ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**D Duke, University of York, UK and A S Evans, University of Bradford, UK
(Eds)**

BCS-FACS Northern Formal Methods Workshop

Proceedings of the BCS-FACS Northern Formal Methods Workshop,
Ilkley, UK. 23-24 September 1996

A Model for Dynamic State-Based Systems

C. Pahl

Springer

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

A Model for Dynamic State-based Systems

Claus Pahl

Department of Computer Science, Trinity College, University of Dublin
Dublin 2, Ireland

Abstract

Objects can be the structuring mechanism, i.e. the main building blocks of a system. In this paper a computation model formalising the notion of an object is presented which can be used for defining state-based systems. It can be used for both describing semantics of programming languages and defining semantics for a specification logic. A general, formalised and extendible notion of a model for state-based systems is the basic prerequisite for any attempt towards formal language engineering, i.e. for defining and integrating languages in a formal way.

Functional aspects are integrated in the presented model. It will be shown that a transition system emerges from the given definition of an object as an entity with state, which may be modified by procedures. Therefore, concurrency can also be dealt with in this approach.

1 Introduction

A *computation model* is a framework around a semantic structure in the sense of logics. The presented computation model is tailored towards state-based requirements. It is intended to be the basis of an approach of *formal language engineering* in this universe of discourse. Formal language engineering copes with defining and integrating languages and language concepts in a structured and comprehensible way.

Formal language engineering is the central issue of the HCM project De Stijl which has recently started. De Stijl¹ is an acronym for Design and Specification through Interfacing and Joining Languages. It aims at providing formal background to the integration of languages. The presented paper is approach towards formal definition of state-based languages within the *De Stijl* context.

In our approach, *objects* are the central building blocks of systems. An object is a formalised notion of an entity which has a (changeable) internal state in which functions can be evaluated and which can be modified by procedures.

We will choose a completely state-based setting because we see *state* as an intrinsic characteristic of a certain kind of systems. Specifying these systems we expect a sufficient functionality to deal with state. This is similar to functional programming where we expect mechanisms to handle the evaluation of functions. Lambda calculus is taken for granted in the functional paradigm. In the same sense we will provide basic mechanisms to deal with states: a state transition system based on a formal notion of objects as state-based entities.

1.1 Paradigms

Functional programming or *specification* (see Figure 1) deals with evaluation by using reduction techniques. This is the principle realized by λ -calculus:

$$(\lambda x.M[x])N \rightarrow M[N]$$

$\lambda x.M[x]$ is the evaluation rule, N denotes a value and $M[N]$ denotes a value after reduction.

State-based development uses state transition to model change (without involvement of concurrency and communication between processes, see Figure 1). Finite state machines or Turing machines are examples of transition systems.

$$STATE \xrightarrow{p} STATE'$$

¹See <http://www.cs.tcd.ie/DeStijl/> for further information.

paradigm	principle	technique
process-based	communication	message passing
state-based	change	state transition
functional	evaluation	λ -calculus

Figure 1: Paradigms

p is a labelled transformation rule, $STATE$ denotes the current state and $STATE'$ the next state reached by p .

Process-based development describes communication by interaction (see Figure 1). Two processes may interact by a number of message passing principles or by indirect communication principles. The process-based paradigm embraces concurrent, distributed and reactive systems.

$$a.x.P[x]|\bar{a}V.Q \rightarrow P[V]|Q$$

P and Q are processes which communicate synchronously via a channel a ($\bar{a}$ denotes the inverse direction). The value V is passed via the channel and bound to x . This is the rule for computing interaction formulated in the π -calculus (see [15, 16]).

There are different approaches to process specification. In the algebraic approach means are provided to describe simultaneous behaviour of multiple control threads by behaviour expressions. These expressions are built up by operators on actions. A behaviour expression is different from a value expression in the functional paradigm in that it isn't a denotation of a value. Transition systems are one possibility to interpret behaviour, which (as we will see) emerge quite easily when we start with a state-based, formalised notion of objects. A behaviour expression may denote a state possibly constructed by transitions on other states. This exhibits the fundamental difference between the functional paradigm on one hand and the process- as well as state-based paradigm on the other. The latter two both rely on a notion of state and behaviour.

When specifying a system, all these paradigms (Fig. 1) can be applied to describe different aspects of the system. A whole system could consist of a number of independent agents communicating with each other. Each agent (a process) is state-based. In each state evaluation of functions can take place. If we have to work with state-based systems, either to develop a system to be implemented or to define a state-based language, we want to express the idea of change and modification by appropriate means. Evaluation is another aspect which e.g. may be used to compute functions in a certain state. Those basic means should be provided by the framework used.

Modelling dynamic, state-based systems requires adequate computation models. Mechanisms like Turing-machines are not practical enough. They lack the possibility of being adapted to different levels of abstraction. Algebras are suited only to a certain degree. They are not based on a notion of state and, thus, state has to be modelled explicitly. An inherently state-based computation model, adaptable to different levels of abstraction shall be presented. Supporting state-based approaches is of major importance, if algorithms are seen in the von-Neumann or Turing-sense, i.e. as descriptions of state-modifying functions. The presented computation model is the formal transcription of the idea of an abstract machine. Taking a collection of objects as independent agents, possibly interacting with each other, a process-oriented framework is necessary.

1.2 Why Computation Models?

The notion of data type is fundamental in computer science. The definition of a data type is done on two levels. On the first (syntactical) level names for value sets and names, argument and range domains for operations are provided by a signature. On the second (semantical) level sets and functions corresponding to the signature are defined. In logics, the notion of *structure* is used for the collection of sets and functions. This approach is about state-based systems, i.e. systems with a notion of modification, based on the execution of operations (the *computation* of operations). Thus, the notion of *computation model* is chosen. It emphasises the operational character, but, as opposed to *abstract machine*, it also comprises the formal aspect.

