Observation and Abstract Behaviour in Specification and Implementation of State-based Systems

Claus Pahl
School of Computer Applications
Dublin City University
Dublin 9, Ireland
cpahl@compapp.dcu.ie

Abstract

Classical algebraic specification is an accepted framework for specification. A criticism which applies is the fact that it is functional, not based on a notion of state as most software development and implementation languages are. We formalise the idea of a state-based object or abstract machine using algebraic means. In contrast to similar approaches we consider dynamic logic instead of equational logic as the framework for specification and implementation. The advantage is a more expressive language allowing us to specify safety and liveness conditions. It also allows a clearer distinction of functional and state-based parts which require different treatment in order to achieve behavioural abstraction when necessary. We shall in particular focus on abstract behaviour and observation. A behavioural notion of satisfaction for state-elements is needed in order to abstract from irrelevant details of the state realisation.

1 Introduction

Algebraic methods have been widely used in the specification of computer systems. Algebraic specification refers to the use of algebraic semantics and equational reasoning for functional systems [1]. During the last decade, algebraic methods have been used to support the more wide-spread state-based software development. Examples of state-based specification include the pre/post-condition technique originally developed by Hoare [2, 3]. This formal technique has found its way into various software development approaches and languages such as the UML Object Constraint Language [4] or the design-by-contract approach [5]. The pre/post-condition technique can be extended or generalised in various ways [6]. We shall address the more general framework of dynamic logic here, including pre- and postconditions. We shall in particular address semantic foundations for a framework for observational specification of state-based systems in this paper. This shall result in a more flexible specification and implementation framework for state-based systems.

A classical example of the usefulness of obervational specification is the realisation of a data type boolean using an implementation of a type integer. The data values needed to interpret terms formulated based on boolean constructors (e.g. true and not) form only a subset of a carrier set for integers. We expect boolean axioms only to hold within this subset. An observability predicate can declare those substructures reachable with boolean constructors as observable. Observation and behaviour oriented extensions to the algebraic specification of data types exist (e.g. [7, 8, 9, 10, 11, 12, 13, 14]), but similar approaches to state-based specification are still lacking. We aim to provide semantical foundations for such a framework of behaviour and observation in the context of state-based systems.

We shall give an outline of our approach here. The classical notion of signatures is extended by introducing hidden, non-observable sorts representing an internal state. Semantic structures interpreting these signatures provide suitable elements. Slightly different techniques apply for defining and reasoning about functional and state-based parts. One

of the most well-known approaches to these problems is the hidden algebra framework developed by the OBJ-group, see e.g. [15, 16]. There, an equational, behaviourally oriented framework for reasoning is established. We follow the hidden algebra approach in its basic idea: we provide signatures with a distinguished state sort, and interpret these in structures with a local state, called objects. An early version of this model has been presented in [17]. Then, we deviate from the path by using concepts from modal logics to introduce a specification and reasoning framework. A dynamic logic will be used. Modal logics, such as dynamic and temporal logics, provide constructs that will allow us to specify safety as well as liveness constraints – a wider range of properties than with equational logics. Safety properties shall guarantee that unwanted things will not occur, and liveness properties should ensure that wanted things will eventually occur. These properties are in particular important in the specification of dynamic and reactive systems. Our particular interest lies in the implementation of specifications. In implementing a specification it has to be shown that the implementation satisfies the abstract requirements of the specifications. In order to allow the simpler equational reasoning to be used, we complement a dynamic logic based implementation with a refinement notion which needs equational reasoning only in proofs. We consider data and operation refinement and show how they relate to the paradigm of observational or behavioural specification, namely, that formulas need only to be true under all possible experiments (or observations).

We start with our semantic model in Section 2. Then, in Section 3, we present the dynamic logic. Section 4 introduces basic concepts for behavioural specifications. In Section 5, we look at data refinement based on behavioural abstraction. In Section 6, the operation refinement is investigated. Proofs, if they are not given here, can be found in [18].

2 A State-based Model

The main building blocks of our computational model – signatures with state, objects as the corresponding models, and an interpretation – shall be formalised in this section. The objective is to formalise the notion of an object - an algebraic entity that encapsulates a local state.

We assume a set of sorts $Sort$ including data sorts $s_1, s_2, \ldots$ and the distinguished sort $state$, and a set of identifiers Id. For each **state signature** Σ in a class of signatures Sig we define:

- a mapping $sorts : Sig \to \mathcal{P}Sort$ with $state \in sorts(\Sigma)$,
- a mapping $attr : Sig \to \mathcal{P}Id$ with a signature $sig(a) = s_1 \times \ldots \times s_n \to s$ for each attribute $a \in attr(\Sigma)$ with $s_1, \ldots, s_n, s \in (sorts(\Sigma) - \{state\})$[1]
- the state sort $state$ as $attr(\Sigma) \to sig(attr(\Sigma))$
- a mapping $tran : Sig \to \mathcal{P}Id$ with a signature $sig(p) = state \times s_1 \times \ldots \times s_m \to state \times s$ for each transition $p \in tran(\Sigma)$.

Basic sorts – not including $state$ – are typically data types used in the definition of attributes. We assume a built-in sort $bool$ with the usual constants and functions. The set of attributes, variables (nullary functions) and other functions, forms an explicit, but still abstract state. The state itself is represented as an assignment, associating attribute identifiers with functions. These functions are the only means to inspect the state content. Transitions are parameterised state transformers, transforming one state into another, also yielding a functional result value.

Example *2.1 Sample sorts in a stack data type are the stack content $stack$ – which shall represent the state-sort – and the data type $elem$. The following Σ is a signature:*

$$\begin{array}{lcl} sorts(\Sigma) & = & \{state, elem, bool\} \\ attr(\Sigma) & = & \{top :\to elem,\ is\_full :\to bool\} \\ tran(\Sigma) & = & \{push : state \times elem \to state,\ pop : state \to state\} \end{array}$$

[1] a can be a nullary symbol, then called a state variable.

