \rightarrow A, \pi_1(v^*(ST, P)) \models \phi)$

Note that the definition of the equality for the state sort is observationally oriented. Two procedure applications are equal (observationally congruent) iff all function applications in the new states yield the same result for both procedures. The functions act as observers on the state.

Example 3.1 *A classical example of illustrating the box operator is the abstract specification of the $push$ procedure of a stack object.*

$$[push(st, b)]\ top() = b$$

The behaviour of $push$ is expressed by the 'observer' top. If the element b is pushed onto the stack st, then b will become the new top element.

Now, we shall address the definition of a component. We use the definition from [NM95]. A component is an abstraction of a software structure that may be used to built bigger systems, while hiding the implementation details of the structure. We shall formulate components (abstracting objects) using the state transition logic as the specification notation. As a first step, we introduce an **object specification** as a pair $\langle \Sigma, E \rangle$ consisting of a signature Σ and a set of well-formed Σ-formulas $E \subseteq \text{WFF}(\Sigma)$. Later on, we will formally define a component as a parameterised object specification.

Each object specification denotes a class of objects which satisfy the constraints formulated by the formulas, called the models of the specification. The class of all Σ-objects is denoted with $Obj(\Sigma)$. The **models** $mod(\langle \Sigma, E \rangle)$ of a specification $\langle \Sigma, E \rangle$ are denoted by their model class $\{A \in Obj(\Sigma) \mid A \models \phi \text{ for all } \phi \in E\}$.

We do not present a full inference system, but one inference rule shall be introduced. This rule will play a crucial role in our approach to reasoning about components and component composition. The consequence rule CONS helps to prove pre/postcondition specifications:

$$\frac{\phi \to \phi_1, \;\; \phi_1 \to [P]\, \psi_1, \;\; \psi_1 \to \psi}{\phi \to [P]\, \psi} \quad [\text{CONS}]$$

This rule will be useful in the definition of a constructive variant of an implementation relation between object specifications. The validity of the rule should be clear from the definition of the satisfaction relation.

4 Implementation and Refinement

The foundation for the composition framework shall be laid in this section. We present two relations between object specifications and how they relate to each other. The first relation, called a refinement, is closely related to the CONS rule presented in the previous section. The second relation, called an implementation, is based on the semantic concept of model class inclusion. The implementation will be used to define the internal correctness of components and also the correctness of component composition. The second relation is the more powerful one, but difficult to prove. We will show that the refinement is much easier to use in proofs. It is, as we will show, a good approximation to the implementation. Therefore, the combination of implementation and refinement constitutes a flexible and easy-to-use framework for specifying and reasoning about components and component composition. In [Pah00] we have presented an application of these two relations, implementation and refinement, in a slightly different framework. There, the underlying semantic structures are Abstract State Machines ASM, foundation of a specification notation developed to specify dynamic state-based systems [Gur93, Gur97]. [Pah00] develops a refinement calculus geared towards the development of ASMs.

Let us now define the refinement.

Definition 4.1 *Let p_1, p_2 be procedural terms with $\phi_{p_1} \to [p_1]\, \psi_{p_1}$ and $\phi_{p_2} \to [p_2]\, \psi_{p_2}$ as their respective specifications. The* **refinement** *relation $p_1 \underset{\text{op}}{\overset{\text{R}}{\rightsquigarrow}} p_2$ holds, if*

$$D(p_1) \wedge D(p_2) \to (\phi_{p_1} \to \phi_{p_2} \wedge \psi_{p_2} \to \psi_{p_1})$$

We will assume terminating (defined) procedure applications for this definition such that by execution of the procedure the postcondition can always be established. This can be found in the literature as the combination of the 'weaker precondition' and the 'stronger postcondition' rule, see e.g. [Mor94]. A module specification shall be considered as correctly implemented, if all constituent procedures are correctly implemented. Note the difference between the consequence rule CONS, as presented earlier on, and the refinement rule here. CONS is a derivation rule guaranteeing partial correctness.

Definition 4.2 *Let sp_1 and sp_2 be object specifications with signature Σ. The* **refinement** *relation $sp_1 \overset{R}{\leadsto} sp_2$ holds, if for all $op \in fctn(\Sigma) \cup proc(\Sigma)$ the relation $op_{sp_1} \underset{op}{\overset{R}{\leadsto}} op_{sp_2}$ holds. op_{sp_1} and op_{sp_2} denote the different specifications of op in sp_1 and sp_2*

Example 4.1 *The refinement shall be illustrated using again the push operation. A bounded stack shall be refined to an unbounded one.*

$$\neg full(st) \rightarrow [push(st, b)]\ top() = b$$

says that push can only be executed if the stack st is not full. The refined specification

$$true \rightarrow [push(st, b)]\ top() = b$$

says that push can be executed at any time. Since $\neg full(st) \rightarrow true$ is a tautology, this represents a valid refinement. Instead of weakening the precondition, the postcondition could have been strengthened, e.g. by conjoining more conditions.

Now, we define an implementation relation.

Definition 4.3 *Let sp_1, sp_2 be object specifications. The* **implementation** *relation $sp_1 \overset{I}{\leadsto} sp_2$ holds, if $sig(sp_2) = sig(sp_1)$ and $Mod(sp_2) \subseteq Mod(sp_1)$.*

The implementation relation is based on the purely semantical criterion of model class inclusion. The implementation between specifications defined by model class inclusion is standard in algebraic specification, see [Wir90].

Lemma 4.1 *The relations $\overset{I}{\leadsto}$ and $\overset{R}{\leadsto}$ form a partial ordering:*

- *reflexivity: $sp_1 \leadsto sp_1$,*
- *antisymmetry: $sp_1 \leadsto sp_2 \wedge sp_2 \leadsto sp_1 \Rightarrow sp_1 = sp_2$,*
- *transitivity: $sp_1 \leadsto sp_2 \wedge sp_2 \leadsto sp_3 \Rightarrow sp_1 \leadsto sp_3$.*

where $\leadsto$ is $\overset{I}{\leadsto}$ or $\overset{R}{\leadsto}$ and sp_1, sp_2, sp_3 are object specifications.

