E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors,
Algebraic Foundations of System Specification, chapter 12,
pages 435-465. Springer-Verlag, Berlin 1999

12 Object Specification*

Hans-Dieter Ehrich

Abteilung Datenbanken, Technische Universität,
Postfach 3329, D-38023 Braunschweig, Germany
HD.Ehrich@tu-bs.de

12.1 Introduction

From an object-oriented point of view, software systems are considered to be dynamic collections of autonomous objects that interact with each other. Autonomy means that each object encapsulates all features needed to act as an independent computing agent: individual attributes (data), methods (operations), behavior (process), and communication facilities. And each object has a unique identity that is immutable throughout lifetime. Coincidentally, object-orientation comes with an elaborate system of classes and types, facilitating structuring and reuse of software.

The object approach is widely accepted in software technology, and there are object-oriented programming languages, database systems, and software development methods. The basic idea is not new, because essential features were already present in the programming language Simula [DMN67]. Wider acceptance came with Smalltalk [KG76,GR83].

While the object approach is successful in practice, it finds more scepticism than enthusiasm among theoreticians. Object-orientation is often considered an ugly area that lacks conceptual coherence. It is true that much of the work in the field cannot serve as counter-evidence. But matters are changing: there is growing interest in clean concepts and reliable foundations. In fact, object-orientation badly needs theoretical underpinning, for improving practice and facilitating teaching. Object theory is evolving into an own area of study.

Our approach to object specification combines ideas from algebraic data type specification, conceptual data modeling, behavior modeling, specification of reactive systems, and concurrency theory. Unlike conventional specification approaches, object specification shows its real virtue when dealing with open, reactive and distributed systems.

We concentrate on fundamental concepts and constructions rather than languages and systems. Our approach is based on experiences with developing the Oblog family of languages and their semantic foundations [SSE87]. Oblog is being developed into a commercial product [ESD93]. In the academic realm, there are two related developments: Troll [JSHS96,SJH93,

* This work was partly supported by the Eu under Esprit-III Bra Wg 6112 Compass, Esprit-III Bra Wg 3023 Iscore and Esprit-IV Wg 22704 Aspire.

HJ95,EH96,DH97] and GNOME [SR94]. The common semantic basis is linear-time temporal logic.

It is not the purpose of this chapter to give an overview of object specification languages, and not of their underlying models and theories either. Models, theories and languages are diverse enough to make a systematic overview, let alone a uniform treatment, impossible within the limits of this chapter. Rather, the author gives his own view and presentation that is based on experiences with developing TROLL, and on joint work with Amílcar and Cristina Sernadas and others on foundations and concepts [SSE87,SFSE88, EGS90,ES90,SJE92,EDS93,EJDS94,SHJE94]. Hints to related work are given in Section 12.7.

The chapter is organized as follows. In the next section, we briefly explain basic concepts: objects, classes, types, and systems. In Section 12.3, we describe a denotational model of object classes that is tailored towards temporal logic. Section 12.4 explains temporal class specification, i.e., the use of linear-time temporal logic for in-the-small specification of object classes. Section 12.5 discusses in-the-large structuring concepts that are typical for object-orientation: inheritance and composition. Inheritance concepts include is-a and as-a specialization, hiding, overriding, overlaying, and covariance. Composition concepts include generalization and aggregation. For specifying interaction among object components, temporal logic is sufficient. For interaction among concurrent objects, an extension of the logic is necessary. We briefly explain our approach using Distributed Temporal Logic DTL.

12.2 Basic concepts

12.2.1 Objects

An object is an encapsulated unit of structure and behavior. It has an identity which persists through change. Objects communicate with each other. They are typed, related by inheritance, and composed to form complex objects.

A widely accepted informal object model has been formulated by Wegner [Weg89]: *An object has a set of operations and a local shared state (data) that remembers the effect of operations. The value that an operation on an object returns can depend on the object's state as well as the operation's arguments. The state of an object serves as a local memory that is shared by operations on it. In particular, other previously executed operations can affect the value that a given operation returns. An object can learn from experience, storing the cumulative effect of its experience – its invocation history – in its state.*

The operations of an object are usually called *methods*. In the object model of object-oriented programming, a method may change state and deliver a value. This model also underlies object specification languages like FOOPS and ETOILE.

The object model of object-oriented data bases is more restricted; it separates state-changing "proper" methods from side-effect free "read" methods

called *attributes*. This model also underlies TROLL. For the sake of simplicity, we adopt this model here.

12.2.2 Classes

An object *class* represents the prototypical structure and behavior of objects. We illustrate the concept by means of two examples, state variables and banks. For class specifications, we use an ad-hoc notation that is closer to the formal model adopted in this chapter than to any of the specification languages mentioned above. Using linear-time temporal logic, however, the specification style is closest in spirit to TROLL.

Example 1. There is a kind of object that is so basic and omnipresent in computing that it has many names: *state variable* in programming, *attribute* in databases and conceptual modeling, *field* in file organization, *slot* or *fluent* in artificial intelligence, and *memory cell*, *register*, *word*, *byte*, *bit* and *flip-flop* in hardware.

Here is the formal specification of a class `Var[`s`]` of state variables, or variables for short, of sort s. A variable x has an attribute x.`val` of sort s denoting its current value. It has actions x`.c` for creation, x`:=`s for assigning values (i.e., an action x`:=v` for every value `v` of sort s), and x`.d` for deletion. In the axioms, we use the *until* temporal operator $\varphi\ \mathcal{U}^\circ\ \psi$ for expressing that φ holds from now on until ψ holds for the next time (cf. Section 12.4 for a precise definition). We use the notation $\triangleright a$ for expressing that action a is enabled, i.e. may occur at the current state, and $\odot a$ for expressing that action a has just occurred.

class `Var[`s`];`
  uses s`;`
  var x`:Var[`s`];`
  attributes x`.val:`s`;`
  actions x`.c;` x`:=`s`;` x`.d;`
  axioms var `v,w:` s`;`
    $\odot x$`.c` $\Rightarrow$ $(\neg \triangleright x$`.c` $\wedge$ $\triangleright x$`:=v` $\wedge$ $\triangleright x$`.d` $)$ $\mathcal{U}^\circ \odot x$`.d,`
    $\odot x$`.d` $\Rightarrow$ $\neg \triangleright x$`.c` $\wedge$ $\neg \triangleright x$`:=v` $\wedge$ $\neg \triangleright x$`.d,`
    $\odot x$`:=v` $\Rightarrow$ x`.val=v` $\mathcal{U}^\circ$ $(\odot x$`:=w` $\vee \odot x$`.d` $)$
end

The axioms say that (1) after creation and before deletion, another creation is disabled but value assignment and deletion are enabled; (2) after deletion, no action is enabled; (3) the value after assignment is retained until the next assignment or deletion.

Notation. In object-oriented notation, it is common practice to omit the local object variable x. We follow this practice in the sequel. The example then reads as follows.

```
class Var[s];
   uses s;
   attributes val:s;
   actions c; :=s; d;
   axioms var v,w: s;
      ⊙c ⇒ (¬▷c ∧ ▷:=v ∧ ▷d ) 𝒰° ⊙d,
      ⊙d ⇒ ¬▷c ∧ ¬▷:=v ∧ ¬▷d,
      ⊙:=v ⇒ val=v 𝒰° (⊙:=w ∨⊙d)
end
```

■

Example 2. The objects in this example are banks and accounts.

An account has a holder and a balance as attributes, and it has actions to open and close an account and to credit and debit money. In the formal specification, we use the temporal operators Y for *previous* (yesterday) and P for *sometime in the past*. The axioms say that (1) after an account is opened, it has the holder and balance given as actual parameters; (2) crediting increases the balance by the amount credited; (3) debiting decreases the balance by the amount debited; (4) an amount m may only be debited if the account once had at least balance m (a simple solvency criterion).

```
class Account;
   uses money, Person;
   attributes holder: Person, balance: money;
   actions   *open(Person,money), credit(money),
                debit(money), +close;
   axioms var p:Person, m,n:money;
      ⊙open(p,m) ⇒ holder=p ∧ balance=m,
      ⊙credit(m) ⇒ ⊙balance:=balance+m,
      ⊙debit(m) ⇒ ⊙balance:=balance-m,
      ⊙debit(m) ⇒ P balance≥m
end
```

* denotes a birth action that may only occur at the beginning, and + denotes a death action that terminates the life of an object. After a birth action, no birth action is enabled, and after a death action, no action is enabled. This may be specified explicitly but we refrain from doing so.