An example for an attribute is $sig(top) = \rightarrow elem$. Examples for transitions are $sig(push) = stack \times elem \rightarrow stack$ or $sig(pop) = stack \rightarrow stack$.

Let Σ, Σ' be state signatures. Σ' is a **subsignature** of Σ, or $\Sigma' \subseteq \Sigma$, if $sorts(\Sigma') \subseteq sorts(\Sigma)$, $attr(\Sigma') \subseteq attr(\Sigma)$ and $tran(\Sigma') \subseteq tran(\Sigma)$.

A **hidden signature** [15] 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 to the signature. Components and transitions depend on the state. Clearly, our state signatures are hidden signatures in the sense of [16].

A **state signature morphism** σ is a signature morphism where $\sigma_{Sort}(state)$ is the identity, and for any operation $op \in attr(\Sigma) \cup tran(\Sigma)$ with $sig(op) = a_1 \times \ldots \times a_i \rightarrow b_1 \times \ldots \times b_j$ we require $sig(\sigma_{Op}(op)) = \sigma_{Sort}(a_1) \times \ldots \times \sigma_{Sort}(a_i) \rightarrow \sigma_{Sort}(b_1) \times \ldots \times \sigma_{Sort}(b_j)$ for appropriate i, j, where σ_{Sort} is a mapping on sorts and σ_{Op} a mapping on attributes and transition identifiers. A subsignature Σ' can be embedded into a signature Σ by an embedding signature morphism.

Example *2.2 Consider the signature Σ':*

$$\begin{array}{lcl} sorts(\Sigma) & = & \{state, elem\} \\ attr(\Sigma) & = & \{top :\rightarrow elem\} \\ tran(\Sigma) & = & \{push : state \times elem \rightarrow state,\ pop : state \rightarrow state\} \end{array}$$

Σ' is a subsignature of Σ – see Example 2.1. The embedding of Σ' into Σ is a signature morphism.

A signature induces a set of syntactically correct expressions, constructed by free variables and the operation names of the signature. In many-sorted signatures, a sort will be associated with each term. Let $X = (X_s)_{s \in sorts(\Sigma)}$ be a $sorts(\Sigma)$-sorted set of free variables. The functions π_1 and π_2 are projections onto the first and second component of their argument, respectively.

- **Σ-terms of data sort** $s \in sorts(\Sigma)$ over X_s are variables $x \in X_s$; every $a(a_1, \ldots, a_n)$ for $a \in attr(\Sigma)$ with $a : s_1 \times \ldots \times s_n \rightarrow s$ and $a_i : s_i$; every projection $\pi_2(p(st, a_1, \ldots, a_m))$ for $p \in tran(\Sigma)$ with $p : state \times s_1 \times \ldots \times s_m \rightarrow state \times s$, $st : state$ and $a_i : s_i$.
- **Σ-terms of state sort** $state \in sorts(\Sigma)$ over X_s are identifiers st to denote the state; and every projection $\pi_1(p(st, a_1, \ldots, a_m))$ for $p \in tran(\Sigma)$ with $p : state \times s_1 \times \ldots \times s_m \rightarrow state \times s$, $st : state$ and $a_i : s_i$.

Example *2.3 A Σ-term of data sort $bool$ is $is\_full()$. A Σ-terms of state sort $state$ are $pop(st)$ or $push(st, e)$.*

With $T(\Sigma)_s$ we denote the set of ground Σ-terms of sort s. With $T(\Sigma, X)_s$ we denote the set of Σ-terms of sort s including variables. A term of sort $state$ is also called a **command**.

We can now capture the idea of a state-based computational model precisely. An algebraic structure is called a **Σ-object** for a state signature Σ, if it has

- a carrier set S, including an undefinedness symbol $\perp$, for each sort s.
- a function of type $S_1 \times \ldots \times S_n \rightarrow S$ for each attribute with signature $s_1 \times \ldots \times s_n \rightarrow s$.
- a carrier set $State$ for sort $state$ containing total assignments $Id \rightarrow F$ if F is the set of functions that match signatures of attributes.
- a designated **initial state** $ST_{init} \in State$.

- a function of type $(State \times S_1 \times \ldots \times S_m) \rightarrow (State \times S)$ for each transition symbol in $tran(\Sigma)$ with the corresponding signature.

Σ-objects can be seen as abstract machines with operations as instructions. Initial states (and the ability to specify properties of the initial state) are essential in the context of dynamic or reactive systems. Underlying each of these models, we assume an equationally specified algebra, realising the data type(s) used in the definition of the attributes. These algebras are static, i.e. their functions can not be changed by state transitions. Σ-objects are extensions of these algebras. The state can also be seen as an algebra consisting of carrier sets for data sorts and functions for the attributes. The **Σ^*-state algebra** for a Σ-object shall be defined as the algebra consisting of carrier sets for the data sorts (excluding $state$) and functions for the attributes $attr(\Sigma)$. The signature Σ^* consists of the data sorts and attribute signatures of Σ. This shows that Σ-objects fit into the states-as-algebras paradigm.

A **hidden algebra** is an algebra which satisfies a hidden specification based on a hidden signature [15]. A hidden algebra should encapsulate an algebra as a substructure (reduct) which represents the data part. Our objects are hidden algebras in this sense.

A Σ-object involves bindings: functions are bound to attribute and transition identifiers. A state ST of sort $state$ can be modified by transitions. 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 val:

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

Domain and range of ST shall be disjoint. Then, x cannot occur in val.

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, i.e., an assignment[2] of functions to identifiers.

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

The value of an identifier x is stored in ST – remember the definition of sort state: $state = attr(\Sigma) \rightarrow sig(attr(\Sigma))$, i.e., that the state associates attributes and functions.

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

The definition of an attribute a can be modified, thus we have to reinterpret a in each current state ST. Transitions p might modify the definition of attributes. The brackets $[\![.]\!]$ are used to denote the function which realises a transition symbol.