The defined computation model serves two purposes. It can be used as a semantic structure for the definition of a notion of validity for formulae of a specification logic. It is an operational computation model usable for the definition of state-based, concurrent programming languages.

Computation models exist for the usual first-order logics, e.g. algebras for the equational logic of algebraic specifications [21, 4, 5]. In the area of state-based systems Kripke-structures exist, known from modal logic [11], which are based on a notion of state. From the area of von-Neumann-architectures and imperative programming, the model of Turing-machines is known. Process-oriented languages are often defined on transition systems or similar models [14, 10]. These approaches do not seem to be suitable to support both intended purposes in an optimal way. These approaches have to be integrated in one framework. Therefore, the model of Σ -**objects** is developed, where Σ denotes the given vocabulary, i.e. the signature. The notion *object* was chosen to emphasise the decomposition of a system into a number of independent subsystems (the objects).

Section 2 defines the computation model. In section 3 the functional paradigm is considered. Section 4 extends the approach for process-based systems. In section 5 some applications are presented.

2 An Abstract Machine

The computation model presented in this section can be illustrated by the idea of an *abstract machine*.

In first-order logic non-empty sets² with operations and relations are called *structures*. They serve as semantics for terms and formulae of a logic (eventually over a given vocabulary, i.e. a signature). In universal algebra structures without relations are called *algebras*. They have shown their ability to define semantics for functional specification in first-order equational logic (see algebraic specification [21, 4, 5]). The approach of algebras shall be adopted here. But an adaptation towards state-based requirements has to be carried out.

2.1 Types

For the following approach, a number of different basic *types* (or *sorts*) are provided. These are the primitive types integer and boolean and a type void, whose value set only consists of the undefined value *null*. Syntactically, the types can be referenced by the type identifiers *int*, *bool* and *void*. Every type has an associated partial order:

$$Int_\omega = (INT_\omega, \sqsubseteq), \quad Bool_\omega = (BOOL_\omega, \sqsubseteq), \quad Void = (\{null\}_\omega, \sqsubseteq)$$

INT is the set of numbers $\mathcal{Z}$, $BOOL = \{true, false\}$. The partial orders are ω -extensions, i.e. the sets of values are extended by the undefined value ω ³. The set elements are (except ω) not ordered wrt. $\sqsubseteq$. These partial orders are called *flat*. All sets are subsets of the universe of values VAL . The sort identifier for a global name space is *ident*. The partial order $Ident_\omega = (IDENT_\omega, \sqsubseteq)$ on the set of identifiers $IDENT$ is flat, too. The value sets are defined as flat partial orders to deal with partiality and later to allow us to define a denotational semantics for recursive functions.

2.2 Signature

The abstract syntax of a language is presented in the form of signatures and terms. A signature describes the syntactic structure of an object. An **object signature** Σ consists of

- a set of data sorts $S = \{s_1, \dots, s_l\}$ and a state sort *state* with *state* $\notin S$,
- a set of state identifiers $Z = \{z_i, | i = 1, \dots, m\}$ with an associated data sort s_i for each z_i ; $s_i \in S, i = 1, \dots, m$ (we write $z_i : s_i$ for associating a state identifier with a sort)
- a set of operations OP

$$op : state \times s_1 \times \dots \times s_n \rightarrow state \times s$$

with $op \in OP, s_i, s \in S, i = 1, \dots, n$.

²Sets are assumed to be finite, because they represent a formal abstraction of a computer.

³ ω indicates the undefined values (partiality can be dealt with by ω). In the context of partial orders $\perp$ is sometimes used instead of ω , both denoting the bottom element in a partially ordered set.

The right-hand side of the operation signature, i.e. $state \times s$, expresses that an operation has two different kinds of effect: one side-effect on the state, i.e. a value of sort $state$, and a functional effect, i.e. a value of sort s . This will allow us to define functions for evaluation in a state and procedures for state transition in a uniform way.

2.3 State

The notion of *environment* generally describes the binding of values to identifiers (see [12] or [1]). Here, this concept is realized by the notion of *state*. For this, a mapping $STATE$ maintaining the bindings is assumed. Names (from $IDENT$) are mapped by $STATE$ to values from VAL . The carrier set of $state$ is denoted by $State$. $STATE$ is an arbitrary element from $State$. $STATE$ is always interpreted as a mapping. $(VAL, \sqsubseteq_{VAL})$ defines a flat value space. $IDENT$ might be a superset of Z , the set of state identifiers used in a signature.

There are a variety of possibilities of defining state for a collection of objects. Some approaches only maintain a global system state. We have realized state local to the objects since we want to describe systems of independent agents/objects. Some approaches consider the whole structure as determining an object state. We focus here on the essential part (assuming that operation definitions are unchangeable in the lifetime of an object). This essential part is in fact the notion of environment usually used for functional approaches.

2.4 Terms

Terms are used to denote semantic entities. The association between terms and semantic entities is defined by an **interpretation**. Terms are constructed on names, function application, brackets, etc. A type (or a sort) will be associated with each term. Σ -**terms** are those terms for which sort and function names are in the signature Σ . Therefore, a signature induces a set of syntactically correct expressions, constructed by free variables and the operation symbols of the signature.

Let $X = (X_s)_{s \in S}$ be a S -sorted set of *free variables* based on a name space $IDENT$. Only state identifier and formal operation parameters are allowed to be free in Σ -terms.

We will distinguish two forms of operations. These are the subsets $ATTR$ and $PROC$ of OP . Each operation is assigned to one of these subsets. Elements from $ATTR$ are called *attributes* and are pure functional operations, i.e. they are not allowed to modify the state. $PROC$ is a set of procedures, i.e. operations that can modify the state but don't yield a (functional) value.

$$\overline{ATTR} := \{pr_2(a) | a \in ATTR\} \text{ and } \overline{PROC} := \{pr_1(p) | p \in PROC\}$$

Thus, $a : state \times s_1 \times \dots \times s_n \rightarrow state \times s$ induces $pr_2(a) : state \times s_1 \times \dots \times s_n \rightarrow s$. These constructions allow us to project on the essential part of the result: the modified state as the first component of an operation result for procedures and a value as the second component for attributes⁴. This construction allows us to associate an operation with its essential sort.