The transitivity of relations in the vertical dimension is called the **vertical composition property**. Vertical development means implementation. In general, more abstract specifications are made more concrete by making design decisions, which is on the semantical level reflected by a smaller model class. There are less models if requirements are added or strengthened.

With the following theorem we will establish a relationship between the refinement and the implementation. The constructive refinement relation $\overset{R}{\leadsto}$ will be shown as a specialisation of the implementation relation $\overset{I}{\leadsto}$. In order to relate the two relations $\overset{I}{\leadsto}$ and $\overset{R}{\leadsto}$, we assume the three simplifications.

- There are no explicit invariants inv[2]. They will be associated to procedure definitions: reformulate $\phi \rightarrow [P]\ \psi$ to $\phi \rightarrow [P]\ \psi \wedge inv$.

[2] Underlying data type and function specifications are considered as invariant.

- There are only procedures. Functions are specified by normal first-order formulas and can be treated as invariants.
- Every procedure is defined by only one formula.

These constraints do not restrict the expressivity of the notation. We can always obtain a specification following these constraints from any specification by semantics-preserving reformulations (as indicated).

Theorem 4.1 *Let Σ be an object signature and sp_1 and sp_2 be Σ-specifications under the assumptions mentioned above. Then $sp_1 \overset{R}{\leadsto} sp_2$ implies $sp_1 \overset{I}{\leadsto} sp_2$.*

The implication $sp_1 \overset{R}{\leadsto} sp_2$ *implies* $sp_1 \overset{I}{\leadsto} sp_2$ can *not* be established, if the models contain functions which may not terminate, those formulas describing states are not satisfiable, or pre-conditions describe states not reachable from a given initial state. Making these three conditions assumptions of the above theorem, we would get equivalence between the two relations. Thus, we see that $\overset{R}{\leadsto}$ is a good criterion (a good approximation) for $\overset{I}{\leadsto}$.

5 Components

In the next section, we will explain how the implementation can be used to define concepts for component composition. In this section, we define components as parameterised specifications. Parameters will be constrained, i.e. input requirements can be specified abstractly. In order to describe an internal correctness notion for components, the implementation shall also be used. A parameterisation concept using formal import to specify requirements for actual parameters is the basis of the component concept presented here. The import will be a part of the component specification. The import will be separated from the component body. The relationship of the import to the body is characterised by a notion of correctness.

Horizontal development is a notion for structuring specifications in the large and techniques to make this structuring feasible [EM85, EM90, Wir90, Hen91, HN92, Wir95, Goe93]. Component composition is one possible technique. Other techniques (not investigated here) include operators on components [PPP94, AAZ93, CL94, CHC90]. Our central aim is to use the implementation in the definition of the component composition technique. We will adapt components to satisfy import requirements in compositions, using syntactic and semantic mechanisms.

Let us assume that a given system is decomposed, or modularised, into components. Each of these components can be realised by hand or by reusing appropriate components for implementation. If, on the level of more abstract specifications, a system of components is correctly composed, then this correctness shall be preserved on the level of implementations of these abstract specifications. The correctness of compositions on the horizontal level shall be preserved. Thus, this property is called the *horizontal composition property*. It shall now be illustrated. Let C_1 and C_2 be components. $C_1(C_2)$ expresses, that C_2 is the actual parameter of C_1. The composition $C_1(C_2)$ is correct, if C_2 satisfies the requirements of the formal import of C_1. Let us now assume two more abstract components S_1 and S_2, and two less abstract (implemented) components I_1 and I_2. If the composition of abstract components $S_1(S_2)$ is correct, then the horizontal composition property requires that, if $S_1 \overset{I}{\leadsto} I_1$ and $S_2 \overset{I}{\leadsto} I_2$ are implementations, then $I_1(I_2)$ is a correct composition. This important property shall be realised in our framework.

5.1 Prerequisites

We need some basic constructs for our component composition approach. Essentially, we introduce the class of specifications as a complete partial ordering (cpo). The ordering relation on specifications is the implementation relation. Sig is the class of all signatures. $\bot_{Sig}$ is the empty signature. The class $Spec$ comprises all possible models of specifications over Sig:

$$Spec := \{ \langle \Sigma, C \rangle \mid \Sigma \in Sig,\ C \subseteq Obj(\Sigma) \}$$

$Obj(\Sigma)$ is the class of all objects, i.e. semantic structures with state for a given signature Σ. $Spec_\bot := Spec \cup \{\bot_{Spec}\}$ is the extension of $Spec$ by the bottom element $\bot_{Spec} = \langle \bot_{Sig}, Obj(\bot_{Sig}) \rangle$. $sp_{|\Sigma}$ denotes the restriction of specification $sp = \langle \Sigma', E \rangle$ to the Σ-relevant parts, if $\Sigma \subseteq \Sigma'$:

$$sp_{|\Sigma} := \langle \Sigma, \{ \phi \in E \mid \phi \in \text{WFF}(\Sigma) \} \rangle$$

Two projection functions sig and Mod shall be provided on $Spec_\bot$:

$$\begin{array}{lcl} sig & : & Spec_\bot \ \rightarrow \ Sig \ \ with \ \ sig(\langle \Sigma, C \rangle) := \Sigma \\ Mod & : & Spec_\bot \ \rightarrow \ \{ C \subseteq Obj(\Sigma) \mid \Sigma \in Sig \} \ \ with \ \ Mod(\langle \Sigma, C \rangle) := C \end{array}$$

Every specification SP is associated with an element $sp \in Spec_\bot$, denoted by $\mathcal{M}(SP)$, where sp is defined by $sp = \langle sig(SP), Mod(SP) \rangle$.