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

Example *2.4 A Σ-object A may contain an attribute* top. state *is the local state variable and* top *is an attribute based on the content of* state. *States can be transformed into another by* push *and* pop. push *is a transition and, therefore, changes the state. By executing* push(st,e) *the state is changed by substituting the binding of* top *through assigning a new value ($v^*(ST, e)$) to the attribute identifier* top. *The effect of* push *can be observed by applying* top.

[2] We use the term *assignment* for the state and *valuation* for identifiers of the signature in general.

3 State Transition Logic

Modal logics assume structures with a notion of state or time [19, 20]. The presented model of Σ-objects is such a structure. Dynamic logic, a particular modal logic, is a first-order predicate logic with a notion of state, which generalises Hoare-logic. Based on these logics, we will introduce a simple dynamic logic. A proof system is not presented here, see [18] for a sound and relatively complete one for a general dynamic logic and [21] for one geared towards the use in refinement calculi.

The state transition formula $\phi \rightarrow [P]\psi$ – where ϕ is a precondition, P is a command (a term of sort $state$) and ψ is a postcondition – describes a state transition axiomatically. The operator $[.]$ is called the modal **box**- or **always**-operator. ϕ and ψ describe properties of attributes by classical first-order formulas. The formula $\phi \rightarrow [P]\psi$ holds in a current state ST in a Σ-object A, if in case ϕ holds in ST and, if P terminates, then ψ has to hold in the following state. The second modal operator is the so-called **diamond**- or **eventually**-operator. This operator allows us to specify liveness properties. The formula $< P > \ \phi$ expresses that P has a terminating path after which ϕ holds. In case of determinsitic programs, it says that P will terminate and ϕ holds afterwards. For example, $< P > \ true$ is a termination assertion. These constructs will be defined formally later on.

Example *3.1 The effect of the stack transition* push *can be specified by*

$$\neg\text{is\_full}() \rightarrow [\text{push(st,e)}]\ \text{top()=e}$$

If the precondition – the stack is not full – is satisfied, then executing push(st,e) *on stack* st *has the effect that* e *is the new top element.*

$$\text{is\_full}() \rightarrow \ <\text{push(st,e)}>\ \text{true}$$

$push$ should terminate if the stack is full (whatever the result might be).

Let Σ be a state signature and X a $sorts(\Sigma)$-sorted set of free variables. 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[3]:

- all Σ-equations and Σ_{state}-equations are in WFF(Σ),
- if $\phi, \psi \in$ WFF(Σ), then $\phi \rightarrow \psi \in$ WFF(Σ),
- if P is a command and $\phi \in$ WFF(Σ), then $[P]\ \phi \in$ WFF(Σ) and $< P > \ \phi \in$ WFF(Σ).

We define the diamond operator as follows:

$$< P > \ \phi := \neg [P]\, \neg \phi$$

An operator $prev$ shall also be defined. The operator refers to the value of an argument variable in the previous state of the command P under consideration. Typically, the $prev$ operator is used to specify the new value of a variable in terms of the old value, e.g. $[\ P\ ]\ a() = prev(a()) + 1$.

A **specification** M is a pair $M = \langle \Sigma, E \rangle$ consisting of a state-based signature Σ and a set E of well-formed formulas $E \subseteq$ WFF(Σ), including an non-modal formula $Init$. $Init$ is a non-modal formula characterising the initial state of a Σ-object.

[3] We have omitted classical connectors, such as negation or disjunction, and quantification. They can be defined as usual for first-order predicate logics.

Example *3.2 A specification $M = \langle \Sigma, E \rangle$ for a stack consists of:*

$$
\begin{array}{ll}
\Sigma: & \\
 & sorts(\Sigma) = \{state, elem, bool\} \\
 & attr(\Sigma) = \{top :\rightarrow elem,\ is\_full :\rightarrow bool\} \\
 & tran(\Sigma) = \{push : state \times elem \rightarrow state,\ pop : state \rightarrow state\} \\
E: & \\
 & \neg is\_full() \rightarrow [push(st, e)]\ top() = e \\
 & top() = e \rightarrow [push(st, e'); pop(st)]\ top() = e \\
 & is\_full() \rightarrow\ <push(st, e)>\ true
\end{array}
$$

The state is the content of the stack, $elem$ is the element sort. The element e shall be of sort $elem$ and st of sort $state$. This example assumes that a sequence operator ; is defined on transitions. This is not a complete specification of a stack as it, for example, does not specify the effect of pop on empty stacks.

As we have seen in the example, command combinators such as the sequence ';' are useful. We define the combinators ; (sequence), + (non-deterministic choice), and $\_^*$ (iteration):

- $v^*(ST, p; q) := v^*(\pi_1(v^*(ST, p)), q)$
- $v^*(ST, p + q) := v^*(ST, p)$ or $v^*(ST, q)$ – non-deterministically chosen
- $v^*(ST, p^*) := v^*(\pi_1(\ldots(\pi_1(v^*((ST, p)), p), \ldots, p)$

For two of them, we can find simple axiomatisations:

- $[p; q]\phi \Leftrightarrow [p][q]\phi$
- $[p + q]\phi \Leftrightarrow [p]\phi \vee [q]\phi$

A notion of satisfaction relates formulas and Σ-objects. Let v be a valuation based on states and v^* an interpretation. 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 $v^*(\pi_1(v^*(ST, t)), a(a_1, \ldots, a_{n_c}))) = v^*(\pi_1(v^*(ST, t')), a(a_1, \ldots, a_{n_c}))$ for all attribute applications
- $A, ST \models \phi \rightarrow \psi$ iff $(\neg\phi) \vee \psi$ [4]
- $A, ST \models [P]\ \phi$ iff $D(P) \rightarrow A, \pi_1(v^*(ST, P)) \models \phi$ where D is the definedness predicate

Nontermination is denoted by $\perp_{state}$. Note, that terms of sort $state$ are **behaviourally** equal. Two terms are only equal if the interpretations of all attribute applications in the subsequent state are equal. This definition was chosen instead of comparing two state structures directly which would have involved non-relevant comparisons.