A Σ -**term** of data sort $s \in S$ over X_s is a variable, a constant, or an attribute call, i.e.

- every $x \in X_s$,
- every $\bar{a} \in \overline{ATTR}$, if $a \in ATTR$ with $a : state \rightarrow state \times s$ and $\bar{a} = pr_2(a)$,
- every $pr_2(a)(st, t_1, \dots, t_n)$ with $a \in ATTR$ $a : state \times s_1 \times \dots \times s_n \rightarrow state \times s$ and t_i Σ -term of data sort $s_i \in S$, $i = 1, \dots, n$; $st : state$.

A Σ -**term** of state sort $state$ is a state variable, a procedure call or a (predefined) command, i.e.

- the identifier st to denote the state mapping,
- every $pr_1(p)(st, t_1, \dots, t_n)$, with $p \in PROC$ $p : state \times s_1 \times \dots \times s_n \rightarrow state \times s$ and t_i Σ -term of data sort $s_i \in S$, $i = 0, \dots, n$; $st : state$.

⁴In an extension we could allow attributes with side effects, but now we want to distinguish these conceptually different forms of operations.

Figure 2: Object as an abstract machine

- $asgn(st, z, t)$, $seq(st, c_1, c_2)$, $if(st, t, c)$, $skip$ and $stop$, if the sort of Σ -term t is a data sort and $st : state$, $c, c_1, c_2 : state$ and z a state identifier.

Besides the operations (given by the signature) we have also introduced some predefined commands: *asgn*, *seq*, *if*, *skip*, *stop*. These commands are the primitive statements we usually find in an imperative programming language. Loops can be defined in a denotational style, if recursive definitions are permitted (see sections 3 and 5.1).

2.5 Objects

Objects are the main semantical entities. They shall now be defined like a semantic structure for logics. Together with the predefined commands (*asgn*, *if*, ...) an abstract machine is provided. Let Σ be a state signature. A Σ -**object** A then consists of

- a $S' = S \cup \{state\}$ -sorted set $\{A_s | s \in S'\}$ of carrier sets. The carrier set A_{state} will be denoted with $State$.
- for each $op \in OP$ with $op : state \times s_1 \times \dots \times s_n \rightarrow state \times s$ a mapping $op^A : State \times A_{s_1} \times \dots \times A_{s_n} \rightarrow State \times A_s$.

Every Σ -object has an associated **current state** $STATE \in State$. This mapping yields the value which is bound to an identifier. $STATE$ is a dynamic mapping, i.e. it can be modified.

Figure 2 visualises a Σ -object in a current state $STATE$. In the upper half, we can find the instructions of the abstract machine (comprising attributes, procedures, and predefined commands). The lower part contains the values that can be used by the operations. The object is centred around its current state. Only the operations can be accessed from outside the objects. Access to the state is encapsulated. Taking the attributes, procedures and commands as instructions, Σ -objects form a formally defined abstract machine.

2.6 Interpretation

Terms are interpreted by elements of Σ -objects. This mapping is realized by a *variable assignment* v and its inductively defined extension v^* (an *interpretation*) for arbitrary terms. Each term depends on the current state. The interpretation is always a pair of a (possibly new) state and a value component which indicates that we want to describe functional and state-transformational effects in a uniform way.

$$v(STATE, x) := [STATE, STATE(x)]$$

Variables x don't change the state (first component). The value of a variable x is stored in $STATE$ (second component) and can be obtained by applying $STATE$ to x .

$$v^*(STATE, a(st, t_1, \dots, t_n)) := [STATE, \mathcal{I}(a(st, t_1, \dots, t_n))]$$

Attributes a don't change the state. Their value is evaluated in a functional style, e.g. as in algebraic specifications (here indicated by applying $\mathcal{I}$).

$$v^*(STATE, p(st, t_1, \dots, t_n)) := [\mathcal{I}(p(st, t_1, \dots, t_n)), null]$$

Procedures p change the state (which is evaluated in a functional style), but don't yield a value (indicated by $null$).

A Σ -object realizes the idea of an abstract machine. Signatures are describing the vocabulary that the machine knows. Terms denote elements of this machine. The machine has an internal state, described by state components z_i and stored in $STATE$. Attributes are functions on state components and parameter values. The commands *asgn*, *if*, *seq*, *skip* and *stop* and procedure calls are machine instructions which are allowed to change the state.

The state modifying construct *asgn* shall now be defined by substitution on the state. $substitute(STATE, x_1 \mapsto v_1, \dots, x_n \mapsto v_n)$ substitutes in the mapping $STATE$ the former binding for a name x_i by a binding of x_i to the value v_i for $i = 1, \dots, n$. $substitute$ yields the first argument in its modified form. $substitute$ is defined as follows:

$$substitute(STATE, x_1 \mapsto v_1, \dots, x_n \mapsto v_n) := STATE'$$

$$where \ STATE'(z) = \begin{cases} STATE(z) & \text{for } z \neq x_i, i = 1, \dots, n \\ y_i & \text{for } z = x_i, i = 1, \dots, n \\ \perp & \text{for } z = \perp \end{cases}$$

The x_i must be pairwise distinct.

$asgn(STATE, z_i, t)$ associates the value of term t to the state component z_i in state $STATE$. The semantics of *asgn* is defined by the interpretation v^* .

$$v^*(STATE, asgn(STATE, z_i, t)) := [substitute(STATE, z_i \mapsto v^*(STATE, t)[2]), null]$$

skip and *stop* are special primitive commands.

$$v^*(STATE, skip) := [STATE, null]$$

The state remains unchanged, no value is yielded. Therefore, *skip* is attribute as well as procedure, but shall be treated as a command from now on.

$$v^*(STATE, stop) := [\perp, \perp]$$

A further execution of the program is not defined. If v^* is strict wrt. $\perp$, this is realized.