The second and third axioms are abbreviations for

⊙credit(m) ∧ Y balance=n ⇒ balance=n+m and
⊙debit(m) ∧ Y balance=n ⇒ balance=n-m, respectively.

As usual, a frame rule is assumed saying that attributes do not change values unless specified otherwise. Among the consequences of the axioms and this frame rule we have, e.g.,

⊙open(p,m) ⇒ G holder=p.

G is the *always* temporal operator.

A bank is a complex object. It has a collection of accounts as components, providing access to their individual attributes and actions. The data type `acct#` gives the set of permissible account numbers. A bank has an owner as an attribute, and actions for establishing a bank, for transferring money from one account to another, and for liquidating a bank. In the following specification, we use the temporal operators X for *next* (tomorrow), $\mathcal{P}^+$ for *precedes in the future*, and $\mathcal{U}^\circ$ for *until* that was introduced above. The axioms say that (1) in the beginning, the owner is the person who established the bank; (2) transferred amounts are debited next after transfer; (3) transferred amounts are eventually credited; however, as long as a transfer is not credited, the same amount may not be transferred from the same account to the same account again; (4) there must not be two different transfers of the same amount from the same account at the same time; (5) transferred amounts are credited before any other amount is debited from the same account.

The third and forth axioms are a simple way to ensure correct transaction management.

class `Bank;`
  uses `money, acct#, Person, Account;`
  attributes `owner: Person;`
  components `Acc(acct#):Account;`
  actions `*establish(Person),`
    `transfer(Account,Account,money),`
    `+liquidate;`
  axioms *var* `p: Person, from, to, to2: acct#, m, n: money;`
    $\odot$`establish(p)` $\Rightarrow$ `owner=p,`
    $\odot$`transfer(from,to,m)` $\Rightarrow$ $\mathsf{X}\,\odot$`Acc(from).debit(m),`
    $\odot$`transfer(from,to,m)` $\Rightarrow$
      $\neg\,\odot$`transfer(from,to,m)` $\mathcal{U}^\circ$ $\odot$`Acc(to).credit(m),`
    $\odot$`transfer(from,to,m)` $\wedge$ $\odot$`transfer(from,to2,m)` $\Rightarrow$ `to=to2,`
    $\odot$`transfer(from,to,m)` $\Rightarrow$
      $\odot$`Acc(to).credit(m)` $\mathcal{P}^+\odot$`Acc(to).debit(n)`
end

The following liveness property is a consequence of the third axiom.

$\odot$`transfer(from,to,m)` $\Rightarrow$ $\mathsf{F}\,\odot$`Acc(to).debit(n ).`

F is the *sometime in the future* temporal operator. The property says that transfers are eventually credited.

So far, the example is unpractical because it does not describe transfers between different banks. One solution is to introduce a complex object `BankWorld` having all banks as components. Transfers between banks may then be specified in the `BankWorld` class, in the same way as transfers between accounts are specified above. A more appropriate solution, however, uses DTL, an extension of temporal logic towards describing concurrent behavior (cf. Example 14). ∎

For formal treatment, we abstract from irrelevant details and put the relevant ones into a handy shape. Let $\Sigma = (S, \Omega)$ be a data signature where S is a set of sorts, and Ω is an $S^* \times S$-indexed set family of operation symbols. We assume that $\texttt{bool} \in S$. If $\omega \in \Omega_{s_1 \ldots s_n, s}$, we write $\omega : s_1 \times \ldots \times s_n \to s \in \Omega$ or briefly $\omega : x \to s$ if $x = s_1 \ldots s_n$ and the reference to Ω is clear from context. Let U be a Σ-algebra.

The syntax of a class specification is given by its attribute and action symbols. For ease of presentation, we assume that these symbols are not parameterized. Parameterized symbols will occur in practice but may be expanded into their instantiations in U by substituting actual parameters in all possible ways.

For the sake of formal simplicity, we treat action symbols as attribute symbols with sort `act`, to be interpreted by value set $\{\triangleright, \odot, \triangleright\odot, \varepsilon\}$. These values indicate that an action is enabled ($\triangleright$) or has occurred ($\odot$) or both ($\triangleright\odot$) or none of these (ε). This way, class states are uniformly characterized by a family of sets of attributes. We write $\triangleright\, a$, $\odot\, a$, $\triangleright\odot\, a$, or εa, respectively, if action a is in the corresponding state. Further action states may be introduced, for instance expressing scoping for dealing with dynamic roles and phases [EJD93].

Definition 1. Let Σ be a data signature. A *class signature* over Σ is an S-indexed set family $\Gamma = \{\Gamma_s\}_{s \in S}$ of attribute symbols.

The intended interpretation of a class signature Γ is a *Γ-class* C (cf. Definition 17) describing the generic structure and behavior of object instances (cf. Subsection 12.2.3). If there is no danger of confusion, we omit the Γ-prefix. We adopt the view that *objects are sequential processes* and *systems are concurrent collections of objects*. Our denotational models for classes and systems are based on labelled event structures, cf. Section 12.3 below.

An in-the small class specification is a pair $Cspec = (\Gamma, \Phi)$ where Γ is a class signature and Φ is a set of class formulae, cf. Definition 21. Different logics may be applied. Our choices, linear-time temporal logic and an extension towards distribution, are elaborated in Sections 12.4 and 12.6.

For dealing with in-the-large specification, morphisms between class signatures and specifications are essential in order to capture relevant constructions like inheritance, hiding, generalization, aggregation, etc. on the syntactic level. The corresponding semantic relationships are captured by class morphisms, cf. Definition 18.

Definition 2. Let Γ_1 and Γ_2 be class signatures over the same data signature. A *class signature morphism* $\gamma\colon \Gamma_1 \to \Gamma_2$ is an S-indexed family of total functions.

For class specifications $Cspec_1 = (\Gamma_1, \Phi_1)$ and $Cspec_2 = (\Gamma_2, \Phi_2)$, a *class specification morphism* $\gamma\colon Cspec_1 \to Cspec_2$ is a class signature morphism $\gamma\colon \Gamma_1 \to \Gamma_2$ preserving properties in the sense that $\Phi_2 \models \gamma(\Phi_1)$ holds.

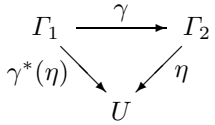
As usual, $\gamma(\Phi_1)$ is obtained by translating formulae syntactically via γ, replacing every attribute symbol by its γ image. The class specification morphism condition says that the translated axioms $\gamma(\Phi_1)$ must be entailed by Φ_2. Satisfaction and entailment for our version of temporal logic are given in Definition 20.

Remark 1. The category **csig** of class signatures and class signature morphisms is complete and cocomplete. Colimits describe syntactic composition of signatures and specifications.

Given a Σ-algebra U, a class signature determines the possible states of a class: a state is a value assignment to attributes.

Definition 3. Let Γ be a class signature. A *class state* over U is an S-indexed set family of total functions $\eta\colon \Gamma \to U$. The set of class states over Γ and U is denoted by $[\Gamma \to U]$, or $[\Gamma]$ for short if U is clear from context.

Let $\gamma\colon \Gamma_1 \to \Gamma_2$ be a class signature morphism. The *class state morphism* $\gamma^*\colon [\Gamma_2] \to [\Gamma_1]$ defined by γ is given by $\gamma^*(\eta) = \eta \circ \gamma$ for each $\eta \in [\Gamma_2]$.



For an action symbol a, we write $\odot[\eta]a$ for $\eta(a) \in \{\odot, \odot\triangleright\}$, $\triangleright[\eta]a$ for $\eta(a) \in \{\triangleright, \odot\triangleright\}$, $\triangleright\odot[\eta]a$ for $\eta(a) = \odot\triangleright$, and $\varepsilon[\eta]a$ for $\eta(a) = \varepsilon$. Note that $\triangleright\odot[\eta]a$ iff $\triangleright[\eta]a \ \wedge \ \odot[\eta]a$, and $\varepsilon[\eta]a$ iff $\neg(\triangleright[\eta]a \ \vee \odot[\eta]a)$.

12.2.3 Types

An object type is a static domain of object instances of the same class. It consists of a set of object *identities* for representing the former, and a class. Object type morphisms (direct and reverse) describe various kinds of object relationship: inheritance, hiding, generalization, and aggregation.