$Mod(SP)|_\rho$ is the reduct with respect to a signature morphism $\rho : \Sigma \rightarrow \Sigma'$ where every model in $Mod(SP)$ is reduced to the elements of signature Σ. Let SP be a specification with signature Σ' and let ρ a signature morphism $\rho : \Sigma \rightarrow \Sigma'$. Then $SP|_\rho$ is a specification with semantics $\langle \Sigma, Mod(SP)|_\rho \rangle \in Spec_\bot$. This notation allows us to express reducts on the level of specifications. This construction is also known as a *forgetful mapping*.

Let the relation $\sqsubseteq_{Spec}$ on $Spec_\bot$ be defined as follows:

$$\begin{array}{lll} \langle \Sigma, C \rangle & \sqsubseteq_{Spec} & \langle \Sigma', C' \rangle \ \ \textit{iff} \ \ \Sigma = \Sigma' \ and \ C' \subseteq C \\ \bot_{Spec} & \sqsubseteq_{Spec} & sp \ \ for \ all \ sp \ in \ Spec_\bot \end{array}$$

The relation $\sqsubseteq_{Spec}$ is based on model class inclusion. We will denote this relationship between SP and SP' with $SP \overset{I}{\rightsquigarrow} SP'$, and call $\overset{I}{\rightsquigarrow}$ an *implementation relation*. This corresponds to the definition of $\overset{I}{\rightsquigarrow}$ we gave earlier on. $(Spec_\bot, \sqsubseteq_{Spec})$ is a cpo with $\bot_{Spec}$ as the bottom element. Chains are based on the inclusion hierarchy.

5.2 Components

A component is a parameterised object specification. A parameterised specification consists of a parameter restriction and a body. Elements of the formal parameter, or import restriction, are intended to be used in the specification of the body. Properties of the import specification should be preserved. An example could be a parameterised stack, where the import restriction describes the element type in its general properties, i.e. an ordering on the elements could be required, and should be preserved by the stack specification.

Definition 5.1 *A parameterised specification C, called a* **component***, is a pair $C = \langle SP_I, SP_B \rangle$ of object specifications SP_I and SP_B. SP_I is called* import specification *and SP_B is called* body specification.

A component $C = \langle SP_I, SP_B \rangle$ can be formulated as a λ-expression $\lambda X : SP_I.\ SP_B$ where SP_I is a parameter restriction and SP_B is a specification. X is a free variable. X can be used in SP_B. X must not be used in SP_I. The actual parameter will be assigned to X.

Definition 5.2 *Let $C = \langle SP_I, SP_B \rangle$, specified as $\lambda X : SP_I.\ SP_B$, be a component. Then the lambda expression $(\lambda X : SP_I.\ SP_B)(SP)$ denotes a specification, if SP does. The component C is called* **correct**, *if*

$$sig(SP_I) \subseteq sig(SP_B) \ \wedge \ SP_B|_{sig(SP_I)} \overset{\perp}{\rightsquigarrow} SP_I$$

The component $C = \langle SP_I, SP_B \rangle$ is correct, if $Mod(SP_I) \subseteq Mod(SP_B)|_{sig(SP_I)}$. Every model of the import can be extended to a model of the body. The body preserves the import, expressed throught the notion of correctness. The condition $sig(SP_I) \subseteq sig(SP_B)$ indicates how the formal parameter can be used in the body specification. It is expected that imported operations are part of the new specification. New operations can also be provided. In case some operations shall be hidden, an explicit export interface is needed. This can be constructed by using again the implementation as the correctness criterion: an export interface is correct if it is syntactically a subset of the body specification and if, semantically, the body implements the export.

A component is semantically defined as a function from $sig(SP_I)$-models to $sig(SP_B)$-models. The formal framework for these semantics is the *$\lambda\pi$-calculus* [Fei89], a λ-calculus which provides flexibility in the treatment of actual parameters. $env \in Env$ is an environment in which free variables $X \in Var$ are bound to their values, i.e. $env : Var \rightarrow Spec_\perp$. These values are the actualisations of the formal import. Env is the set of all environments. $\mathcal{M}^{par} : Comp \rightarrow (Env \rightarrow (Spec_\perp \rightarrow Spec_\perp))$ shall denote the semantics of a λ-application for a component. $Comp$ denotes the class of all components.

Definition 5.3 *Let $\mathcal{M}$ be the semantic function for non-parameterised specifications. $\mathcal{M}^{par}(PS) : Env \rightarrow Spec_\perp \rightarrow Spec_\perp$ is the semantic function for components (parameterised specifications) C represented by $\lambda X : SP_I.\ SP_B$ where SP_I, SP_B are specifications:*

$$\mathcal{M}^{par}(\lambda X : SP_I.\ SP_B)(env)(sp) := \begin{cases} \mathcal{M}(SP_B)(env[X \mapsto sp]) & \\ \qquad\qquad if\ sp \neq \perp_{Spec}\ and\ \mathcal{M}(SP_I)(env) \sqsubseteq_{Spec} sp & \\ \perp_{Spec} \quad otherwise & \end{cases}$$

$env[X \mapsto sp]$ describes the binding of X to an actual value sp in env.

The actual import $sp \in Spec_\perp$ has to satisfy the import restriction SP_I, expressed in the model semantics by $\mathcal{M}(SP_I)(env) \sqsubseteq_{Spec} sp$. This means that under consideration of the environment env, sp shall be an *implementation* of the parameter restriction SP_I. The semantics $\mathcal{M}(SP)$ of a primitive specification SP is the associated element $\langle \Sigma, C \rangle$ from $Spec_\perp$. If X is the only free variable, then the definition of a component equals the implementation of SP_I by sp, i.e. $Mod(sp) \subseteq Mod(SP_I)$ or $SP_I \overset{\perp}{\rightsquigarrow} sp$.