On these primitive commands, a conditional command and a sequence can be easily realized.

$$\begin{aligned} v^*(STATE, if(b, c)) := & \\ & \text{if } v^*(STATE, b)[2] =_{bool} true \text{ then} \\ & \quad v^*(STATE, c) \\ & \text{else if } v^*(STATE, b)[2] =_{bool} \perp_{bool} \text{ then} \\ & \quad [STATE, \omega] \\ & \text{else } [STATE, null] \\ & \text{end} \\ & \text{end} \end{aligned}$$

The conditional command c is only executed, if the evaluation of the condition b yields $true$. *if* is strict wrt. $b = \perp$.

$$\begin{aligned} v^*(STATE, seq(com1, com2)) := & \\ & v^*(v^*(STATE, com1)[1], com2) \end{aligned}$$

Figure 3: State transformation on a stack

The second command works on the successive state evaluated by the first command. The sequence operator is associative.

In Figure 3 the state transformation is visualised for the classical stack example. By executing `push(st,6)` only the state is changed (see centre of objects). The bindings are modified by assigning new values to the state variables `stack` and `nb_elements`.

2.7 Discussion

For state-based systems we can find a number of approaches trying to formalise the notion of state, e.g. MLCM models [6], d-oids [2, 3], Kripke-structures [11], and evolving algebras [7]. The first three approaches are intended to give semantics for specification logics. But mostly, they are not presented in a modular way and they don't cover a variety of language concept so that dealing with real languages in the sense of formal language engineering may cause difficulties. In the case of evolving algebras there is an integration of process-oriented mechanisms, but here a formal treatment of functional aspects is lacking.

3 Functional Aspects

3.1 Recursion and λ -Calculus

Evaluation of functions in λ -calculus is state-less. Therefore, we can think of the evaluation of our attributes in a particular state as function evaluation in λ -calculus, but also procedures and commands are interpreted by functions taking states as arguments and yielding possibly changed states as part of the result. Thus, we will consider both forms of operations in this section.

We haven't dealt with explicit recursive definitions of operations so far. Now, recursive operations shall be defined in a denotational style, i.e. a restricted (implicit) λ -calculus is introduced and formalised by domain theory (fixpoint semantics). λ -calculus is the essence of functional programming. We will not make a λ -notation with λ -abstraction and λ -application explicit. We won't have λ -terms on the syntactic level, but operation definition and application is still available. First, a new sort *operation* is introduced which denotes functions on the semantic level. They will be used to define recursive operations by fixpoint semantics. A notion of an operation sort is needed to type operation definitions.

The set of sorts S of a signature shall always be extended by the sort symbol *operation*. Syntactic constructs of sort *operation* are terms which denote functions on carrier sets of data sorts and the sort *state*. Each $op \in OP$ with $op : state \times s_1 \times \dots \times s_n \rightarrow state \times s$ is of sort *operation* and is interpreted by a function of type $State \times A_{s_1} \times \dots \times A_{s_n} \rightarrow State \times A_s$ ⁵ in a corresponding Σ -object.

⁵This allows us to type operation symbols in the way it is done in typed λ -calculus.

The consistency of the definitions so far can be proved. A prerequisite for denotational semantics is that all sets involved are complete partial orders. We have already started considering this by defining the basic types as flat partial orders. Functions on these complete partial orders have to be continuous, i.e. they have to preserve least upper bounds in chains of the cpos.

Theorem 3.1 *Let v^* be an interpretation for an object signature Σ .*

1. *for each term t of sort $s \in S \cup \{state\}$ holds:
 $v^*(STATE, t) \in A_{State} \times A_s$ for all states $STATE \in State$*
2. *v^* is continuous.*

This theorem states that the definition of v^* is well-defined and continuous. This can be proved similar to fixpoint semantics for functions without state in a pure functional framework, i.e. via structural induction. For the functional part, i.e. the terms of data sorts, this proceeds as usual (see e.g. [12]). But for the state-related constructs, there are some prerequisites:

- $STATE$ is a continuous mapping.
- $(State, \sqsubseteq)$ is a cpo.
- $substitute$ is a continuous function on $(State, \sqsubseteq)$.

$STATE$ as a mapping on flat cpos $(IDENT_\omega, \sqsubseteq)$ and $(VAL_\omega, \sqsubseteq)$ is continuous. $State$ is a finitary set of mappings $STATE$ due to the finiteness of $IDENT_\omega$. With a bottom element in $State$ and only finite chains, $(State, \sqsubseteq)$ is a cpo. Based on these properties the consistent definition of v^* for terms of sort $state$ can be proved by structural induction on the terms⁶.

Continuity is a prerequisite for applying fixpoint semantics to define semantics for recursive operation definitions. We will only deal here with direct recursion: $op(x_1, \dots, x_n) = t$ where op is called within t .

3.2 Fixpoint Semantics

We will now introduce a fixpoint operator μ which can be applied to the functional Φ of a recursive equation used to define an operation.

Let $P \equiv op(x_1, \dots, x_n) = t$ for $op : state \times s_1 \times \dots \times s_n \rightarrow state \times s$.

$$v^*(STATE, P) := \mu \Phi_{v^*}(STATE, P)$$

with

$$\Phi_{v^*}(STATE, P) : state \times operation \rightarrow state \times operation$$

The functional Φ can be expressed by a lambda abstraction on the level of Σ -objects:

$$\Phi_{v^*}(STATE, P) = \lambda F. \lambda x_1, \dots, x_n. v^*(STATE, t)$$

where F is a variable of type (i.e. sort) $operation$ and F substitutes op syntactically in t . $\Phi_{v^*}(STATE, P)$ is the functional belonging to the recursive operation, $\mu \Phi_{v^*}(STATE, P)$ denotes the least fixpoint of $\Phi_{v^*}(STATE, P)$ which is the semantics for the program fragment P .