Definition 4. An *object type signature* is a pair $\Theta = (Id, \Gamma)$ where Id is a set of *object identities* and Γ is a class signature.

The *object instance signature* Υ determined by Θ provides an individual *object signature* $i.\Gamma$ for each object with identity i. If $a \in \Gamma$, then its instance for object i is denoted by $i.a$. Technically, Υ is the coproduct $\coprod_{i \in Id} \Gamma$ in **csig**.

Object type signatures may be related by morphisms mapping identities and relating class signatures by class signature morphisms. We refrain from giving a formal definition because the concept is not needed in this chapter. However, it is implicit in object type morphisms, see below. A corresponding remark applies to object instance signature morphisms.

Definition 5. An *object type* with signature $\Theta = (Id, \Gamma)$ is a pair $T = (Id, C)$ where C is a Γ-class.

An *instance of* T is a pair $i.C$ where $i \in Id$ is an identity. The object instances of T are given by $\overline{T} = Id.C = \{i.C \mid i \in Id\}$. Each object instance $i.C$ is an individual process that is an isomorphic copy of class C with signature $i.\Gamma$.

Definition 6. Let $T_1 = (Id_1, C_1)$ and $T_2 = (Id_2, C_2)$ be object types. An *object type morphism* $\tau\colon T_1 \rightrightarrows T_2$ consists of a map $\tau'\colon Id_1 \to Id_2$ and a class morphism $\tau''\colon C_1 \to C_2$.

Class morphisms are given in Definition 18. A class morphism $\tau''\colon C_1 \to C_2$ has an underlying class signature morphism $\gamma\colon \Gamma_2 \to \Gamma_1$ going in opposite direction. The corresponding class state morphism $\gamma^*\colon [\Gamma_1] \to [\Gamma_2]$ maps C_1 states into C_2 states.

For identities $i \in Id_1$ and $j \in Id_2$, we may define an *object instance morphism* $i\text{-}j.\tau\colon i.C_1 \to j.C_2$ as the corresponding isomorphic copy of τ.

We give examples of relationships typically expressed as object type morphisms.

Example 3. The fact that each customer *is-a* person is expressed by an object type morphism $\tau\colon \mathit{Customer} \rightrightarrows \mathit{Person}$ such that $\tau'\colon Id_{\texttt{Customer}} \hookrightarrow Id_{\texttt{Person}}$ is an inclusion. The class morphism $\tau''\colon C_{\texttt{Customer}} \to C_{\texttt{Person}}$ restricts the customer class to its underlying person class, cf. Example 8 for a more detailed treatment. The underlying class signature morphism $\gamma\colon \Gamma_{\texttt{Person}} \to \Gamma_{\texttt{Customer}}$ embeds person attributes into those of the customer class. Conversely, the class state morphism $\gamma^*\colon [\Gamma_{\texttt{Customer}}] \to [\Gamma_{\texttt{Person}}]$ reduces customer states to person states, forgetting about the additional customer attributes.

The example reflects temporal specialization in the sense that persons may temporarily and repeatedly play the role of customers. ∎

Example 4. The specification $\mathit{Customer} = \mathit{Person} + \mathit{Company}$ says that each *Customer* is either a *Person* or a *Company*. This is *generalization* or alternative choice. The customer signature contains the attributes (including actions) that are common to both persons and companies and that are relevant to their roles as customers. Fixing these is a design decision. For the sake of simplicity, we assume that the *Person* and *Company* classes are already restricted to these roles, i.e., we assume $\Gamma_{\texttt{Customer}} = \Gamma_{\texttt{Person}} = \Gamma_{\texttt{Company}}$.

We have object type morphisms $\tau_1\colon \mathit{Person} \rightrightarrows \mathit{Customer}$ and $\tau_2\colon \mathit{Company} \rightrightarrows \mathit{Customer}$, embedding the constituent types into the generalization (cf. Subsections 12.3.3 and 12.5.2). In fact, + denotes coproduct in the category of object types and object type morphisms over a fixed class signature.

Note that each person *is-a* customer here, the other way round as in the previous example. So we have identity inclusions $\tau_1'\colon Id_{\texttt{Person}} \hookrightarrow Id_{\texttt{Customer}}$ and $\tau_2'\colon Id_{\texttt{Company}} \hookrightarrow Id_{\texttt{Customer}}$ as well as class morphisms $\tau_1''\colon C_{\texttt{Person}} \to C_{\texttt{Customer}}$ and $\tau_2''\colon C_{\texttt{Company}} \to C_{\texttt{Customer}}$. The class signature morphisms are identities under the assumption mentioned above, and so are the class state morphisms. ∎

Example 5. Components in the sense of Example 2 are meant to be interpreted by *object aggregation*: complex objects are built from constituent objects and synchronize the behaviors of these components. For example, we may specify $Car = Motor \bowtie Chassis$ saying that each *Car* object is a complex object consisting of a *Motor* object and a *Chassis* object. The car signature contains all motor and chassis attributes and distinguishes between them, i.e., $\Gamma_{\texttt{Car}} = \Gamma_{\texttt{Motor}} + \Gamma_{\texttt{Chassis}}$ is the disjoint union with inclusions $\gamma_1 \colon \Gamma_{\texttt{Motor}} \hookrightarrow \Gamma_{\texttt{Car}}$ and $\gamma_2 \colon \Gamma_{\texttt{Chassis}} \hookrightarrow \Gamma_{\texttt{Car}}$.

We have object type morphisms $\tau_1 \colon Car \rightrightarrows Motor$ and $\tau_2 \colon Car \rightrightarrows Chassis$ that give the projections to the components. In fact, $\bowtie$ denotes product in the category of object types and object type morphisms. $id_{\texttt{Car}} = id_{\texttt{Motor}} \times id_{\texttt{Chassis}}$ is the cartesian product with projections $\tau_1' \colon id_{Car} \to id_{Motor}$ and $\tau_2' \colon id_{Car} \to id_{Chassis}$. Class morphisms $\tau_1'' \colon C_{\texttt{Car}} \to C_{\texttt{Motor}}$ and $\tau_2'' \colon C_{\texttt{Car}} \to C_{\texttt{Chassis}}$ are projections of the product in the category of classes, cf. Subsection 17. ∎

There are relationships between object types that are more naturally expressed by a variant of object type morphism where the class morphism goes the other way. We give examples after the formal definition.

Definition 7. Let $T_1 = (Id_1, C_1)$ and $T_2 = (Id_2, C_2)$ be object types. A *reverse object type morphism* $\tau \colon T_1 \rightleftarrows T_2$ consists of a map $\tau' \colon Id_1 \to Id_2$ and a class morphism $\tau'' \colon C_2 \to C_1$.

The class morphism $\tau'' \colon C_2 \to C_1$ has an underlying class signature morphism $\gamma \colon \Gamma_1 \to \Gamma_2$. Thus, the map between identities goes the same way as the class signature morphism whereas the class state morphisms conform with the class morphisms.

For identities $i \in Id_1$ and $j \in Id_2$, the *reverse object instance morphism* $i\text{-}j.\tau \colon i.C_2 \to j.C_1$ is the corresponding isomorphic copy of τ.

We give examples for typical relationships expressed as reverse object type morphisms.

Example 6. The fact that a constrained kind of rectangle may be viewed *as-a* square is expressed by a reverse object type morphism $\tau \colon Square \rightleftarrows Rectangle$. Again, each square *is-a* rectangle, so τ' is an inclusion. The class signature morphism embeds square attributes into those of the rectangle class, $\gamma \colon \Gamma_{\texttt{Square}} \to \Gamma_{\texttt{Rectangle}}$. The class state morphism goes the other way, $\gamma^* \colon [\Gamma_{\texttt{Rectangle}}] \to [\Gamma_{\texttt{Square}}]$, reducing rectangle states to square states and forgetting about the additional rectangle attributes. The class morphism goes the same way, $\tau'' \colon C_{\texttt{Rectangle}} \to C_{\texttt{Square}}$, projecting rectangle behavior to square behavior. Cf. Example 9 for a more detailed treatment. ∎

Like is-a inheritance, as-a inheritance is intended to express a kind of temporal specialization. The difference is that the former describes that *all* features are inherited and new ones may be added, whereas as-a inheritance describes that *some* features are inherited and possibly used in a special way.