Finally, we show that the new syntactical construct of parameterised object specifications is semantically defined in a constructive way as an order-preserving function. The implementation is based on the ordering $\sqsubseteq$ on $Spec_\perp$ which is preserved.

Lemma 5.1 *Let $C \equiv \lambda X : SP_I.\ SP_B$ be a component with X not in SP_I. Then $\mathcal{M}^{par}(PS)(env) : Spec_\perp \rightarrow Spec_\perp$ is a strict and monotonous function on $Spec_\perp$ for every $env \in Env$.*

Note, that the implementation relation was used twice in the previous definitions: in the definition of the internal correctness of components and also in the correctness of the actualisation of an import restriction.

Example 5.1 *Let $C = \langle Pair, Stack \rangle$ be a component specifying a stack of pairs. $Pair$ is a parameter restriction. It shall limit element to be stored on the stack to pairs of values. The signature of $Pair$ could look like:*

$$\begin{array}{ll} pair: & e \times e \to p \\ left: & p \to e \\ right: & p \to e \end{array}$$

Additionally, $Pair$ will have some axioms such as

$$left(pair(x,y)) = x \text{ and } right(pair(x,y)) = y.$$

The stack specification $Stack$ shall be the union of the stack described earlier on and the pair specification. Signatures and axioms are simply added together. Note, that the component C provides stack and pair operations. Since we have reused $Pair$ in $Stack$, the component C is correct.

6 Composition of Components

We will introduce the *composition of components* in this section. Composition means plugging components together. Composition denotes here the actualisation of a formal import of one component by the service specifications of another.

Definition 6.1 *Let $C = \langle SP_I, SP_B \rangle$ and $C' = \langle SP'_I, SP'_B \rangle$ be components. A* **composition** *of components is defined by*

$$compose\ \langle SP_I, SP_B \rangle\ with\ \langle SP'_I, SP'_B \rangle\ via\ \rho\ := \ \langle SP'_I, (\lambda X : SP_I . SP_B)(SP'_B|_\rho) \rangle$$

where ρ shall be a signature morphism $\rho : sig(SP_I) \to sig(SP'_B)$.

The composition $compose\ \langle SP_I, SP_B \rangle\ with\ \langle SP'_I, SP'_B \rangle\ via\ \rho$ is called **correct**, *if $SP_I \overset{I}{\rightsquigarrow} SP'_B|_\rho$.*

In the definition, C imports from C'. All imports from SP_I have to be satisfied, but not all exports of SP'_B have to be used. The signature morphism ρ has to describe this adaptation. The actual import $SP'_B|_\rho$ has at least to fulfill the formal import SP_I – or it can be better. We do not require an exact matching – neither syntactically nor semantically. This definition using a signature morphism for syntactical adaptation and an implementation for property preservation provides a high degree of flexibility in combining components.

Besides the flexibility in the relation between provided export of one component and actually required import of another component, composition also preserves correctness of components.

Theorem 6.1 *If $\langle SP_I, SP_B \rangle$ and $\langle SP'_I, SP'_B \rangle$ are correct components, and*

$$compose\ \langle SP_I, SP_B \rangle\ with\ \langle SP'_I, SP'_B \rangle\ via\ \rho$$

is a correct composition with $sig(SP'_B) \subseteq sig((\lambda X : SP_I.\ SP_B)(SP'_B|_\rho))$, then the composed component is correctly composed.

$$\begin{tikzcd}
\langle SP_I, SP_B\rangle \arrow[d, squiggly, "comp"] & & \langle SP_I, SP_B\rangle \arrow[d, squiggly, shift left=4] \\
\langle IM_I, IM_B\rangle & \text{if} & \langle IM_I, IM_B\rangle \arrow[u, squiggly, shift left=4]
\end{tikzcd}$$

Figure 1: Implementation of Components

The implementation was only defined for simple object specifications. The implementation relation $\overset{\mathrm{I}}{\rightsquigarrow}$ shall now be extended for components.

Definition 6.2 *Let $SP = \langle SP_I, SP_B\rangle$ and $IM = \langle IM_I, IM_B\rangle$ be correct components. The component $\langle IM_I, IM_B\rangle$ is an* **implementation** *of $\langle SP_I, SP_B\rangle$, or $\langle SP_I, SP_B\rangle \overset{\mathrm{I}}{\rightsquigarrow}_{comp} \langle IM_I, IM_B\rangle$, if*

- *$sig(IM_B) = sig(SP_B)$ and $Mod(IM_B) \subseteq Mod(SP_B)$, and*
- *$sig(IM_I) = sig(SP_I)$ and $Mod(SP_I) \subseteq Mod(IM_I)$.*

Again, the definition is based on implementations $SP_B \overset{\mathrm{I}}{\rightsquigarrow} IM_B$ and $IM_I \overset{\mathrm{I}}{\rightsquigarrow} SP_I$. This is illustrated in Figure 1.

Example 6.1 *The parameter restriction $Pair$ from the previous example can be actualised by, for instance, complex number providing a pairing operator $complex : int \times int \rightarrow cplx$ and projections $re : cplx \rightarrow int$ and $im : cplx \rightarrow int$. A standard specification of complex numbers, which shall be called $Cplx$, certainly satisfies the $Pair$-requirements. Additionally, arithmetic operations such as addition can also be provided by $Cplx$. In order to compose stacks with complex numbers, only the pairing operation and the projections are relevant. We need to actualise the formal parameter elements $pair$, $left$ and $right$ with the actual parameter elements $complex$, re and im, respectively. We can match the signatures easily. Under the assumption that complex numbers as constructed as pairs, the semantics also matches. An implemention between $Pair$ and $Cplx$ requires the same signatures (achieved by reducing $Cplx$ to the signature of $Pair$) and model class inclusion. The reduct of $Cplx$ should have a smaller model class than $Pair$, i.e. it should satisfy at least the $Pair$-requirements. Thus, the implementation*

$$Pair \overset{\mathrm{I}}{\rightsquigarrow} Cplx|_{sig(Pair)}$$

holds, meaning that $Cplx$ satisfies the $Pair$ requirements. Thus, the composition is correct.