Lemma 3.1

1. $\Phi_{v^*}(STATE, P)$ is continuous.
2. $\Phi_{v^*}(STATE, P)$ is continuous, therefore, the least fixpoint of Φ_{v^*} exists, i.e.
 $\mu \Phi_{v^*}(STATE, P)$ where $\mu \Phi_{v^*}(STATE, P) = \sqcup \{(\Phi_{v^*}(STATE, P))^i(\perp) \mid i \geq 1\}$.

⁶We will only sketch proofs here, for full proofs see [17]).

This is proved based on Theorem 3.1 and Kleene's fixpoint Theorem (see e.g. [12]).
The operation call of a recursively defined operation can be defined as follows:

$$\begin{aligned} v^*(STATE, op(a_1, \dots, a_n)) &:= \\ v^*(STATE, op)(STATE, v^*(STATE, a_1)[2], \dots, v^*(STATE, a_n)[2]) \\ \text{with } v^*(STATE, op) &:= \mu \Phi_{v^*}(STATE, op(x_1, \dots, x_n) = t) \end{aligned}$$

Theorem 3.1 can now be extended for recursive operations.

Theorem 3.2 *For every recursive operation $P \equiv op(x_1, \dots, x_n) = t$ holds:*

1. $v^*(STATE, P) \in A_{state} \times A_s$ for all states in *State*,
2. $v^*(STATE, P)$ is continuous.

This can be proved based on the former results, the continuity of the fixpoint operator μ , and the continuity of function composition.

Now, fixpoint semantics are defined. Therefore, recursive operation definitions are permitted.

4 Process-based Aspects

In the introduction process-based specification has been mentioned. This approach should now be dealt with. Process theory is normally understood as the modelling and verification of the behaviour of a concurrent system [19]. We will concentrate here on the modelling aspect.

Models for concurrent systems can be classified in three dimensions (see e.g. [20]).

- Interleaving and non-interleaving models are distinguished. Examples of interleaving models are transition systems [20], synchronisation trees [13] and acceptance trees [9]. These models abstract from the fact that a system is actually composed of independent agents and model the behaviour in terms of purely sequential patterns of actions, i.e. by nondeterministic merging or interleaving the sequential behaviours of the agents. Non-interleaving models are e.g. Petri nets.
- Models may be distinguished by their treatment of nondeterminism, i.e. linear time versus branching time models.
- The treatment of state is another criterion. Some models abstract from state, others have an explicit notion of state.

We will deal with interleaving, linear time models with explicit state.

4.1 Transition Systems

Transition systems are the basis of a number of process-based specification approaches (among them Hoare's CSP [10] and Milner's CCS⁷ [14]). The following definitions are based on [20]. We will show in this section how a transition system can be obtained from a Σ -object, how morphisms on both levels are related and how common operations on transition systems are realized on the Σ -object level.

A **transition system** TS is a quadruple $(S, i, L, Tran)$ where

- S is a set of states with initial state i ,
- L is a set of labels,
- $Tran \subseteq S \times L \times S$ is the transition relation.

⁷CCS is originally defined on synchronisation trees; but a transition system can be unfolded to a synchronisation tree (see [20] Chapter 4).

We write $s \xrightarrow{a} s'$ for $(s, a, s') \in Tran$.

Let A be a Σ -object with carrier sets A_s for all data sorts s , $State$ the carrier set of states, a set of functions $op^A : State \times A_{s_1} \times \dots \times A_{s_n} \rightarrow State \times A_s$ for all $op \in OP$ and a current state $STATE \in State$. Then TS_A is the transition system $(S, i, L, Tran)$ associated with A defined by

- $S := State$ and $i := STATE_\omega$ where $STATE_\omega(x) = \omega$ for all $x \in IDENT$
- $L := \{p \mid p \in PROC(\Sigma)\} \cup \{asgn, if, seq, skip, stop\}$
- $Tran := \{(s, a, s') \mid s, s' \in State, a \in PROC^+, a_A(s, x_1, \dots, x_n)[1] = s' \text{ for arbitrary } x_1 \in A_{s_1}, \dots, x_n \in A_{s_n}\}$

$PROC^+$ denote the extension of $PROC$ by all predefined commands.

Lemma 4.1 TS_A (constructed by the procedures and predefined commands on states in a Σ -object) is a transition system.

$$STATE \xrightarrow{p} STATE'$$

p is a (parameterisable) transformation rule defined by a procedure or a command, $STATE$ denotes the current state and $STATE'$ the next one, i.e. $p(STATE, \dots)[1] = STATE'$. The transition system is constructed by taking procedures or commands as transitions between states. A transition system could be completely defined within a Σ -object without referring to the signature. Function identifiers could be used as labels, but we see labels as syntactic elements and therefore choose signature symbols (commands and procedures) as labels. Constructs of sort *state* are denotations of transitions in a Σ -object.

$*$ denotes the idle transition $(s, *, s)$. $*$ is the equivalent of *skip*. $L \rightarrow_* L'$ expresses that $*$ does not appear in either L or L' (analogously for the Cartesian product $\times_*$ instead of the function space constructor $\rightarrow_*$).

4.1.1 Morphisms

Morphisms in general are mappings between entities preserving their structure. We will define morphisms on the transition system level as well as on the object level and show how a morphism on the former can be derived from one on the latter.

ts-morphisms can be used e.g. to express that a process refines another or that one process is a component of another (see e.g. [20]).