Example 7. The hiding relationship between databases and their views is expressed by a reverse object type morphism $\tau\colon \mathit{View} \rightleftarrows \mathit{Database}$. Databases and their views are different objects, so we have $Id_{\texttt{View}} \cap Id_{\texttt{Database}} = \emptyset$. The class signature morphism maps view attributes into those of the database class. The class morphism in turn projects database behavior onto view behavior, and the class state morphism reduces database states to view states. ∎

Remark 2. Given a category **class** of classes and class morphisms, object types and object type morphisms form a category **ot(class)** that inherits limits and colimits from **class**. Comments on **class** are given at the end of Section 12.3.

12.2.4 Systems

An object system is a collection of object instances operating concurrently and interacting by exchanging messages. Our interaction model is synchronous and symmetric "handshake".

Definition 8. An *object system signature* is a pair $\Sigma = (S, \Omega)$ where $S = S^d + S^o$ is a set of *sorts*, subdivided into disjoint sets of *data sorts* S^d and *object sorts* S^o. $\Omega = \{\Omega_{x,s}\}_{x \in S^*, s \in S}$ is an $S^* \times S$-indexed set family of *operation symbols*.

A data signature is the special case where the set S^o of object sorts is empty, so we use the same notation. Like for data signatures, interpretation is given in a Σ-algebra U. The intended interpretation of a data sort $d \in S^d$ in U is a set U_d of data elements that is the carrier set of d in U. The data type of sort d is defined by this set and the operations defined on it.

The intended interpretation of an object sort $s \in S^o$ in U is a set $U_s = Id_s.C_s$ of object instances $i.C_s$ with signatures $i.\Gamma_s$, $i \in Id_s$. The class C_s describes the common structure and behavior of its instances. The object type of sort s is defined by the set U_s of instances and the operations defined on it. For simplicity, we ignore object operations in the sequel, i.e., we consider object types to be sets of instances with the same behavior, as introduced in the previous section.

The intended interpretation of an object system signature is the concurrent composition of its instances, cf. Section 12.6.

In a sense, a data type is an object type with a trivial class only allowing for static "behavior". On the other hand, the identity parts Id_s of object types are like data types; they may be specified with any of the data specification techniques.

Definition 9. The *object instance signature* Υ *over* Σ *and* U is given by $\Upsilon = (\boldsymbol{Id}, \boldsymbol{\Gamma})$ where $\boldsymbol{Id} = \{Id_s\}_{s \in S^o}$ is the set family of object identities, and $\boldsymbol{\Gamma} = \{i.\Gamma_s\}_{i \in Id_s, s \in S^o}$ is the set family of individual object signatures.

Definition 10. An *object system specification* is a pair $Sspec = (\Upsilon, \Phi)$ where Υ is an object instance signature, and $\Phi = \{\Phi_i\}_{i \in Id_s, s \in S^o}$ is the set family of object axioms.

There is a problem with using temporal logic for object axioms. For specifying single classes, temporal logic is sufficient. For specifying concurrent object systems, however, temporal logic is not expressive enough: we extend it towards distributed temporal logic in Subsection 12.6.2.

12.3 A denotational class model

In order to model the sequential behavior of objects and the concurrent behavior of object systems, many models of concurrency may be adopted and integrated in our framework. The question is which model to adopt. According to the classification in [SNW93], we may choose along three dimensions: (1) a behavior or a system model, (2) an interleaving or a noninterleaving model, and (3) a linear or a branching time model.

For denotational semantics, a behavior model is more appropriate than a system model because it is more abstract. For linear-time temporal logic, an interleaving model is sufficient because concurrency cannot be expressed. However, we need a noninterleaving model in order to capture concurrency. According to [SNW93], *labelled event structures* are a fair choice when a noninterleaving denotational model is needed.

We follow this advice also for the following reasons. The relationship between event structures and other models of concurrency is well investigated, and the interpretation structures of temporal logic, linear-time as well as branching-time, can be embedded into event structures, leaving room for extending the logic while staying in the interpretation domain. Cf. Section 12.7 for references to related work.

12.3.1 Event structures

An elegant aspect of event structures is that they capture the behavior of systems, single system runs, objects and single object runs in one and the same model. We review the basic definitions of prime event structures, concentrating on the case where causality is the reflexive and transitive closure of a base relation of "step causality". We use the notation $\rightarrow$ for the latter, and $\rightarrow^*$ for causality[1]. $\rightarrow^+$ denotes the irreflexive transitive closure of $\rightarrow$.

Definition 11. A *prime event structure* – or *event structure* for short – is a triple $E = (Ev, \rightarrow^*, \#)$ where Ev is a set of events, and $\rightarrow^*, \# \subseteq Ev \times Ev$ are binary relations called *causality* and *conflict*, respectively. Causality $\rightarrow^*$ is a

[1] The standard notation for causality is $\leq$ but this symbol is also standard for subsorting expressing inheritance (cf. Subsection 12.5.1). We use it for the latter.

partial ordering, and conflict $\#$ is symmetric and irreflexive. For each event $e \in E$, its *local configuration* $\downarrow e = \{e' \mid e' \rightarrow^* e\}$ is finite. Conflict propagates over causality, i.e., $e\#e' \rightarrow^* e'' \Rightarrow e\#e''$ for all $e, e', e'' \in Ev$.

Two events $e, e' \in Ev$ are *causally related*, $e \sim e'$, iff $(e \rightarrow^* e' \vee e' \rightarrow^* e)$.

Two events $e, e' \in Ev$ are *concurrent*, $e \,co\, e'$, iff $\neg\, e \sim e' \wedge \neg\, e\#e'$.

A *configuration* in E is a downward-closed conflict-free set of events $C \subseteq Ev$, i.e., the following conditions hold: (1) $\forall e \in C : \downarrow e \subseteq C$, and (2) $\forall e_1, e_2 \in C : \neg(e_1\#e_2)$. The set of configurations in E is denoted by $\mathcal{C}(E)$.

A *life cycle* in E is a maximal configuration in E. The set of life cycles in E is denoted by $\mathcal{L}(E)$.

In the sequel, order-theoretic notions refer to causality. The finiteness condition for local configurations is a temporal reachability condition: only events $e \in Ev$ that may happen within finite time after system start are taken into consideration. Obviously, we have $\mathcal{L}(E) \subseteq \mathcal{C}(E)$ for every event structure E. Intuitively, a life cycle represents a possible run of a system, and a configuration represents a (not necessarily proper) prefix of it. This represents a state of a part of the system. Please note that life cycles may be finite or infinite, so the behaviors of terminating programs and of nonterminating reactive systems can both be modelled adequately.

Proposition 1. *For event structures E_1 and E_2, we have $E_1 = E_2$ iff $\mathcal{C}(E_1) = \mathcal{C}(E_2)$ iff $\mathcal{L}(E_1) = \mathcal{L}(E_2)$.*

Definition 12. For $i = 1, 2$, let $E_i = (Ev_i, \rightarrow_i^*, \#_i)$ be event structures. An *event structure morphism* $g\colon E_1 \rightarrow E_2$ is a partial function $g\colon Ev_1 \rightharpoonup Ev_2$ such that, for every configuration $C \in \mathcal{C}(E_1)$, the following conditions hold: (1) $g(C) \in \mathcal{C}(E_2)$, and (2) $\forall e_1, e_2 \in C \cap dom(g) :\ g(e_1) = g(e_2) \Rightarrow e_1 = e_2$.

Event structure morphisms preserve, and are injective on configurations. An event structure morphism $g\colon E_1 \rightarrow E_2$ is often used to express how behavior in E_1 synchronizes with behavior in E_2: the occurrence of event e_1 in $dom(g)$ implies the simultaneous occurrence of event $e_2 = g(e_1)$ in E_2 and vice versa.

The following lemma gives a characterization of event structure morphisms that uses only local configurations [WN95].

Lemma 1. *For $i = 1, 2$, let $E_i = (Ev_i, \rightarrow_i^*, \#_i)$ be event structures. Let $g\colon Ev_1 \rightharpoonup Ev_2$ be a partial function. Then g is an event structure morphism iff the following conditions hold for all $e, e_1, e_2 \in dom(g)$: (1) $\downarrow g(e) \subseteq g(\downarrow e)$, and (2) $(g(e_1) = g(e_2) \vee g(e_1)\#_2 g(e_2)) \Rightarrow (e_1 = e_2 \vee e_1\#_1 e_2)$.*

Remark 3. The category **ev** of event structures is complete and has coproducts. Products and coproducts are useful for modeling concurrent composition and generalization, respectively [WN95]. Pullbacks may be utilized for modeling composition with sharing, e.g. event sharing for handshake communication.