The correctness, which is essentially implementation, can be proven by a specifier using the refinement as an approximation to the implementation (as shown above in Section 4). The refinement is the tool for proving correctness notions in component specification and composition.

The following theorem formulates the central result of this section, the *horizontal composition property* for components. Elements of the horizontal composition property are illustrated in Fig. 2.

Theorem 6.2 *Let us assume:*

- *correct components $\langle SP_I^1, SP_B^1\rangle$, $\langle IM_I^1, IM_B^1\rangle$, $\langle SP_I^2, SP_B^2\rangle$, $\langle IM_I^2, IM_B^2\rangle$,*

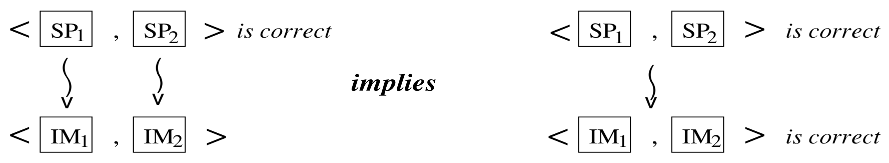

Figure 2: Horizontal Composition Property

- *the component implementations*

$$\langle SP_I^1, SP_B^1\rangle \overset{I}{\leadsto}_{comp} \langle IM_I^1, IM_B^1\rangle \text{ and } \langle SP_I^2, SP_B^2\rangle \overset{I}{\leadsto}_{comp} \langle IM_I^2, IM_B^2\rangle$$

- *a signature morphism $\rho : sig(SP_I^1) \rightarrow sig(SP_B^2)$,*
- *the composition compose $\langle SP_I^1, SP_B^1\rangle$ with $\langle SP_I^2, SP_B^2\rangle$ via ρ is correct.*

Then the following holds:

- *$\rho' : sig(IM_I^1) \rightarrow sig(IM_B^2)$ with $\rho' := \rho$ is a signature morphism,*
- *the composition compose $\langle IM_I^1, IM_B^1\rangle$ with $\langle IM_I^2, IM_B^2\rangle$ via ρ' is correct,*
- *the component implementation relation holds between the compositions:*

$$\begin{array}{ll} & compose\ \langle SP_I^1, SP_B^1\rangle\ with\ \langle SP_I^2, SP_B^2\rangle\ via\ \rho \\ \overset{I}{\leadsto}_{comp} & \\ & compose\ \langle IM_I^1, IM_B^1\rangle\ with\ \langle IM_I^2, IM_B^2\rangle\ via\ \rho' \end{array}$$

The theorem guarantees that the correctness of compositions on the abstract level is preserved by their implementations. This allows components to be specified and composed on an abstract level and then implemented separately preserving the correctness of the whole system. There is no need to prove the correctness for implementations.

7 Related Work

During the last decade, algebraic methods have been used to support state-based software development. The classical notion of algebraic signatures is extended by introducing hidden, non-observable sorts representing an internal state. One of the well-known approaches to these problems is the hidden algebra framework developed by the OBJ-group, see e.g. [Gog99, GM00]. There, an equational, behaviourally oriented framework for reasoning is established. A hidden signature [Gog99] is a signature with disjoint hidden and visible sorts. The signature ensures data encapsulation: a hidden signature can only be embedded into a visible one, if no new operations are added. Components and transitions depend on the state. Our state signatures are clearly hidden signatures in the sense of [GM00]. A hidden algebra is an algebra which satisfies a hidden specification based on a hidden signature. A hidden algebra should encapsulate an algebra as a substructure (reduct) which represents the data part. Our objects are hidden algebras in this sense. Our objects are slightly less general since we only provide one hidden sort, the state sort $state$. One the other hand, our definition is more flexible in defining procedures, which can have a result value in our framework. Goguen and Malcolm's work show the way how to move from an object-based towards and object-oriented framework. Order sorted algebras could be used to model inheritance.

The naming of one of the relations – refinement – indicates its similarity to rules in the Refinement Calculus [Bac88, Mor90, Mor94]. Our refinement is derived from the consequence rule for dynamic and Hoare logics, see [Fra92, Cou90]. The consequence rule is a rule for reasoning about partial correctness. We have realised a refinement rule for total correctness, i.e. partial correctness plus termination, in order to ressemble the rules in the Refinement Calculus. Therefore, we have introduced a definedness predicate to make this explicit, see [MB98] for a treatment of partiality, definedness and termination in the Refinement Calculus.

The use of formal methods in component specification and composition is not new [Nor98, BS97, Wec97]. Weck [Wec97] discusses contracts (pre- and postconditions) as a means for component composition. Büchi and Sekerinski [BS97] present an interface definition language for components based on pre- and postconditions. They explore the idea of using refinement in their work, but their framework is less general than ours. We have realised a notion of observable behaviour through our definition of equality on the sort $state$. A similar idea of making observable behaviour the basic notion of component composition is presented by Nordhagen in her thesis [Nor98].

We have provided a framework for component composition. Similar aims are pursued by Nierstrasz et.al. [LSNA97, LAN00]. The main difference is that Nierstrasz' group considers objects as processes, using process calculi in the formalisation of the component and composition framework.

Another approach which relates to our work is described in [Goe93] where a component framework is developed. There, components are defined as a very abstract notion which fits various kinds of abstractions such as procedural abstractions or module abstraction. This work also considers state-based systems, but it does not provide a proof support in the way we do.

In the area of algebraic specification, we can find various approaches to specification in the large and in particular the composition of specification and operators on specifications [EM90, Wir95, HN92, PPP94].

8 Conclusions

Most of the current object-oriented programming languages provide only an ad-hoc collection of mechanisms for constructing and composing objects. A more structured and formally defined approach is needed. The composition framework we have presented here can be used as the basic foundation for such an approach.