A **ts-morphism** $f : T_0 \rightarrow T_1$ with $T_0 = (S_0, i_0, L_0, Tran_0)$ or $T_1 = (S_1, i_1, L_1, Tran_1)$ is a pair $f = (\sigma, \lambda)$ where

- $\sigma : S_0 \rightarrow S_1$
- $\lambda : L_0 \rightarrow_* L_1$ such that $\sigma(i_0) = i_1$ and $(s, a, s') \in Tran_0 \Rightarrow (\sigma(s), \lambda(a), \sigma(s')) \in Tran_1$

Let Σ be a signature, A and B be Σ -objects. $h : A \rightarrow B$ is a Σ -**homomorphism**, if

- $h = \{h_s : A_s \rightarrow B_s \mid s \in S \cup \{state\}\}$ is a S -sorted set of mappings,
- h preserves the operations, i.e. for $op : state \times s_1 \times \dots \times s_n \rightarrow state \times s$, $a_i \in A_{s_i}$, $st \in State$ holds:
 - $h_s(a^A(st, a_1, \dots, a_n)[2]) = a^B(h_{state}(st), h_{s_1}(a_1), \dots, h_{s_n}(a_n))[2]$, if $a \in ATTR$,
 - $h_{state}(p^A(st, a_1, \dots, a_n)[1]) = p^B(h_{state}(st), h_{s_1}(a_1), \dots, h_{s_n}(a_n))[1]$, if $p \in PROC^+$.

Lemma 4.2 Let A and B be Σ -objects and $h : A \rightarrow B$ a Σ -homomorphism. Then a ts-morphism $t = (\sigma, \lambda)$ for the associated transition systems of A and B can be defined from h by:

- $\sigma := h_{state}$

- $\lambda := id_{PROC^+}$, where id_{PROC^+} is the identity on $PROC^+$

$\sigma(STATE_{\omega_A}) = STATE_{\omega_B}$ holds due to the assumption that every $STATE$ is defined totally on $IDENT$ and that the initial state of every object maps every identifier to ω . $(s, a, s') \in Tran_0 \Rightarrow (\sigma(s), \lambda(a), \sigma(s')) \in Tran_1$ holds, because from

$$h_{state}(p^A(st, a_1, \dots, a_n)[1]) = p^B(h_{state}(st), h_{s_1}(a_1), \dots, h_{s_n}(a_n))[1]$$

we get that st is mapped to $h_{state}(st)$ (which is the definition for σ) and the same holds for the results of both sides of the equation (which represent the states s' and $\sigma(s')$).

Therefore, a transition morphism can be obtained from a Σ -homomorphism. This construction disregards the possibility of renaming labels (or actions) which is allowed in transition morphisms. A ts-morphism with renaming labels could only be realized on the level of Σ -objects, if a renaming signature morphism for object signatures is introduced. Then, the notion of homomorphism has to be extended so that functions in Σ -objects are labelled by the corresponding signature symbols and that these labels might be modified.

4.2 Operations on Transition Systems

The operation *product* on transition systems shall be presented as it are usually defined (see [20]) as a basis for parallel composition. The effect is analysed in the underlying Σ -objects. Simpler operations as *restrict* or *relabel* are introduced as well.

4.2.1 Relabelling

Let $T = (S, i, L, Tran)$ be a transition system, let $\lambda : L \rightarrow L$ be a total function. A **relabelling** $T\{\lambda\}$ is defined by $(S, i, L', Tran')$ where

$$Tran' = \{(s, \lambda(a), s') \mid (s, a, s') \in Tran\}$$

The actions names are renamed. A relabelling for Σ -objects is done via a signature morphism, which renames the procedures. The objects themselves are not effected.

4.2.2 Restrict

Let $T' = (S, i, L', Tran')$ be a transition system. Assume $L \subseteq L'$ and let $\lambda : L \rightarrow L'$ be a partial function, here the inclusion morphism taking a in L to a in L' . $T' \downarrow \lambda$ is the **restricted transition system** $(S, i, L, Tran)$ with

$$Tran = \{(s, a, s') \in Tran' \mid a \in L\}$$

The number of actions are restricted. This operation applied to a Σ -object reduces the procedures that can be used for state transition. Let A' be a Σ' -object. Then the corresponding restricted Σ -object $A' \downarrow \lambda$ is defined by a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ with $\sigma(\Sigma) = \Sigma'$ except $opns(\Sigma) = opns(\Sigma') \cap L$ and $A' \downarrow \lambda = A'$ except that there is no function for a symbol in $L' \setminus L$. $A' \downarrow \lambda$ is a Σ -object by construction, σ is an embedding signature morphism. By construction $T' \downarrow \lambda$ is the transition system obtained from $A' \downarrow \lambda$.

4.2.3 Products on Transition Systems

Let T_0 and T_1 be transition systems defined by $T_0 = (S_0, i_0, L_0, Tran_0)$ or $T_1 = (S_1, i_1, L_1, Tran_1)$. The **product** $T_0 \times T_1$ is defined by $(S, i, L, Tran)$ where

- $S = S_0 \times S_1$ with $i = (i_0, i_1)$ and projections $\rho_0 : S_0 \times S_1 \rightarrow S_0$ and $\rho_1 : S_0 \times S_1 \rightarrow S_1$
- $L = L_0 \times_* L_1 = \{(a, *) \mid a \in L_0\} \cup \{(*, b) \mid b \in L_1\} \cup \{(a, b) \mid a \in L_0, b \in L_1\}$ with projections π_0 and π_1 .
- $(s, a, s') \in Tran_* \Leftrightarrow (\rho_0(s), \pi_0(a), \rho_0(s')) \in Tran_{0*} \wedge (\rho_1(s), \pi_1(a), \rho_1(s')) \in Tran_{1*}$

Products are special kinds of parallel composition. More general forms of parallel composition can be obtained by applying restriction and relabelling on products. All conceivable parallel compositions which occur by synchronisation can be represented by a combination of those three operations (see [20] chapter 2.2.3 and 2.2.4 for more details).

Communication of parallel processes takes place via *actions* of mutual synchronisation, possibly with the exchange of values. One possibility to do so, is to communicate over channels by linking ports together. Input actions have to be connected with output actions; these two together form a synchronisation action.