12.3.2 Event groves

In this subsection, we concentrate on *sequential* event structures because they are appropriate models for classes: they provide an interpretation domain for class specifications using temporal logic.

Since conflict is a derived concept in sequential event structures, we introduce a simpler model that presents precisely the sequential event structures.

Definition 13. An *event grove* is an acyclic graph $G = (Ev, \rightarrow)$ such that, for all events $e_1, e_2 \in Ev$, we have $e_1 \sim e_2$ if there is an event $e_3 \in Ev$ for which $e_1 \rightarrow^* e_3$ and $e_2 \rightarrow^* e_3$ holds.

Remember that $e_1 \sim e_2$ means that $e_1 \rightarrow^* e_2$ or $e_2 \rightarrow^* e_1$. The definition means that, if both events e_1 and e_2 are causal for some future event, then they are causally related. Thus, an event grove is a set of rooted trees. An event grove $G = (Ev, \rightarrow)$ determines a prime event structure in a canonical way by letting all causally unrelated events be in conflict.

Definition 14. Let $G = (Ev, \rightarrow)$ be an event grove. The *event structure presented by* G is $E(G) = (Ev, \rightarrow^*, \#)$ where, for all $e_1, e_2 \in Ev$, $e_1 \# e_2$ holds iff $\neg\, e_1 \sim e_2$.

It is obvious that this defines a valid event structure. The configurations and life cycles are totally ordered, they are linear traces of events. If G is an event grove, we write $\mathcal{C}(G)$ for $\mathcal{C}(E(G))$ and $\mathcal{L}(G)$ for $\mathcal{L}(E(G))$.

The concurrency relation *co* of an event grove is empty. Event structures without concurrency have been called sequential in [LT87]; they are precisely those that can be presented by event groves. Sequential event structures are the event structures of synchronization trees [Win84] that are an appropriate behavior model of interleaving concurrency. They are elementary in the sense of [NPW81].

Sequential event structures allow for global timing of events along their traces: each initial event is timed 0, and the times of successor events are increased by 1.

Definition 15. Let $G = (Ev, \rightarrow)$ be an event grove. The *timing function* $\tau_G \colon Ev \rightarrow \mathbb{N}$ of G is defined by $\tau_G(e) = 0$ iff e is minimal, and $\tau_G(e') = \tau_G(e) + 1$ iff $e \rightarrow e'$ holds.

Definition 16. Let $G_1 = (Ev_1, \rightarrow_1)$ and $G_2 = (Ev_2, \rightarrow_2)$ be event groves. An *event grove morphism* $g \colon G_1 \rightarrow G_2$ is a graph morphism preserving time, i.e., a total map $g \colon Ev_1 \rightarrow Ev_2$ such that, for all events $e, e' \in Ev_1$, we have $e \rightarrow_1 e'$ implies $g(e) \rightarrow_2 g(e')$, and $\tau_{G_1}(e) = \tau_{G_2}(g(e))$.

A morphism between event groves coincides with a total morphism between the event structures presented by them, i.e., a map $g \colon Ev_1 \rightarrow Ev_2$ is an event grove morphism $g \colon G_1 \rightarrow G_2$ iff it is a total event structure morphism

$g\colon E(G_1) \to E(G_2)$. The rationale for restricting to total morphisms is that event groves related by morphisms should be fully synchronized: traces are mapped in a bijective way. Different traces, however, may be mapped to the same trace.

Remark 4. The category **egr** of event groves with their morphisms is complete and cocomplete. Products in **egr** do not coincide with those in **ev**; while the latter model concurrent object composition, the former model sequential object aggregation. Pullbacks may also be utilized for modeling aggregation with sharing. Coproducts coincide with those in **ev**. Pushouts may be utilized for modeling generalized objects with shared constituents.

12.3.3 Classes

In order to model object classes, events are labelled by class states describing the current values of attributes. Let Γ be a class signature.

Definition 17. A *Γ-class* – or *class* for short if Γ is clear from context – is a triple $C = (G, \lambda, \Gamma)$ where $G = (Ev, \to)$ is an event grove and $\lambda\colon Ev \to [\Gamma]$ is a map called *Γ-labelling for G*. λ must satisfy the following constraint: for all events $e \in Ev$ and all action symbols $a \in \Gamma_{\mathtt{acs}}$, $\circ[\lambda(e)]a$ implies that there is an event $e' \in Ev$ such that $e' \to e$ and $\triangleright[\lambda(e')]a$.

Classes are related by class morphisms which are event structure morphisms preserving labels in a canonical way.

Definition 18. Let $C_i = (G_i, \lambda_i, \Gamma_i)$, $i = 1, 2$, be classes over the same data signature. A *class morphism* $f\colon C_1 \to C_2$ is a pair $f = (g, \gamma)$ where $g\colon G_1 \to G_2$ is an event grove morphism and $\gamma\colon \Gamma_2 \to \Gamma_1$ is a class signature morphism such that $\gamma^* \circ \lambda_1 = \lambda_2 \circ g$. f is called *strict* iff γ is injective.

$$\begin{array}{ccccc} G_1 & \xrightarrow{\lambda_1} & [\Gamma_1] & & \Gamma_1 \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \gamma^*} & & \uparrow{\scriptstyle \gamma} \\ G_2 & \xrightarrow{\lambda_2} & [\Gamma_2] & & \Gamma_2 \end{array}$$

A helpful intuition for a class morphism $f\colon C_1 \to C_2$ is that f extracts C_2 as a kind of "view" from C_1, with events projected by g and states reduced via γ^*.

Example 8. Referring to Example 3, the class morphism $\tau''\colon C_{\mathtt{Customer}} \to C_{\mathtt{Person}}$ is given by $\tau'' = (g, \gamma)$ where g views customer events as person events, and γ embeds person attributes into those of customers. g is neither injective nor surjective: different customer traces may appear as the same person trace, namely if the differences are only in the customer specific parts of the states, and there may be person traces that do not appear for customers. ∎

Example 9. Referring to Example 6, the class morphism $\tau'' : C_{\texttt{Rectangle}} \to C_{\texttt{Square}}$ is given by $\tau'' = (g, \gamma)$ where $g = id$ is the identity map: rectangle events *are* square events!

What happens is that each rectangle event represents a rectangle state together with the trace of events that led to this state. With rectangle state labels reduced to square state labels along this trace, the same events define a square trace ending in a particular square state. For instance, if we assume that the square side length attribute x is inherited as the first of the two rectangle side length attributes (x, y), then the square image of a rectangle in state (3,7) is a square with side length 3. Of course, rectangles with equal side lengths coincide with their square images. Different square traces may carry the same state information along their way, so they are not distinguishable. Also some moves in a square trace may look spontaneous because nothing (visible) happens: the changes (if any) are in the specific rectangle parts of the states.

γ embeds square attributes into those of rectangles. Thus, γ^* forgets about the specific rectangle attributes. ∎

Remark 5. Corresponding with event grove category **egr**, we may define a category **class** of classes and class morphisms, and full subcategories Γ**-class** for each class signature Γ. Limits and colimits in **egr** carry over to **class**. Products in **class** model object aggregation. Pullbacks may be utilized for modeling object aggregation with overlapping components. Coproducts in Γ**-class** are useful for modeling generalization or alternative choice, and also for distinguishing instances. Pushouts may be utilized for modeling generalized classes with overlapping constituents.

12.4 Class specification in the small

For specifying classes by explicit axiomatic description, we may choose among many possible logics; see [AGM92] for a comprehensive overview. We choose a temporal logic because it is the simplest logic that can express safety as well as liveness and fairness properties. Since the pioneering work of Pnueli [Pnu77], temporal logic is widely accepted as a suitable formalism for giving axiomatic descriptions of system dynamics on a high level of abstraction. Temporal logic has been succesfully applied to a number of reactive systems specification problems. It is the simplest logic that can not only deal with safety properties but also with liveness properties. Cf. Section 12.7 for references to the literature.

Among the many temporal logics that have been suggested and investigated, we choose a propositional linear-time temporal logic which we call PTL.