We have considered components as black-box entities that encapsulate service implementations behind a well-defined interface. An important feature of our approach is that implementation and composition are realised as orthogonal concepts, connected only by the horizontal composition property. Our approach to composition is flexible, since it requires neither an exact syntactical nor semantical match between component interfaces.

The main focus of our paper was on mechanisms for reasoning about component composition. We have used an implementation relation in the definition of various correctness notions, including an internal component correctness and a correctness for the composition. This powerful relation is supported by a so-called refinement relation. This relation has two important characteristics. Firstly, it is easy to handle since it reduces the complexity of proofs in dynamic logic to non-modal first-order logic. Secondly, it is also a good approximation of the implementation. These two properties together make it an ideal proof support tool for components and component composition.

In the future, we want to build up upon our formal foundations for a composition framework and develop a composition language. In order to support a full composition framework, more (different kinds of) connectors are sought. The composition mechanism we have been looking at is connecting interfaces via parameterisation. This is the standard way of composing software components. Nevertheless, other forms of composition could be looked at, e.g. including communication between components.

Acknowledgements

Many thanks to go the anonymous referees for their constructive and helpful comments.

References

[AAZ93] D. Ancona, E. Astesiano, and E. Zucca. Towards a Classification of Inheritance Relations. In U. Lipeck and G. Koschorrek, editors, *IS-CORE'93 Workshop, Informatik-Berichte 01/93*, pages 90–113. Universität Hannover, 1993.

[Bac88] R.J.R. Back. A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25:593–624, 1988.

[BS97] M. Büchi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In *Proceedings 2nd International Workshop on Component-Oriented Programming WCOP '97*. Turku Center for Computer Science, General Publication No.5-97, Turku University, Finland, 1997.

[CHC90] W.R. Cook, W.L. Hill, and P.S Canning. Inheritance is not Subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135. ACM Press, 1990.

[CL94] Y. Cheon and G.T. Leavens. The Larch/Smalltalk Interface Specification Language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, 1994.

[Cou90] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 841–996. Elsevier Science Publishers, 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Modules and Constraints, EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.

[Fei89] L.M.G. Feijs. The calculus $\lambda\pi$. In *Algebraic Methods: Theory, Tools and Applications*, pages 307–328. Springer-Verlag, 1989.

[Fra92] Nissim Francez. *Program Verification*. Addison Wesley, 1992.

[GM00] J. Goguen and G. Malcolm. A Hidden Agenda. *Theoretical Computer Science*, 2000. Special Issue on Algebraic Engineering – to appear.

[Goe93] M. Goedicke. On the Structure of Software Description Languages: A Component Oriented View. Habilitationsschrift, Forschungsbericht 473, Universität Dortmund, Fachbereich Informatik, 1993.

[Gog99] J. Goguen. Hidden Algebra for Software Engineering. In *Proceedings Conference on Discrete Mathematics and Theoretical Computer Science, Auckland, New Zealand*, pages 35–59. Australian Computer Science Communications, Volume 21, Number 3, 1999.

[Gur93] Y. Gurevich. Evolving Algebras: An Attempt to Discover Semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.

[Gur97] Y. Gurevich. May 1997 Draft of the ASM Guide, 1997. University of Michigan EECS Department Technical Report CSE-TR-336-97.

[Hen91] R. Hennicker. Context Induction: A Proof for Behavioural Abstractions and Algebraic Implementations. *Formal Aspects of Computing*, 3:326–345, 1991.

[HN92] R. Hennicker and F. Nickl. A Behavioural Algebraic Framework for Modular System Design with Reuse. Bericht LMU - 9206, Informatik, Ludwig-Maximilians-Universität München, 1992.

[LAN00] M. Lumpe, F. Achermann, and O. Nierstrasz. A Formal Language for Composition. In G.T. Leavens and M. Sitamaran, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[LS00] G.T. Leavens and M. Sitamaran. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[LSNA97] M. Lumpe, J.-G. Schneider, O. Nierstrasz, and F. Achermann. Towards a Formal Composition Language. In G.T. Leavens and M. Sitamaran, editors, *Proceedings European Conference on Software Engineering ESEC'97*, pages 178–187. Springer-Verlag, 1997.

[MB98] J.M. Morris and A. Bunkenberg. Partiality and Nondeterminacy in Program Proofs. *Formal Aspects of Computing*, 10:76–96, 1998.

[Mor90] J.M. Morris. Programs from Specifications. In E.D. Dijkstra, editor, *Formal Development of Programs and Proofs*. Addison-Wesley, 1990.

[Mor94] C. Morgan. *Programming from Specification 2e*. Addison-Wesley, 1994.

[NM95] O. Nierstrasz and T.D. Meijler. Requirements for a Composition Language. In P. Ciancarini, O. Nierstrasz, and A Yonezawa, editors, *Object-based Models and Languages for Concurrent Systems*, pages 147–161. Springer-Verlag, 1995.

[Nor98] E.K. Nordhagen. *A Computational Framework for Verifying Object Component Substitutability*. PhD thesis, University of Oslo, November 1998.

[Pah97] C. Pahl. A Model for Dynamic State-based Systems. In A.S. Evans and D.J. Duke, editors, *Proc. Northern Formal Methods Workshop, Sept.'96, Bradford, UK*. Springer-Verlag, 1997.

[Pah00] C. Pahl. Towards an Action Refinement Calculus for Abstract State Machines. In *Proceedings Abstract State Machines ASM'2000, March 2000, Monte Verita, Switzerland.* 2000.

[PPP94] F. Parisi-Presicce and A. Pierantonio. An Algebraic Theory of Class Specification. *ACM Transactions on Software Engineering and Methodology*, 3(2):166–199, April 1994.

[Wec97] W. Weck. Inheritance Using Contracts & Object Composition. In *Proceedings 2nd International Workshop on Component-Oriented Programming WCOP '97*. Turku Center for Computer Science, General Publication No.5-97, Turku University, Finland, 1997.