Example *3.3 The transition* push *on stack has been specified by*

$$\neg \text{is\_full()} \rightarrow \text{[push(st,e)] top()=e}$$

Only the observable behaviour of push *is specified, i.e. if* push *is executed in a state satisfying* ¬is_full() *and it terminates, then it should terminate in a state whose properties can be inspected by attributes such as* top.

[4] with the usual definitions for $\neg$ and $\vee$

The class of all Σ-objects is denoted with $Obj(\Sigma)$. The **models** of a specification $M = \langle \Sigma, E \rangle$ are denoted by their model class.

$$mod(\langle \Sigma, E \rangle) := \{A \in Obj(\Sigma) \mid A, ST \models \phi \text{ for all } \phi \in E \text{ and } ST \in State\}$$

Note that E contains an initial state condition, i.e. models are only those objects whose designated initial state satisfies that condition. Model construction not requiring closedness with respect to isomorphism is usually called *loose*. Here, even reachability – the no-junk property – is not required. We call this model semantics *very loose*.

We will also introduce notions of a model of a transition and a respective satisfaction relation. Let a transition p in M_1 be specified by $\phi_1 \rightarrow [p]\ \psi_1$ and in M_2 by $\phi_2 \rightarrow [p]\ \psi_2$. Then $mod^{tr}_M(p)$ denotes **models of a transition** p, i.e. those functions which interpret a transition p in a model of a specification M. Let $A \in mod(M)$, $[\![p]\!] \in mod^{tr}_M(p)$ and $p \in tran(sig(M))$. Then the **satisfaction** $\models_{tr}$ is defined as follows: $[\![p]\!] \models_{tr} \phi$ holds, if $A \models \phi$ holds for all objects $A \in mod(M)$ which interpret p by $[\![p]\!]$.

Lemma 3.1 *With the previous two definitions, we get the following property for two Σ-formulas ϕ, ψ specifying a transition p by $\phi \rightarrow [p]\ \psi$:*

$$[\![p]\!] \in mod^{tr}_M(p) \text{ iff } [\![p]\!] \models_{tr} \phi \rightarrow [p]\ \psi$$

Proof: Obvious. □

4 Observational Specification

Let us look at notions of *behavioural* and *observational* specification now. We briefly present some basic constructs used in the majority of observational algebraic specification, see e.g. [11, 13, 14, 16].

Definition 4.1 *Let Σ be any signature. A Σ-**context** is a Σ-term, which contains exactly one special variable z_s for each sort s. The variables z_s are used as meta-variables for terms of a specific sort.*

$$c[t] := c[z_s/t] \text{ for } t : s$$

The variable z_s can be substituted by t in the term c.

This shall be illustrated by a stack example.

Example *4.1 $top(z)$ shall be a context with special variable z. Then, the equation $pop(push(s, e)) = s$ does not need to hold, only the application of the observational context to the equation:*

$$top(pop(push(s, e))) = top(s)$$

$pop(push(s, e))$ and s substitute z.

In our approach, we would model $push$ and pop as procedures, since they modify the state, here the stack. top would be an attribute.

$$top() = e \rightarrow [\ seq(push(e'), pop())\ ]\ top() = e$$

This guarantees that only observations on the state using *top* are relevant. This is already behaviourally oriented. We have also defined a behavioural satisfaction for equations on $state$.

The standard equality can be generalised in observational approaches: two terms shall be considered equal, if they cannot be distinguished through observations. An observational predicate Obs can be applied to any terms that denote carriers. There is an observational subset for each carrier set. An **observational Σ-context** is a Σ-context with observational sort.

Example *4.2 For instance $top(z)$ would be an observational Σ-context, if the sort of $top[t]$ for a suitable t is a observational sort.*

The notion of a context can also be defined for the state-based setting. A term $c \in T(\Sigma, X)_{s \in S \cup \{state\}}$ is called **context over** Σ, if c contains exactly one $z_s \in Z = \{z_s | s \in S \cup \{state\}\}$. A context over z_s shall be called **observational**, if s is not the sort $state$.

An observational subalgebra (A, Obs^A) is the subalgebra of A that contains only those elements denotable by ground terms defined as observable. The **observational satisfaction** $\models_{OBS}$ is defined by:

$$(A, Obs^A) \models_{OBS} t = t' \text{ if } A \models c[t] = c[t']$$

for all observational contexts c. Hennicker [8, 10, 11] defines an *observational predicate* $Obs(t)$ for term $t \in T(\Sigma, X)$ which **holds** in a Σ-algebra (A, Obs^A) — or $(A, Obs^A) \models_{OBS} Obs(t)$ — if for all valuations $v : X \to A$ the interpretation of t by v is element of the observational subalgebra (A, Obs^A). i.e. $v^*(t) \in Obs^A$. Models are consequently defined by:

$$Mod_{OBS}(M) := \{A \in Alg_{OBS}(\Sigma) \mid A \models_{OBS} \phi \text{ for all } \phi \in E\}$$

We will present a variation of this observation predicate and observable substructures in Section 5. An **implementation relation** $\xrightarrow{I}$ can be defined as usually through model class inclusion, now based on observational model class construction:

$$M \xrightarrow{I} M' \text{ iff } Mod_{OBS}(M') \subseteq Mod_{OBS}(M)$$

for equal signatures of M and M'.

Example *4.3 Define a specification $N = \langle \Sigma, F \rangle$ with*

$$\begin{array}{ll} \Sigma: & \\ & sorts(\Sigma) = \{state, elem, bool\} \\ & attr(\Sigma) = \{top :\to elem,\ is\_full :\to bool\} \\ & tran(\Sigma) = \{push : state \times elem \to state,\ pop : state \to state\} \\ F: & \\ & [push(st, e)]\ top() = e \\ & [push(st, e'); pop(st)]\ top() = etrue \end{array}$$

Then the specification M – as in Example 3.2 – is an implementation of N, i.e. $N \xrightarrow{I} M$. The specification M is more specific since it defines the behaviour for full stacks.