In most approaches, a future-oriented temporal logic is adopted. We include past-oriented operators as well. This does not add expressive power

[LS95] but it adds convenience for specification, and it does not complicate the logic too much.

12.4.1 Temporal logic

Let Σ be an object system signature, and let Γ be a class signature (cf. Definitions 8 or 1, respectively).

Definition 19. The *syntax* of PTL is given by

$$\begin{aligned} \text{PTL} &::= \text{ATOM} \mid (\text{PTL} \Rightarrow \text{PTL}) \mid (\text{PTL}\ \mathcal{U}\ \text{PTL}) \mid (\text{PTL}\ \mathcal{S}\ \text{PTL}) \\ \text{ATOM} &::= \mathit{false} \mid T_\Sigma\, \theta\, T_\Sigma \mid T_\Gamma\, \theta\, T_\Sigma \end{aligned}$$

In the atoms, θ is a comparison operator like $=, \leq, \ldots$ It is used for comparing data terms with data terms and class terms with data terms. Class terms are just attribute constants, including actions.

Class terms are *flexible*, i.e., we intend to give them time-dependent meanings. The other symbols are *rigid*, i.e., we intend to give them time-independent meanings.

false is the usual logical constant, $\Rightarrow$ denotes logical implication, $\mathcal{U}$ is the *until* temporal operator, and $\mathcal{S}$ is the *since* temporal operator.

$\varphi\, \mathcal{U}\, \psi$ means that φ will always be true from the next moment on until ψ becomes true for the next time; φ need not be true any more as soon as ψ holds; ψ must eventually become true.

$\psi\, \mathcal{S}\, \varphi$ means that, up to the previous moment, ψ was always true since φ was true for the last time; ψ need not have been true as long as φ was; φ must once have been true.

As usual, we introduce derived connectives as follows. $\neg\, \varphi$ for $\varphi \Rightarrow \mathit{false}$, *true* for $\neg\, \mathit{false}$, $\varphi \vee \psi$ for $\neg\, \varphi \Rightarrow \psi$, etc.

We also introduce the following future-oriented derived temporal operators. $\mathsf{X}\, \varphi$ for $\mathit{false}\, \mathcal{U}\, \varphi$, $\mathsf{X}^?\, \varphi$ for $\neg(\mathsf{X}(\neg\, \varphi))$, $\mathsf{F}\, \varphi$ for $\varphi \vee \mathit{true}\, \mathcal{U}\, \varphi$, $\mathsf{G}\, \varphi$ for $\neg(\mathsf{F}(\neg\, \varphi))$, $\varphi\, \mathcal{U}^\circ\, \psi$ for $\varphi \wedge \varphi\, \mathcal{U}\, \psi$, and $\varphi\, \mathcal{P}^+\, \psi$ for $\neg((\neg\, \varphi)\, \mathcal{U}^\circ\, \psi)$.

$\mathsf{X}\, \varphi$ means that φ is true at the next point in time (*tomorrow*) which must exist. $\mathsf{X}^?\, \varphi$ means that φ is true at the next point in time if it exists. $\mathsf{F}\, \varphi$ means that φ is eventually true. $\mathsf{G}\, \varphi$ means that φ is true forever. $\varphi\, \mathcal{U}^\circ\, \psi$ means that, from *this* moment on, φ will always be true until ψ becomes true for the next time. $\varphi\, \mathcal{P}^+\, \psi$ means that φ will precede ψ, i.e., φ will be true before ψ becomes true. The latter need never happen.

Corresponding past-oriented derived temporal operators are introduced as follows. $\mathsf{Y}\, \varphi$ stands for $\mathit{false}\, \mathcal{S}\, \varphi$, $\mathsf{Y}^?\, \varphi$ for $\neg(\mathsf{Y}(\neg\, \varphi))$, $\mathsf{P}\, \varphi$ for $\varphi \vee \mathit{true}\, \mathcal{S}\, \varphi$, $\mathsf{H}\, \varphi$ for $\neg(\mathsf{P}(\neg\, \varphi))$, $\varphi\, \mathcal{S}^\circ\, \psi$ for $\varphi \wedge \varphi\, \mathcal{S}\, \psi$, and $\varphi\, \mathcal{P}^-\, \psi$ for $\neg((\neg\, \psi)\, \mathcal{S}\, \varphi)$.

$\mathsf{Y}\, \varphi$ means that φ was true at the previous point in time (*yesterday*) which must exist. $\mathsf{Y}^?\, \varphi$ means that φ was true at the previous point in time if it exists. $\mathsf{P}\, \varphi$ means that φ was once true. $\mathsf{H}\, \varphi$ means that φ was always true. $\varphi\, \mathcal{S}^\circ\, \psi$ means that, up to *this* moment, ψ was always true since φ was true

for the last time. $\varphi\ \mathcal{P}^-\ \psi$ means that, if φ ever was true, then φ preceded ψ, i.e., φ was true before ψ was true.

Furthermore, we apply the usual rules for omitting brackets.

Formal interpretation is described in terms of *possible worlds* cast in event grove terminology. If $G = (Ev, \rightarrow)$ is an event grove, then Ev is the set of possible worlds, $\rightarrow$ is an accessibility relation corresponding to *next*, and $\rightarrow^*$ is an accessibility relation corresponding to *eventually*.

Interpretation is given at events in class life cycles. Let U be a Σ-algebra, $C = (G, \lambda, \Gamma)$ a class, $L \in \mathcal{L}(G)$ a life cycle in G, and $e \in L$ an event. Satisfaction $C, L@e \models \varphi$ means that φ is valid at event e in life cycle L in class C.

Definition 20. The satisfaction relation $\models$ is inductively defined as follows.

$C, L@e \models \mathit{false}$	does not hold;
$C, L@e \models t_1 \theta t_2$	holds iff $e \in L$ and $t_{1U} \theta_U t_{2U}$;
$C, L@e \models a\theta t$	holds iff $e \in L$ and $\lambda(e)(a)\theta_U t_U$;
$C, L@e \models (\varphi \Rightarrow \psi)$	holds iff $e \in L$ and $C, L@e \models \varphi$ implies $C, L@e \models \psi$;
$C, L@e \models (\varphi\ \mathcal{U}\ \psi)$	holds iff $e \in L$ and there is a future event $e' \in L$, $e \rightarrow^+ e'$, where $C, L@e' \models \psi$ holds, and $C, L@e'' \models \varphi$ holds for every event $e'' \in L$ such that $e \rightarrow^+ e'' \rightarrow^+ e'$;
$C, L@e \models (\psi\ \mathcal{S}\ \varphi)$	holds iff $e \in L$ and there is a past event $e' \in L$, $e' \rightarrow^+ e$, where $C, L@e' \models \varphi$ holds, and $C, L@e'' \models \psi$ holds for every event $e'' \in L$ such that $e' \rightarrow^+ e'' \rightarrow^+ e$.

By the abbreviations introduced above, we may derive satisfaction conditions for the other connectives and temporal operators, e.g.,

$C, L@e \models (\mathsf{X}\,\varphi)$	holds iff $e \in L$ and there is a next event $e' \in L$, $e \rightarrow e'$, where $C, L@e' \models \varphi$ holds.

As usual, a formula φ is said to be *satisfiable* in a class C iff $C, L@e \models \varphi$ holds for some life cycle L in C and some event e in L. A formula φ is said to be *valid* in C, in symbols $C \models \varphi$, iff $C, L@e \models \varphi$ holds for all life cycles L in C and all events e in L. A set of formulae Φ *entails* a formula φ in C, in symbols $C, \Phi \models \varphi$, iff φ holds at every event in every life cycle in C at which all formulae in Φ hold.

12.4.2 Temporal specification

Let Γ be a class signature as given in Definition 1.

Definition 21. A *class specification* is a pair $\mathit{Cspec} = (\Gamma, \Phi)$ where Φ is a set of PTL formulae over Γ.

Semantics is defined in the subcategory Γ**-class** $\subseteq$ **class** where the signature Γ is fixed and, for every morphism $f = (g, \gamma)$, γ is the identity morphism.

Γ**-class** has final elements[2]. Such an element is unique up to isomorphism. It represents the most liberal behavior over Γ. Intuitively, its traces are all those that can be formed over Γ.

A corresponding result holds for subcategories of classes satisfying given temporal axioms. Let $Cspec = (\Gamma, \Phi)$ be a specification. Let *Cspec*-**class** be the full subcategory of Γ-**class** consisting of all classes over Γ satisfying Φ.