[Weg90] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *ACM OOPS Messenger*, pages 8–87, 1990.

[Wir90] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 675–788. Elsevier Science Publishers, 1990.

[Wir95] M. Wirsing. Algebraic Specification Languages. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types, 1994*, pages 81–115. Springer-Verlag, 1995.

A Proofs

The proofs of all Lemmas and Theorems are listed here.

Proof Lemma 4.1

- Reflexivity: the relation $\overset{I}{\rightsquigarrow}$ is based on model-class inclusion $\subseteq$, which is reflexive. Analogously, $\overset{R}{\rightsquigarrow}$ is based on implications, which are also reflexive.
- Antisymmetry: as for reflexivity, the underlying constructs model-class inclusion and implication are antisymmetric.
- Transitivity: The implementation $M_1 \overset{I}{\rightsquigarrow} M_2$ holds, if $sig(M_1) = sig(M_2)$ and $mod(M_2) \subseteq mod(M_1)$. The transitivity of $\overset{I}{\rightsquigarrow}$ follows immediately, since $=$ and $\subseteq$ are transitive. Analogously for the refinement $\overset{R}{\rightsquigarrow}$ and the implication. $\square$

Proof Theorem 4.1 Models consist of carrier sets, functions and the mapping $STATE$. It will be shown that model class inclusion only depends on functions which interpret procedures. Thus, the theorem can be proved w.l.o.g. for one procedure.

We need a few definitions. Let a procedure p in sp_1 be specified by $\phi_1 \to [p]\ \psi_1$ and in sp_2 by $\phi_2 \to [p]\ \psi_2$. $Mod^{op}_{sp}(p)$ denotes models of a procedure p, i.e. those functions which interpret a procedure p in a model of a specification sp. A respective satisfaction relation shall be defined. Let $A \in Mod(sp), p^* \in Mod^{op}_{sp}(p)$ and $p \in opns(sig(sp))$.

$$p^* \models_{op} \phi\ ,\ \ \text{if for all objects } A \in Mod(sp),\ \text{which interpret } p \text{ by } \ p^*,\ \text{it holds } A \models \phi$$

Thus, $p^* \in Mod^{op}_{sp}(p)$ iff $p^* \models_{op} \phi \to [p]\ \psi$. Carrier sets, the state mapping $STATE$ and functions interpreting attributes do not need to be considered.

- The state $STATE$ is independent from concrete signatures, since it is defined as a total mapping on the domain Id. $STATE$ is in this form contained in all Σ-objects of all signatures.
- Since the model semantics is very loose, all carrier sets can contain arbitrary non-reachable elements for sorts in S. The only requirement with respect to carrier sets is the possibility to interpret the formulas ϕ and ψ of a procedure specification $\phi \rightarrow [p]\ \psi$, i.e. the Σ-terms in the formulas must be interpretable by carrier elements. This is guaranteed for both specifications sp_1 and sp_2, since they have the same signature, i.e. the same terms can be formed.
- Properties of attributes are part of the invariants and are, therefore, part of specifications of procedures ($\phi \wedge inv \rightarrow [P]\ \psi \wedge inv$).

Instead of arguing for specifications and Σ-objects, we can now, according to the investigations above, restrict ourselves to operations. Then, we can apply the implication:

$$Mod^{op}_{sp_2}(p) \subseteq Mod^{op}_{sp_1}(p) \Rightarrow Mod(sp_2) \subseteq Mod(sp_1)$$

The model class inclusion for procedures shall now be shown.

$$Mod^{op}_{sp_2}(p) \subseteq Mod^{op}_{sp_1}(p)$$

iff

$$p^* \in Mod^{op}_{sp_2}(p) \quad \Rightarrow \quad p^* \in Mod^{op}_{sp_1}(p)$$

iff

$$p^* \models_{op} \phi_2 \rightarrow [p]\ \psi_2 \quad \Rightarrow \quad p^* \models_{op} \phi_1 \rightarrow [p]\ \psi_1$$

The hypothesis $sp_1 \overset{\mathrm{R}}{\rightsquigarrow} sp_2$ holds, i.e. $\phi_1 \rightarrow \phi_2 \wedge \psi_2 \rightarrow \psi_1$ holds for procedure p. We can use the inference rule CONS:

$$\frac{\phi_1 \rightarrow \phi_2,\ \ \phi_2 \rightarrow [p]\ \psi_2,\ \ \psi_2 \rightarrow \psi_1}{\phi_1 \rightarrow [p]\ \psi_1}$$

With $p^* \in Mod^{op}_{sp_2}(p)$, i.e. $p^* \models_{op} \phi_2 \rightarrow [p]\ \psi_2$, together with the CONS rule and the assumptions it follows $p^* \models_{op} \phi_1 \rightarrow [p]\ \psi_1$. Thus, model class inclusion is guaranteed: $Mod^{op}_{sp_2}(p) \subseteq Mod^{op}_{sp_1}(p)$. This holds for all procedures $p \in OP$. Other elements of Σ-objects are, as shown above, neutral. Thus, the model class inclusion $Mod(sp_2) \subseteq Mod(sp_1)$, i.e. $sp_1 \overset{\mathrm{I}}{\rightsquigarrow} sp_2$ holds. □

Proof Lemma 5.1 We have to show strictness and monotonicity.

1. Strictness: by definition.
2. Monotonicity: Let $sp \sqsubseteq_{Spec} sp'$. Then we get

   $$\mathcal{M}^{par}(\lambda X : SP_I .\ SP_B)(env)(sp) \sqsubseteq_{Spec} \mathcal{M}^{par}(\lambda X : SP_I .\ SP_B)(env)(sp'),$$

   due to the following two cases for specification sp:

   (a) $sp = \bot_{Spec}$: $\bot_{Spec} \sqsubseteq_{Spec} sp'$ according to definition of $\sqsubseteq_{Spec}$, thus $sp' = \bot_{Spec} \Rightarrow sp = \bot_{Spec} \Rightarrow sp \sqsubseteq_{Spec} sp'$.