We will consider implementations later on in Section 6. In order to prove correct implementations in behavioural settings, for example context induction can be applied. The induction is based on the application of observational contexts to ground terms, see work by Hennicker and Bidoit [10, 14] and also work on co-induction, e.g. [16].

5 Data Refinement

Our notion of satisfaction is already defined on the idea of observational behaviour (cf. Section 3). Internal properties of the state are invisible. They can only be inspected by attributes. The notion of observation is in particular important if implementation is taken into consideration. In implementation, concepts are realised whose properties are not relevant with respect to some abstract specification. Still, implementations shall be considered correct if they satisfy some abstract observational behaviour.

Considering the process of implementing and refining specifications and the necessity of proving the correctness of those, we find that we need means to abstract from implementation details in order to prove that implementations show

the same observational behaviour as the specifications. We have used a very loose semantics for specifications, i.e. not all carrier elements need to be representable by terms, i.e. are reachable. We have ignored reachability in order to obtain a widely defined notion of models. Non-reachable elements will be accepted, since they do not influence the behaviour of models. This allows the implementation of specifications based on libraries of existing standard implementations (for example basic data types), which might not match specifications exactly. Due to this requirement, the very loose semantics approach is sensible.

Example *5.1 Standard models of the data type $Integer$ can be used to implement a specification of boolean values. By applying an observational predicate Obs*

$$Obs_{\{true,false,not\}_{int}}(i)$$

we want to express that only the carrier elements of a carrier set $Integer$ for sort int reachable by $true$, $false$ and not are observable. These could for example be the numbers 0 and 1. All other numbers are unreachable (with boolean constructors), and therefore not observable.

The principle problem with models including non-reachable elements – that structural induction based on the syntactic structure is not possible – can be countered by using the Obs predicate.

Example *5.2 With the formula*

$$Obs_{\{true,false,not\}_{int}}(t) \rightarrow (not(not(t)) = t)$$

we specify that $not(not(t)) = t$ shall only hold (and needs to be proved) in a substructure reachable with $true$, $false$ and not.

We can verify in the substructures defined through the Obs-predicate. Another use of the Obs-predicate shall be introduced. Obs can be used to define constructors.

Example *5.3 With $Obs_{int}(zero)$ and $Obs_{int}(n) \rightarrow Obs_{int}(succ(n))$ we define the constructors $zero$ and $succ$ for natural numbers.*

The term 'observational' indicates that more than reachability is aimed at. Observability allows to relax a correctness condition for implementations. The predicate $Obs_{\Sigma'}$ can be defined for every subsignature Σ' of Σ. $Obs_{\Sigma'}(t)$ shall express that a carrier element $v^*(ST, t)$ can be denoted by a Σ'-ground term:

$$Obs_{\Sigma'}(t) \text{ iff } \exists t' \in T(\Sigma') \,.\, v^*(ST, t) = v^*(ST, t') \text{ for } t : s \text{ and any state } ST$$

Since we are talking about reachability or observability in the context of sorts or specific operation symbols, we offer three forms to describe the signature Σ' of the predicate $Obs_{\Sigma'}$:

- Σ' : any subsignature Σ' of a given signature Σ,
- Σ_s : a sort s indicating that only elements of this sort are relevant,
- $\Sigma_{\{op_1,\ldots,op_n\}_s}$: a list of selected operation symbols of a given sort s.

We have seen an example of the last form – which is the most general form – in Examples 5.1 and 5.2. An example for the second form can be found in Example 5.3.

We introduce two abbreviations. Firstly, a **relativised universal quantification** $\forall x : s.\ Obs_\Sigma(x) \rightarrow \phi$ or short $\forall x : \Sigma.\ \phi$: for all elements ϕ holds or they are not reachable, i.e., ϕ hold for all reachable elements. Secondly, a **relativised existential quantification** $\exists x : s.\ Obs_\Sigma(x) \wedge \phi$ or short $\exists x : \Sigma.\ \phi$: there are observable element for which ϕ holds. With $Obs_{\Sigma'}$ and $\Sigma = \Sigma'$ we model reachability.

Example *5.4 Consider Example 5.2. Instead of*

$$\forall t : s \,.\, Obs_{\{true,false,not\}_{int}}(t) \rightarrow (not(not(t)) = t)$$

we would write

$$\forall t : \{true, false, not\}_{int} \,.\, not(not(t)) = t$$

In other approaches, e.g. Hennicker's work, contexts are used to defined observable constructs. Here, we use subsignatures.

With A^{Obs} we shall denote the **observable Σ'-subobject** for a Σ-object A:

$$a \in Obs^A_{\Sigma'_s} \text{ iff } \exists t \in T(\Sigma')_s \text{ with } a = v^*(ST, t) \text{ for } s \neq state \text{ and any state } ST$$

This is a standard definition based on elements reachable via ground terms of the subsignature. We construct the observable subobject as follows:

- $A^{Obs}_s := Obs^A_{\Sigma'_s}$ for data sorts $s \in (sorts(\Sigma) - \{state\})$,
- $A^{Obs}_{state} := Obs^A_{\Sigma'_{state}}$ for sort $state$,
- $op^{Obs}_A := op^A_{|Obs}$[5] for any operation $op \in attr(\Sigma) \cup tran(\Sigma)$,
- $ST^{Obs} := ST$ for $ST \in A^{Obs}_{state}$.

Lemma 5.1 *The observable Σ'-subobject of A is a Σ'-object.*

Proof: Follows from the definition of a Σ-object and observable Σ-subobjects. □

Satisfaction with respect to observable substructures can be expressed by:

$$A^{Obs}, ST \models Obs_{\Sigma'}(t) \text{ iff } D(ST, t) \text{ for } t \in T(\Sigma')$$

Lemma 5.2 *Let A be model of a specification $< \Sigma, E >$ and let t be a Σ-term of sort s. Then:*

$$A, ST \models Obs_{\Sigma'}(t) \textit{ iff } \Sigma' \subseteq \Sigma \; and \; v^*(ST, t) \in Obs^A_{\Sigma'_s}$$

Proof: Follows from the definitions of the satisfaction of Obs and observable subobjects. □

In order to illustrate how to implement a data type, we use the data type *set*, which shall be implemented based on an existing model of tuples. The implementation shall be formalised through model class inclusion, i.e., M' implements M, or $M \stackrel{1}{\rightarrow} M'$, if $Mod(M') \subseteq Mod(M)$. The resulting set specification will still be an abstract specification. We will implement sets of natural numbers on tuples of integers. In the set of integers, only natural numbers shall be observable. It shall be assumed that *SET* and *TUPLE* are generic specifications which can be instantiated with element types such as *NAT* or *INT*.