Cspec-**class** has final elements. A final element is obtained as the maximal subclass of a final element in Γ-**class** satisfying Φ.

Corresponding finality results are given in [CSS94].

We may utilize these results for assigning standard abstract semantics to temporal class specifications. In order to increase specification power and manageability, we envisage to give direct axiomatic specification only to basic classes. Derived and complex classes may be specified using in-the-large structuring mechanisms like those described in the next section.

12.5 Class specification in the large

For an effective specification method, in-the-large structuring mechanisms are indispensable. We express structuring by structured object sorts and corresponding object type morphisms.

Object sorts are structured in two ways, by order sorting and by object sort constructors. Inheritance (is-a and as-a) and hiding are expressed by binary ordering relationships on object sorts. Composition is expressed by binary sort constructors for generalization and aggregation. Interaction between components of a complex object can be specified in the framework developed so far.

We note in passing that state variables (cf. Example 1) may be understood as specified by a sort constructor Var: $S^d \to S^o$. This is one way to express parameterization. Parameterization is a powerful in-the-large concept. Unfortunately, because of space limitations, we cannot go into this issue here.

12.5.1 Inheritance

Inheritance describes how a class reuses features from another one. On a representation level, the inheriting class adopts attributes and possibly adds some. On an implementation level, the inheriting class reuses code, i.e., it implements inherited attributes (and actions) in the same way as in the original class.

We look at inheritance from a semantic point of view. This means that the inheriting class provides attributes (and actions) with the same syntax and semantics as the original one. Nothing is incurred for implementation. In fact,

[2] For obvious reasons, we avoid the term *object* for the elements of a category although it is standard in category theory.

even if its semantics does not change, an inherited action may have a separate implementation, for instance a more efficient one. This approach allows for multiple representations and implementations, even within one type. This may be useful, for instance, in a distributed environment with heterogeneous processors.

Let $\Sigma = (S, \Omega)$ be an object system signature where $S = S^d + S^o$, with S^d and S^o being the sets of data and object sorts, respectively. Let U be a Σ-algebra.

On the syntactic level, inheritance is expressed by *object sort ordering*, i.e., by a partial ordering $\leq$ on object sorts S^o.

Let $r, s \in S^o$ be object sorts. The intended interpretation of $r \leq s$ is that each object of sort r inherits from an object of sort s. Depending on the kind of inheritance considered, it is formalized in our approach by an object type morphism $\tau\colon U_r \rightrightarrows U_s$ or a reverse object type morphism $\tau\colon U_r \rightleftarrows U_s$.

We distinguish between *is-a inheritance* $r \leq_{\rightarrow} s$, *as-a inheritance* $r \leq_{\leftarrow} s$, and *hiding* $r \sqsubset s$. For the first two, we indicate how to generalize the concepts so as to allow for overriding and overlaying. All kinds of inheritance may be multiple.

Is-a inheritance. An is-a inheritance relationship $r \leq_{\rightarrow} s$ is intended to express a kind of temporal specialization. Technically, $r \leq_{\rightarrow} s$ is to be interpreted by an is-a inheritance morphism defined as follows.

Definition 22. An *is-a inheritance morphism* is an object type morphism $\tau\colon U_r \rightrightarrows U_s$ such that $\tau'\colon Id_r \subseteq Id_s$ is an inclusion and $\tau''\colon C_r \to C_s$ is strict.

For example, each customer is a person, cf. Example 3. In systems where objects are represented by their identities, is-a inheritance naturally leads to polymorphism because the type of an object is not unique.

The class morphism $\tau''\colon C_r \to C_s$ that goes with an is-a inheritance morphism reflects the associated class inheritance. Referring to Example 8, if a customer is a person, then the class morphism $\tau_1''\colon C_{\texttt{Customer}} \to C_{\texttt{Person}}$ says that every customer life cycle has a valid underlying person life cycle.

Is-a inheritance may be *multiple*, i.e., we have $r \leq_{\rightarrow} s_1, \ldots, r \leq_{\rightarrow} s_n$ where the object sorts s_k, $1 \leq k \leq n$, are distinct. On the interpretation level, this leads to object type morphisms $\tau_k\colon U_r \rightrightarrows U_{s_k}$, $1 \leq k \leq n$, expressing from which objects in $U_{s_1}, \ldots, U_{s_n}$ a given object in U_r inherits and what is being inherited.

The *multiple inheritance problem* does not appear in our context, it has to do with resolving naming conflicts on the syntactic level. An elegant categorial treatment of this issue may be found in [LcP90].

It is obvious from the definition that is-a inheritance $\leq_{\rightarrow}$ is a reflexive and transitive relation. Antisymmetry does not hold in a strict sense but $r \leq_{\rightarrow} s$ and $s \leq_{\rightarrow} r$ implies that we have $Id_r = Id_s$, and C_r and C_s are equivalent in a rather strong sense. Because of space limitations, we cannot elaborate on this point.

As-a inheritance. Like is-a inheritance, as-a inheritance $r \leq s$ is intended to express a kind of temporal specialization. The difference is that the former describes that *all* features are inherited and new ones may be added, whereas as-a inheritance describes that *some* features are inherited and possibly used in a special way.

Technically, $r \leq s$ is intended to be interpreted by an as-a inheritance morphism defined as follows.

Definition 23. An *as-a inheritance morphism* is a reverse object type morphism $\tau\colon U_r \rightleftarrows U_s$ such that $\tau'\colon Id_r \subseteq Id_s$ is an inclusion and $\tau''\colon C_s \to C_r$ is strict.

For example, we view a rectangle as a square in Example 6. On the identity level, there is no difference to is-a inheritance. In fact, each square is a rectangle as well. Consequently, also as-a inheritance leads to polymorphism.

The difference is that on the class level the morphisms go the other way. For example, if we take a rectangle as a square, then there is a class morphism $\tau_2''\colon C_{\texttt{Rectangle}} \to C_{\texttt{Square}}$ saying how rectangles can be viewed as squares, cf. Example 9.

The class morphism $\tau''\colon C_s \to C_r$ that goes with an as-a inheritance morphism reflects the intended behavior relationship. The rectangle-as-a-square example suggests that it is natural to assume that the underlying event grove morphism is an identity.

We may also have multiple as-a inheritance. In fact, multiple inheritance may involve any combination of is-a and as-a.

It is obvious from the definition that as-a inheritance $\leq$ is a reflexive and transitive relation. As for antisymmetry, the situation is similar to the is-a case.

Overriding and overlaying. While pure inheritance as introduced above only allows for adding new attributes (and actions) while adopting the old ones, overriding gives the possibility to change some of the old ones, i.e., give them a new meaning. Overlaying provides new versions of attributes while keeping the old ones accessible.

We explain the matter for is-a inheritance, the case is similar for as-a inheritance. A *weak* is-a relationship allowing for overriding is denoted by $r \leq s$. Let $U_r = (Id_r, C_r)$ and $U_s = (Id_s, C_s)$. Technically, $r \leq s$ is to be interpreted by

- an inclusion $Id_r \subseteq Id_s$,
- strict class morphisms $C_r \to C' \leftarrow C_s$ for some class C', and
- a class signature injection $\gamma\colon \Gamma_s \to \Gamma_r$,

where Γ_s and Γ_r are the signatures of C_r and C_s, respectively. Let Γ' be the signature of C'. For ease of notation, we assume that the signature morphisms

underlying the strict class morphisms are inclusions. Then we have $\Gamma' \subseteq \Gamma_r \cap \Gamma_s$.

If $a_s \in \Gamma_s$, then $a_r = \gamma(a_s)$ is a_s's new version in class C_r. We may have one of the following cases.

$a_s \in \Gamma'$: a_s is inherited ...
- $a_r = a_s$: ... and coincides with the new version.

  $a_r \neq a_s$: ... but is overlayed by a_r ...

  $a_r \in \Gamma'$: ... that is equal to some other inherited item.
- $a_r \notin \Gamma'$: ... that is new.

$a_s \notin \Gamma'$: a_s is overridden by a_r ...

$a_r \in \Gamma'$: ... that is equal to some other inherited item.
- $a_r \notin \Gamma'$: ... that is new.

The bullet • indicates cases of practical interest.