(b) $sp \neq \perp_{Spec}$: $\mathcal{M}(SP_B)(env[X \mapsto sp]) \sqsubseteq_{Spec} \mathcal{M}(SP_B)(env[X \mapsto sp'])$ will be shown by induction over the structure of SP_B. Let SP_B be primitive. SP_B does not depend on X, thus we have $\mathcal{M}(SP_B) \sqsubseteq_{Spec} \mathcal{M}(SP_B)$. With $\mathcal{M}(SP_B)(env[X \mapsto sp]) \sqsubseteq_{Spec} \mathcal{M}(SP_B)(env[X \mapsto sp'])$ we get $\mathcal{M}^{par}(\lambda X : SP_I .\, SP_B)(env)(sp) \sqsubseteq_{Spec} \mathcal{M}^{par}(\lambda X : SP_I .\, SP_B)(env)(sp')$ by definition.

□

Proof Theorem 6.1 According to Definition 5.3, the component

$$compose\ \langle SP_I, SP_B\rangle\ with\ \langle SP'_I, SP'_B\rangle\ via\ \rho$$

is correct, if

$$Mod(SP'_I) \subseteq Mod((\lambda X : SP_I .\, SP_B)(SP'_B|_\rho))|_{sig(SP'_I)}$$

i.e. if

$$Mod(SP'_I) \subseteq Mod(SP_B(env(X \mapsto SP'_B|_\rho)))|_{sig(SP'_I)}$$

holds. $Mod(SP'_I) \subseteq Mod(SP'_B)|_{sig(SP'_I)}$ holds, since $\langle SP'_I, SP'_B\rangle$ was assumed as a correct component. $Mod(SP'_B) \subseteq Mod((\lambda X : SP_I .\, SP_B)(SP'_B|_\rho))|_{sig(SP'_B)}$ holds, since according to the definition of the λ-expression and the assumed correctness of composition, SP'_B is syntactically embedded into$(\lambda X : SP_I .\, SP_B)(SP'_B|_\rho)$ by the environment env, i.e. each model of SP'_B can be extended to a model of $(\lambda X : SP_I .\, SP_B)(SP'_B|_\rho)$. The requirement of signature inclusion $sig(SP'_B) \subseteq sig((\lambda X : SP_I .\, SP_B)(SP'_B|_\rho))$ allows this. Together with $sig(SP'_I) \subseteq sig(SP'_B)$ and $Mod(SP'_I) \subseteq Mod(SP'_B)|_{sig(SP'_I)}$, the required model class inclusion holds. □

Proof Theorem 6.2

a) Since the implementations from the assumptions hold, we have $sig(SP_I^1) = sig(IM_I^1)$ and $sig(SP_B^2) = sig(IM_B^2)$. Thus, ρ' is a signature morphism, if ρ is a signature morphism.

b) It shall be shown that the composition is correct. Thus,

$$IM_I^1 \overset{\perp}{\rightsquigarrow} IM_B^2|_{\rho'}$$

has to hold. By assumption, we have

$$SP_I^1 \overset{\perp}{\rightsquigarrow} SP_B^2|_\rho,\ IM_I^1 \overset{\perp}{\rightsquigarrow} SP_I^1,\ SP_B^2 \overset{\perp}{\rightsquigarrow} IM_B^2$$

Due to the transitivity of the implementation $\overset{\perp}{\rightsquigarrow}$ and the equality between the signature morphisms $\rho = \rho'$, and thus $SP_B^2|_{\rho'} \overset{\perp}{\rightsquigarrow} IM_B^2|_{\rho'}$, the hypothesis holds.

c) The implementation relation $\overset{\perp}{\rightsquigarrow}_{comp}$ holds according to Definitions 6.1 and 6.2, if

1. $sig((\lambda X : SP_I^1 .\, SP_B^1)(SP_B^2|_\rho)) = sig((\lambda X : IM_I^1 .\, IM_B^1)(IM_B^2|_{\rho'}))$ and
2. $Mod((\lambda X : IM_I^1 .\, IM_B^1)(IM_B^2|_{\rho'})) \subseteq Mod((\lambda X : SP_I^1 .\, SP_B^1)(SP_B^2|_\rho))$,
3. $sig(IM_I^2) = sig(SP_I^2)$ and
4. $Mod(SP_I^2) \subseteq Mod(IM_I^2)$.

The validity of the four statements can be shown as follows:

1. Satisfied, since
$$\langle SP_I^1, SP_B^1\rangle \overset{\mathrm{I}}{\leadsto}_{comp} \langle IM_I^1, IM_B^1\rangle$$
implies the equality of signatures
$$sig(SP_B^1) = sig(IM_B^1)$$
(it holds $sig((\lambda X : SP_I.\, SP_B)(SP'_B|_\rho)) = sig(SP_B)$ by definition).
2. The inclusion
$$Mod((\lambda X : IM_I^1.\, IM_B^1)(IM_B^2|_{\rho'})) \subseteq Mod((\lambda X : SP_I^1.\, SP_B^1)(SP_B^2|_\rho))$$
or
$$Mod(IM_B^1(env[X \mapsto IM_B^2|_{\rho'}])) \subseteq Mod(SP_B^1(env[X \mapsto SP_B^2|_\rho]))$$
holds, since due to the assumed component implementations
$$Mod(IM_B^1) \subseteq Mod(SP_B^1) \text{ and } Mod(IM_B^2) \subseteq Mod(SP_B^2)$$
hold (see Definition 5.3) and the λ-expression defines a specification function, i.e. it is monotonous with respect to $\subseteq$ (Theorem 5.1).
3. Analogous to 1.
4. By assumption.

$\square$