Example *5.5*

[5] $op^A_{|Obs}$ is op^A restricted to the observable subsets of carrier sets.

spec SET(NAT)_by_TUPLE(INT) is
extend TUPLE(INT) by
opns
set_insert: tuple $\times$ *int* $\rightarrow$ *tuple*
set_delete: tuple $\rightarrow$ *tuple*
empty_set: $\rightarrow$ *tuple*
axioms (t : tuple; i : int)
$Obs_{\{empty\_set, set\_insert\}_{tuple}}(t)$ $\rightarrow$ *'set axioms'*
$Obs_{\{zero, succ\}_{int}}(i)$ $\rightarrow$ *'Peano axioms'*
end spec

The operations set_insert and $empty\_set$ shall be the constructors, i.e. they denote the same reachable carriers as the whole set of operations of the signature. Only those set elements are observable which can be reached via ground terms based on $zero$ and $succ$. These are the natural numbers. The axioms have to hold only for them. This is of importance if we implement on standard models of predefined types such as $Integer$. $Obs_{\{zero, succ\}_{int}}(i)$ holds, if $v^*(ST, i)$ is element of the subset reachable by $zero$ and $succ$ for any state ST. Thus, *SET(NAT)_by_TUPLE(INT)* is a correct implementation of a specification *SET(NAT)* with operations *set_insert, set_delete, empty_set* and the usual set axioms and set elements which are natural numbers that obey the Peano axioms. These are still abstract specifications, i.e., proof obligations if an executable version is derived.

The example illustrates the main application of the observability predicate: to realise a specification using another, existing one. Examples are the realisation of boolean values on a data type integer or sets on a data type list/tuple. This allows a developer to realise a specification based on suitable, existing library components. Typically, the boolean or set terms (generated by their constructors) are only interpreted in subsets of the integer or list carriers, respectively. With the Obs predicate, we can declare these subsets as observable and reason within these substructures. The technique can also be used to define export interface for existing specifications (e.g., restricting the set of constructors).

The definition of the Obs predicate has been illustrated using familiar data sort specification and implementation so far; the sort $state$ has been excluded. The predicate Obs_{state} for sort $state$ works in the same way as Obs_s for data sorts s. Applied to states it would express the reachability or observability of states, i.e., whether there is a path from an initial state to the state under consideration.

The technique of observation-based specification can also be applied to the sort $state$. Instead of applying the Obs predicate to constructors of data types such as boolean or set - as in the previous example - we can also use constructors of sort $state$. Applied to the stack object that has been specified in previous examples, these constructors would be a constant empty stack and the push and pop operations. We could realise a stack on a list object. There might be a executable implementation for a list object specification in some component library. We assume a list object specification with the usual operations (head, tail, a list constructor, concatenation) for this example.

Example *5.6*

spec STACK_by_LIST is
extend LIST by
opns
empty_stack: $\rightarrow$ *list*
push: list $\times$ *elem* $\rightarrow$ *list*
pop: list $\rightarrow$ *list*
axioms (l : list)
$Obs_{\{empty\_stack, push, pop\}_{list}}(l)$ $\rightarrow$ *'stack axioms'*
end spec

The stack axioms are defined in previous examples. Again, the specification in the example formulates the proof obligations as axioms.

Example *5.7 More concrete specifications could contain the following axioms:*

empty_stack() = empty_list()
top(st) = head(st)
push(st,e) = concat(list(e),st)

This describes the implementation of stacks in terms of lists more explicitly.

6 Operation Refinement

An **implementation** between two specifications M and M', $M \xrightarrow{\mathrm{I}} M'$ shall be defined based on model class inclusion, i.e. we require $Mod(M') \subseteq Mod(M)$. Model class inclusion formally captures the idea of making design decisions: there are less models if requirements are added or strengthened[6]. Model class inclusion guarantees that properties established for a specification are preserved in an implementation. We have not used the Mod_{OBS}-construction introduced in Section 4 here, instead we use the variant from Section 3. As we have already pointed out, this incorporates the concept of observation. How to develop a more substantial refinement approach is described in [21]. Technical details of this section including proofs can be found in [18].

An inference rule shall be introduced. The consequence rule helps to prove pre/postcondition specifications:

$$\frac{\phi \to \phi',\ \ \phi \to [P]\ \psi,\ \ \psi' \to \psi}{\phi' \to [P]\ \psi'}$$

This rule will be useful in the definition of a constructive variant of an implementation relation, called refinement, between specifications.

Example *6.1 Consider the following formula:*

$$\neg is\_full() \to [push(st, e)]\ top() = e$$

Then the formula

$$true \to [push(st, e)]\ top() = e\ \wedge\ \neg is\_empty()$$

is a consequence of the first formula, since $top() = e\ \wedge\ \neg is\_empty() \to top() = e$ *and* $\neg is\_full() \to true$*. The postcondition is strengthened and the precondition is weakened.*