Products of transition systems are characterised by states as pairs and labels as pairs. A sample product transition is therefore

$$((s_0, s_1), (a, b), (s'_0, s'_1))$$

where (s_0, s_1) is the current system state which is transformed to (s'_0, s'_1) . By (a, b) a synchronisation between two objects can take place during the transition, if a and b denote suitable actions.

4.2.4 Products on Σ -objects

There is no equivalent notion of product in the computation model of Σ -objects so far. A Cartesian product is neither defined on states nor on procedures. Collections of objects have to be introduced. These objects are administered in an environment which stores references from object identifiers to objects. This construction is needed when communication between objects has to be introduced.

The **product operator** $\times$ applied to two objects produces a pair of objects. Let A and B be Σ -objects with possibly different signatures.

$$A \times B := (A, B)$$

(A, B) can be associated with the transition system $T_A \times T_B$ where T_A and T_B are the transition systems obtained from A and B (resp.).

We assume that Σ -objects are uniquely named. There might be name clashes between internal names in objects (variables, operations). If necessary, these names are associated with their origin (the name of the object).

To have a Σ -object associated with every transition system we will introduce here the notion of a *constructed Σ -object*. A **constructed Σ -object** $C_{A \times B}$ of two different Σ -objects A and B is defined by:

- the carrier sets for data sorts from A and B indexed with their origins. The carrier set $State_{A \times B}$ is defined by $\{(STATE_A, STATE_B) | STATE_A \in State_A, STATE_B \in State_B\}$.
- operation names are indexed by their origins and then unified (disjointly). The origin of each operation should be kept accessible.

The current state is $STATE_{A \times B} = (STATE_A, STATE_B)$ where $STATE_A$ and $STATE_B$ are the current states of A and B . The initial state is $STATE_{A \times B_\omega} = (STATE_{A_\omega}, STATE_{B_\omega})$.

Because there is no order on the objects of a product, the product operator is associative and commutative.

4.2.5 Relating Products

Now a product of transition systems can be obtained from a product of Σ -objects by taken the construction for simple transition systems and by defining:

$$L_{A \times B} := \{(p_A, skip) | p_A \in PROC_A^+\} \cup \{(skip, p_B) | p_B \in PROC_B^+\} \\ \cup \{(p_A, p_B) | p_A \in PROC_A^+, p_B \in PROC_B^+\}$$

4.2.6 Communication

In a product transition $((s_0, s_1), (a, b), (s'_0, s'_1))$ the labels a and b can be arbitrary procedures or commands from underlying Σ -objects. To define a synchronisation they must be special **communication actions** which send or receive a message. Therefore, we extend our collection of predefined commands by

read reads data (from another object)
write writes data (to another object)

read and *write* have to be defined by the interpretation v^* . There are different possible instantiations of these actions depending on the chosen communication paradigm. A process algebra can formalise such a paradigm (see [20] for CCS- and CSP-like communication styles). One possible definition is presented in section 5.3. We will call a product state transition a *synchronisation*, if both actions involved are the corresponding *read* and *write* actions.

5 Applying the Computation Model

As an abstract machine, Σ -objects are well suited to be the basis of a formal definition of an imperative programming language. Another advantage is the use of the computation model for defining modal logics, e.g. the dynamic logic. An extension towards collections of objects allow communication, here in a π -calculus style, to be realized.

In this section small language concepts shall be presented which highlight how the concepts and formalisms presented above might be applied for language definition and language integration.

5.1 Definition of Imperative Languages

The usual constructs of an imperative language can easily be mapped onto the constructs of the abstract machine of Σ -objects.

$$(z := t)(st) := asgn(st, z, t)$$

The assignment $z := t$ is similar to the *asgn*-construction. Loops are defined by using recursion.

$$\begin{aligned} v^*(STATE, while\ e\ do\ S\ od) &:= \\ &\text{if } v^*(STATE, e)[2] =_{bool} true \text{ then} \\ &\quad v^*(STATE, seq(S, while\ e\ do\ S\ od)) \\ &\text{else} \\ &\quad [STATE, null] \end{aligned}$$

For simple imperative constructs, the abstract machine provides a kind of abstract syntax with formal semantics.

5.2 Definition of Dynamic Logic

Using logical formulae we can make statements about semantical structures. Modal logics assume structures with a notion of state [11, 18]. The presented computation model of Σ -objects defines such a structure. A satisfaction relation relates formulae and semantic structures. Modal logics are an extension of first-order predicate logic. With special modal constructors state transformations can be described. The presented computation model is therefore an ideal basis for a certain class of logics — the modal logics. Dynamic logic is special form in which program fragments are made explicit. See also [18], where transition systems are used as semantic structures for defining a modal logic.

Hoare-logic, a special dynamic logic, provides constructs for simple state transitions. In the formula

$$\phi \rightarrow [P] \psi$$

(normally written $\{\phi\}P\{\psi\}$ in Hoare-logic) ϕ is a precondition, P is a program fragment (i.e. a procedure call or a command) and ψ is the postcondition. ϕ and ψ are formulae describing properties of states. The formula $\phi \rightarrow [P] \psi$ holds in a state $STATE$ in a Σ -object A , if:

$$\phi \text{ holds in } STATE \wedge v^*(STATE, P)[1] \neq \perp \rightarrow \psi \text{ holds in } STATE'$$

where $STATE' = v^*(STATE, p)[1]$. The validity of the formula is expressed as a *partial correctness assertion*. Other predicate logic formulae may be defined as usual⁸.

⁸A sound and relatively complete proof system for a dynamic logic, which uses Σ -objects as semantic structures, can be found in [17].

5.3 Definition of Interaction

5.3.1 The π -Calculus

The following is an example of interaction formulated in the π -calculus (see [16, 15]).

$$ax.P[x]|\bar{a}V.Q \rightarrow P[V]|Q$$

It is the basic interaction rule in that calculus. P and Q are processes which communicate synchronously via a channel a (named $\bar{a}$ to indicate the inverse direction). The bar $|$ is used to separate the two processes. The value V (precisely a name denoting a value) is passed via the channel and bound to x .