We define a few other relations between specifications. Let p_1, p_2 be transitions with the same signature and $\phi_{p_1} \to [p_1]\ \psi_{p_1}$ and $\phi_{p_2} \to [p_2]\ \psi_{p_2}$ be their respective specifications. The **implementation** $p_1 \xrightarrow[\mathrm{tr}]{\mathrm{I}} p_2$ holds, if every function interpreting p_2 in a Σ-model which satisfies $\phi_{p_2} \to [p_2]\ \psi_{p_2}$ is also interpretation of p_1 in a Σ-model which satisfies $\phi_{p_1} \to [p_1]\ \psi_{p_1}$. The **implementation** $M_1 \xrightarrow{\mathrm{I}} M_2$ holds iff $Mod(M_2) \subseteq Mod(M_1)$ for equal signatures of M and M'. The **refinement** relation $p_1 \xrightarrow[\mathrm{tr}]{\mathrm{R}} p_2$ holds, if $D(p_1) \wedge D(p_2)$ and $\phi_{p_1} \to \phi_{p_2}\ \wedge\ \psi_{p_2} \to \psi_{p_1}$ hold. The consequence rule and Example 6.1 illustrate this. We will assume terminating operations for this definition such that by execution of the operations the postcondition can always be established. Let M_1 and M_2 be specifications. The **refinement** relation $M_1 \xrightarrow{\mathrm{R}} M_2$ holds, if for all $p \in tran(\Sigma_{M_1})$ the relation $p_{M_1} \xrightarrow[\mathrm{tr}]{\mathrm{R}} p_{M_2}$ holds.

Lemma 6.1 *The relations* $\xrightarrow[\mathrm{tr}]{\mathrm{I}}$, $\xrightarrow{\mathrm{I}}$, $\xrightarrow[\mathrm{tr}]{\mathrm{R}}$ *and* $\xrightarrow{\mathrm{R}}$ *form a partial ordering.*

This is usually called the vertical composition property - a useful property for implementation relations.

In order to relate the two relations $\xrightarrow{\mathrm{I}}$ and $\xrightarrow{\mathrm{R}}$, we assume the three simplifications.

[6]Certainly, it should be allowed to add new elements in the implementation which do not influence the semantics of the original specification. This can be expressed by allowing the signature of M to be a subsignature of M. In order to simplify the approach here, we have ignored this.

- There are no explicit invariants inv[7]. They will be associated to transition definitions: reformulate $\phi \rightarrow [P]\ \psi$ to $\phi \rightarrow [P]\ \psi \wedge inv$.
- There are only transition specifications, i.e. box- and diamond operators and equations on sort $state$. Attributes are specified by normal first-order formulas and can be treated as invariants.
- Every transition is defined by only one formula – as justified in Lemma 6.2 below.

The following lemma needs the expressiveness of the base logic as a prerequisite – this shall be assumed here. In [18] Section 3.4, we have addressed the question of expressiveness in this context. A similar lemma can be derived for the diamond-operator.

Lemma 6.2 *Any set of formulas $\{\phi_i \rightarrow [P]\ \psi_i \mid i = 1, \ldots, n\}$ can be transformed into a formula $\phi \rightarrow [P]\ \psi$, which holds iff all formulas $\phi_i \rightarrow [P]\ \psi_i$ hold.*

There is a related requirement for the equational specification of hidden algebras, see [15], saying that equations on data sorts are not allowed in that framework. Any such equation has to be asserted and proved separately.

We have already seen that the refinement is compositional, i.e., the refinement between specifications depends on the refinement between the constituent transitions – by definition. The following lemma shows that the implementation is compositional in the same way. Carrier sets, the state ST, and functions interpreting state observers do not need to be considered.

Lemma 6.3 *Let M_1, M_2 be specifications with $sig(M_1) \subseteq sig(M_2)$ under the assumptions specified above. If for all transitions $t \in tran(sig(M_1))$ the inclusion $mod^{tr}_{M_2}(t) \subseteq mod^{tr}_{M_1}(t)$ holds, then $mod(M_2|_{sig(M_1)}) \subseteq mod(M_1)$, or $M_1 \xrightarrow{\text{I}} M_2$.*

Now, we state the essential theorem of this section, i.e., that the refinement is a specialisation of the implementation.

Theorem 6.1 *Let M_1, M_2 be specifications under the assumptions specified above. Then it holds:*

$$M_1 \xrightarrow{\text{R}} M_2 \ \Rightarrow\ M_1 \xrightarrow{\text{I}} M_2$$

The proof relies essentially on the compositionality of the underlying relations on transitions.

This theorem allows us to prove an implementation using the refinement. The refinement is a good approximation to the implementation since the only cases excluded are

- models which contain functions which do not terminate,
- formulas (describing states) that cannot be satisfied, and
- preconditions that describe states unreachable by transitions.

The refinement is only based on two non-modal first-order implications. Standard equational reasoning can be applied here.

[7] Underlying data type and attribute specifications are considered as invariant.

7 Related Work

Our work is clearly motivated by observational and behavioural extensions to the algebraic specification approach, in particular by work from Bidoit, Broy, Hennicker, and Wirsing [7, 8, 9, 10, 11, 12, 13, 14]. Wirsing and Broy introduce ultra loose semantics, allowing behavioural abstractions by a notion of relativised quantification. This compares to the introduction of relativised quantifications $\forall x : \Sigma.\ \phi$ and $\exists x : \Sigma.\ \phi$ here in Section 5. Bidoit and Hennicker's approach is based on a partial observational equality. Our concepts are similar to ideas of using an explicit observability predicate as presented by Hennicker in [8, 11]. We have adapted some of his ideas to the context of state-based specification and dynamic logic. Some detailed comparisons have been made in Section 5.

Other techniques of refinement and implementation for algebraically specified systems are based on a strategy of applying different implementation, restriction and other specificaton-building operations, see [9] Chapter 8 or [22]. We have tried to capture the idea of implementation or refinement in a single construct.

The idea of using modal logics as logics for state-based specification and implementation is not new. A few examples include the specification language COLD [23, 24] or the Modal Action Logic [25, 26] and successive work [27, 28]. Recently, modal logics have also been used in the context of coalgebras. An example is [29], where coalgebras are used to define the semantics for object-oriented programming supported by a modal logic. Coalgebras can generalise transition systems, and modal logics are a natural choice to capture this in a logic. The approach taken by the developers of hidden algebras [15, 16] – on which our semantical structures are based – follow the same direction.

We have used a dynamic logic, i.e., a modal logic that makes commands explicit. Modalities are indexed by commands. Another form of modal logics – temporal logics – are also suitable for the specification and development of state-based systems. In particular TLA (the Temporal Logic of Actions) [30, 31] offers advanced concepts for implementation and refinement. We have chosen a dynamic logic since it provides the more adequate theory for the support of the pre/post-condition technique.

Finally, we shall briefly address our own related work. The basic formal framework – excluding observational aspects – is presented in [18]. Two of our papers extend the ideas of operation refinement: [21] presents refinement ideas for the ASM (Abstract State Machines) notation, [32] is based on the idea of refinement as the matching technique for component composition.

8 Conclusions

We have presented a framework for specifying and developing state-based systems based on the idea of behavioural reasoning and observational implementation. In contrast to related approaches we have used a dynamic logic instead of an equational logic. Modal logics give us more expressive power in the specification of dynamic and reactive systems. The principle disadvantage of reasoning about implementations with the more complex modal logics is compensated using an equationally oriented refinement notion and the definition of observable substructures. One of our objectives is to develop an extension of the pre/post-condition technique for state-based systems, which would allow an improved technique to be used in component-based development approaches considering non-determinism and liveness besides the safety-specification of the classical pre- and postcondition technique. It allows us to use equational reasoning to prove dynamic systems using a refinement inference rule a the key tool.

We have seen that a behavioural notion of implementation is needed for state-based specification where we want to abstract from the realisation of the state itself. Observation is a useful tool in the realisation of specifications based on existing libraries of execuatble components. Our notion of satisfaction for the modal operators and equations on the state sort satisfy this requirement. Our data refinement notion is observational, based on the idea of relativisation.

We can specify dynamic state-based systems solely using modal constructors and equations of sort $state$. In most situations implementations can be proved using the equational refinement relation. A further step in improving the implementation of abstract specifications on predefined standard models might be using congruences to express that

certain elements of the implementation shall be considered equivalent, or observationally equal. Ideas from Hennicker and Bidoit might be suitable.

References

[1] 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.

[2] C.A.R Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580,583, 1969.

[3] K. R. Apt. Ten Years of Hoare's Logic: A Survey – Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[4] Object Management Group. UML 1.3 Specification, 1999. http://www.omg.org/technology/uml.

[5] Bertrand Meyer. Applying Design by Contract. *Computer*, pages 40–51, October 1992.

[6] G.T. Leavens and A.L. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[7] M. Wirsing and M. Broy. A Modular Framework for Specification and Implementation. In J. Diaz and F. Orejas, editors, *TAPSOFT'89 Proceedings Int. Conference on Theory and Practise of Software Development, Barcelona, Spain*, pages I:42–73. Springer-Verlag, 1989.

[8] R. Hennicker. Implementation of Parameterized Observational Specifications. In J. Diaz and F. Orejas, editors, *TAPSOFT'89 Proceedings Int. Conference on Theory and Practise of Software Development, Barcelona, Spain*, pages I:290–305. Springer-Verlag, 1989.

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

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

[11] R. Hennicker. Observational Implementation of Algebraic Specifications. *Acta Informatica*, 28:187–230, 1991.

[12] M. Bidoit, R. Hennicker, and M. Wirsing. Characterizing Behavioural Semantics and Abstractor Semantics. In D. Sannella, editor, *Proceedings ESOP'94*. Springer-Verlag, LNCS 788, 1994.

[13] M. Bidoit, R. Hennicker, and M. Wirsing. Behavioural and Abstractor Semantics. *Science of Computer Programming*, 25, 1995.

[14] R. Hennicker and M. Bidoit. Observational Logic. In M. Wirsing and M. Nivat, editors, *Proceedings 7th Int. Conf. on Algebraic Methodology and Software Technology, AMAST'98*, pages 263–277. Springer-Verlag, LNCS 1548, 1998.

[15] 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.

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

[17] 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.

[18] C. Pahl. A Logical Framework for Horizontal and Vertical Development in a State-based Setting. Report CA-3499, School of Computer Applications, Dublin City University, 1999.

[19] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier Science Publishers, 1990.

[20] C. Stirling. Modal and Temporal Logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 477–563. Oxford University Press, 1992.

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

[22] H. Partsch. *Specification and Transformation of Programs*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[23] H.B.M. Jonkers. An Introduction to COLD-K. In *Algebraic Methods: Theory, Tools and Applications*, pages 139–206. Springer-Verlag, 1989.

[24] L.M.G. Feijs. An Overview of the Development of COLD. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *1st International Workshop on Semantics of Specification Languages, Utrecht, 1993*, pages 15–22. Springer-Verlag, 1994.

[25] J. Fiadero and T. Maibaum. Describing, Structuring and Implementing Objects. In J.W. de Bakker, W.P. Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, LNCS 489*, pages 274–310. Springer-Verlag, 1990.

[26] J. Fiadero, T. Maibaum, and M. Ryan. Sharing Actions and Attributes in Modal Action Logic. In T. Ito and A.R. Meyer, editors, *Proceedings Conference on Theoretical Aspects of Computer Software TACS'91, Sendai, Japan*. LNCS 526, Springer-Verlag, September 1991.

[27] J.L. Fiadero and T. Maibaum. Verifying for Reuse — Foundations of Object-oriented System Verification. Technical report, Imperial College, 1993.

[28] J. Fiadero and T. Maibaum. Interconnecting Formalisms: Supporting Modularity, Reuse, and Incrementality. In Gail E. Kaiser, editor, *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 72–80. ACM Software Engineering Notes 20 (4), October 1995.

[29] A. Kurz. Specifying Coalgebras with Modal Logic. In *Proc. Coalgebraic Methods in Computer Science, CMCS'98 Lisbon, Portugal*. ENTCS Volume 11, 1998.

[30] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[31] L. Lamport. Specifying Concurrent Systems with TLA$^+$. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. IOS Press, Amsterdam, 1999.

[32] C. Pahl. Modal Logics for Reasoning about Object-based Component Composition. In *Proc. 4rd Irish Workshop on Formal Methods, July 2000, Maynooth, Ireland*. 2000.