Let us think of P and Q as two independent agents or processes, which are realized by Σ -objects (with possibly different signatures). Both objects must have ports via which the communication is done. Each port connects to a channel giving an indication about the intended use (*read* or *write*). If the channel is connected from both sides with opposite intentions, the communication (that is passing data) may take place.

The π -calculus is a formal calculus. It can be used to support a process language like CCS. It consists of action terms (communication actions), general terms and a computation rule. Action terms are $\bar{x}z.P$ (send z along x) and $xy.P$ (receive any y along x). A general term is e.g. $P_1 \mid P_2$ (parallel composition). The occurrences of y are binding. The basic rule of computation is presented above.

5.3.2 Integration of Interaction

Parallel composition can be defined via products. Parallel composition should be commutative and associative. These properties are preserved due to the product definition.

The available commands for Σ -objects are extended by new commands:

$$read(channel\ c, expr\ e) \text{ and } write(channel\ c, expr\ e)$$

The expression e has to be either a variable (*read*) or an expression denoting a value (*write*)⁹. *read* is a command, i.e. it modifies the state. The value read (denoted by V) is assigned to the variable given in the command. Thus, it can be defined by the assignment statement. Let A be a Σ -object in a state $STATE$.

$$read(a, x)(A) := if(SYNC, asgn(STATE, x, V))$$

write has no effect on the sending object itself.

$$write(a, e)(A) := skip$$

But both operation only have their synchronisation meaning, if in the system state a synchronisation occurs (see definition of product operator). This is expressed by the predicate *SYNC*. *SYNC* guards the execution of the value passing. A daemon is to be assumed that maintains a flag *SYNC* indicating that synchronisation may take place.

6 Conclusions

In the development of large systems we are often confronted with the necessity of describing different aspects of a system in different paradigms. For each aspect (evaluation, change, communication) we assume appropriate means to be available. We have shown that starting from a general state-based view the structure and the dynamics of a system can be specified by integrating different paradigms on the basis of objects as the central notion.

Formal language engineering deals with foundations for language definition and language integration. This approach is therefore a starting point which allows to define languages with different styles, e.g. an imperative programming language or a logic-based specification language. An integration of a process-oriented language in the given framework is also possible.

⁹In the π -calculus everything is a name. For the sake of simplification we take V here as a value.

The presented approach is intended to be used in the formalisation and extension in the context of VDM. This allows to give adequate semantics for the language, but also enabled languages extensions, e.g. by introducing process-oriented features or by extending the expressivity of the specification logic in direction of modal logics.

Formal language engineering requires a modular concepts of semantics. Starting with a basic package a number of extensions should be provided. These packages should be independent so that no unintended feature interactions occur. Possible extensions are provided in [17] comprising e.g. locality of objects or extensions of the type system.

As the presented approach is intended as a general model, different approaches have to be investigated. The π -calculus is a promising, very general model for describing the basic principle of interaction. Other description means for reactive systems, e.g. the event-driven statecharts [8], have to be taken into account to evaluate the given approach. Then, the aspect of communication can be dealt with in depth.

Acknowledgements

The author would like to thank Andrew Butterfield and Mícheál Mac an Airchinnigh for their helpful comments and suggestions.

References

- [1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.
- [2] D. Ancona, E. Astesiano, and E. Zucca. Towards a Classification of Inheritance Relations. In U. Lipceck and G. Koschorrek, editors, *IS-CORE'93 Workshop, Informatik-Berichte 01/93*, pages 90–113. Universität Hannover, 1993.
- [3] D. Ancona and E. Zucca. A Formal Framework for Modules with State. In M. Wirsing and M. Nivat, editors, *Proceedings 5th Int. Conf. on Algebraic Methodology and Software Technology, AMAST'96*. Springer-Verlag, LNCS 1101, 1996.
- [4] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.
- [5] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Modules and Constraints*, EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1990.
- [6] R. Groenboom and G.R. Renardel de Lavalette. Reasoning about Dynamic Features in Specification Languages – A Modal View on Creation and Modification. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *1st International Workshop on Semantics of Specification Languages*, Utrecht, 1993, pages 340–356. Springer-Verlag, 1994.
- [7] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide*. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.
- [8] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5), 1988.
- [9] M. Hennessy. Acceptance Trees. *Journal of the ACM*, 32(4):896–928, October 1985.
- [10] C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [11] 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.
- [12] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley-Teubner, 1984.
- [13] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, LNCS 92, 1980.

- [14] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [15] R. Milner. *Elements of Interaction*. *Communications of the ACM*, 36:78–89, 1993.
- [16] R. Milner, J. Parrow, and D. Walker. *A calculus of Mobile Processes, part I*. *Information and Computation*, 100(1):1–40, 1992.
- [17] C. Pahl. *Foundations for Formal Specification of State-based Systems (in German)*. PhD thesis, Universität Dortmund, Germany, 1996. <ftp://ls10-www.informatik.uni-dortmund.de/pub/Dissertationen>, also published as SWT-Memo No. 89, Lehrstuhl Software-Technologie, University of Dortmund, Germany, September 1996.
- [18] 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.
- [19] R.J. van Glabbeek. *The Linear Time – Branching Time Spectrum*. In J.C.M. Baeten and J.W. Klos, editors, *Proceedings CONCUR’90*, pages 278–297. Springer-Verlag, LNCS 458, 1990.
- [20] G. Winskel and M. Nielsen. *Models for Concurrency*. In S. Abramsky, D.M. Gabbay, and T.S.E Maibaum, editors, *Handbook of Logics in Computer Science Vol. IV (Semantic Modelling)*, pages 1–148. Clarendon Press, Oxford, 1995.
- [21] M. Wirsing. *Algebraic Specification*. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pages 675–788. Elsevier Science Publishers, 1990.