Hiding. While is-a and as-a inheritance reflect different kinds of temporal specialization, the intention of hiding is quite different. It may therefore not be justified to subsume it under inheritance but the formalities on the class level are similar to as-a inheritance.

A hiding relationship $r \sqsubset s$ is intended to express that r-objects are *interfaces* or *views* of s-objects. These are dependent objects like as-a specializations, but they have identities of their own.

Definition 24. A *hiding morphism* is a reverse object type morphism $\tau\colon U_r \rightleftarrows U_s$ such that $Id_r \cap Id_s = \emptyset$ and $\tau''\colon C_s \to C_r$ is strict.

There is no point to allow for overriding or overlaying, interfaces are designed to fully reflect what happens in the base object.

In analogy to multiple inheritance, we may have "multiple interfaces", i.e., objects that are interfaces of several base objects. Multiple interfaces synchronize their base objects because each action, attribute, etc, in the interface is shared by all base objects. Categorially, the synchronized behavior is given by a limit in **class**.

The other way round, we may also have several interfaces of the same base object, even of the same sort. The analogous inheritance situation is to have several specializations of the same object – which cannot be of the same sort, though.

There is another difference to as-a specialization. The intended way to work with the latter is to use only the restricted features of the specialization while the other features are not accessible, they are out of scope. Technically, this is enforced by giving the specialization the same identity and rely on the usual object-oriented rule that identities are unique at any time. In contrast, giving interfaces and their base objects different identities means that they may coexist in a state.

It is obvious from the definition that hiding $\sqsubset$ is an irreflexive relation. Transitivity holds if the identity sets involved are pairwise disjoint. As for antisymmetry, the situation is similar to the is-a and as-a cases.

Covariance vs. contravariance. There is one more aspect of inheritance that is worth mentioning, namely changing sorts of inherited attributes. In functional approaches using subsorting, operators are *contravariant* with respect to range sorts in a natural way: if $\omega\colon s \to t$ and $s' \leq s$, then we have $\omega\colon s' \to t'$ for any t' such that $t \leq t'$. This also holds for attributes and other object-sorted functions if we work with inheritance. For instance, if *spouse* : *Person* is an object-valued attribute for *Person*, if each *Male* is a *Person*, and if each *Person* is a *LivingBeing*, then we naturally have *spouse* : *LivingBeing* as an attribute for *Male*.

While this is undoubtedly true, it is not particularly helpful. If we work with inheritance in object-oriented systems, we have in mind to refine the specification with more detail. In our example, if we introduce *Male* and *Female* as special kinds of *Person*, then we will take the opportunity to refine the *spouse* attribute as well and specify *spouse* : *Female* for *Male*, and *spouse* : *Male* for *Female*.

Therefore, it is natural to have *covariance* with respect to range sorts in object-oriented systems: if $\omega\colon s \to t$ and $s' \leq s$, then we impose $\omega\colon s' \to t'$ for a suitable $t' \leq t$.

12.5.2 Composition

Composition describes how constituent objects are put together to form complex objects. On the syntactic level, composition is expressed by object sort constructors. We have two kinds of composition, generalization $r \oplus s$ and aggregation $r \otimes s$. These constructors are associative, so the same constructor may be iterated a finite number of times.

On the semantic level, generalization is expressed by coproducts in a category $\mathbf{ot}(\Gamma\textbf{-class})$ of object types over a given class subcategory Γ**-class** with a fixed signature Γ, and aggregation is expressed by products in the category $\mathbf{ot}(\mathbf{class})$ of object types over the category **class** of all classes with varying signatures.

Generalization. Composition by generalization $r \oplus s$ is intended to express how objects can be put together to reflect alternative choice among its constituents. We assume that we have $\Gamma_r = \Gamma_s = \Gamma$ for the class signatures of sorts r and s, respectively, cf. Example 4 for motivation.

Interpretation is given by $U_{r\oplus s} = U_r + U_s$, the coproduct in $\mathbf{ot}(\Gamma\textbf{-class})$. The coproduct morphisms describe how the constituent object types are embedded into the generalization.

If $U_r = (Id_r, C_r)$ and $U_s = (Id_s, C_s)$, then $U_{r\oplus s} = (Id_{r\oplus s}, C_{r\oplus s})$ where $Id_{r\oplus s} = Id_r + Id_s$ is the disjoint union, and $C_{r\oplus s} = (G_r + G_s, \lambda_r + \lambda_s, \Gamma)$ where $G_r + G_s$ is the coproduct of event groves, and $\lambda_r + \lambda_s\colon Ev_r + Ev_s \to [\Gamma]$ associates the original label with each event.

If $G_r = (Ev_r, \to_r)$ and $G_s = (Ev_s, \to_s)$ are event groves, their coproduct is $G_r + G_s = (Ev_r + Ev_s, \to_r + \to_s)$. The associated sequential event structure

$E(G_r + G_s)$ has conflict relation $\# = \#_r + \#_s + Ev_r \times Ev_s$ provided that Ev_r and Ev_s are disjoint (otherwise, disjoint copies of these sets must be taken). Intuitively, this means that no events from different constituents of a generalization may be in one and the same life cycle, so there is a choice at the beginning to pursue a life cycle that lies in one of the constituents.

Example 10. We refer to Example 4 where *Customer* = *Person* + *Company*. Given specifications for *Person* and *Company* with object sorts `Person` and `Company`, respectively, we may specify `Customer` = `Person`⊕`Company` in order to achieve the desired interpretation. ∎

For object sorts $r, s \in S^o$, we have $r \leq r \oplus s$ and $s \leq r \oplus s$. For instance, in the example above, we may view a person as a customer.

Aggregation. Composition by aggregation $r \otimes s$ is intended to express how objects are put together to form complex objects. A life cycle of a complex object is a step-by-step synchronization of life cycles of its components, sharing events at each step.

Interpretation is given by $U_{r\otimes s} = U_r \bowtie U_s$, the product in the category **ot(class)**. The product morphisms describe projections from a complex object to its components.

If $U_r = (Id_r, C_r)$ and $U_s = (Id_s, C_s)$, then $U_{r\otimes s} = (Id_{r\otimes s}, C_{r\otimes s})$ where $Id_{r\otimes s} = Id_r \times Id_s$ is the cartesian product, and $C_{r\otimes s} = (G_r \times G_s, \lambda_r \times \lambda_s, \Gamma_r + \Gamma_s)$ where $G_r \times G_s$ is the product of event groves, and $\lambda_r \times \lambda_s \colon Ev_r \times Ev_s \to [\Gamma_r + \Gamma_s]$ associates the labellings of component events with a complex event: if, say, $a \in \Gamma_r$, then $\lambda_r \times \lambda_s(e_r, e_s)(a) = \lambda_r(e_r)(a)$.

If $G_r = (Ev_r, \to_r)$ and $G_s = (Ev_s, \to_s)$ are event groves, their product is given by $G_r \times G_s = (Ev_{r\otimes s}, \to_{r\otimes s})$ where $Ev_{r\otimes s} = \{(e_r, e_s) \mid e_r \in E_r, e_s \in E_s,$ and $\tau_{G_r}(e_r) = \tau_{G_s}(e_s)\}$, and, for all $e_r, e'_r \in Ev_r$ and all $e_s, e'_s \in Ev_s$, $(e_r, e_s) \to (e'_r, e'_s)$ holds iff both $(e_r \to_r e'_r)$ and $(e_s \to_s e'_s)$ hold.

Example 11. We refer to Example 5 where *Car* = *Motor* ⋈ *Chassis*. Given specifications for *Motor* and *Chassis* with object sorts `Motor` and `Chassis`, respectively, we may specify `Car` = `Motor` ⊗ `Chassis` in order to achieve the desired interpretation. ∎

For object sorts $r, s \in S^o$, we impose $r \sqsubset r \otimes s$ and $s \sqsubset r \otimes s$. For instance, in the example above, we view both a motor and a chassis as car interfaces, giving a partial view and hiding the rest. As a consequence, in any interpretation, each component belongs to precisely one object. However, note that we may have several hiding relationships on r or s or both so that we may have shared components.

Component interaction. Synchronous interaction between components of a complex object may be described by temporal axioms within the aggregation.

Example 12. In the `Bank` Example 2, we may replace the second and fifth axioms by the strict rule

$\odot$ `transfer(from,to,m)` $\Rightarrow$ ($\odot$ `A(from).debit(m)` $\wedge$ $\odot$ `A(to).credit(m)`)

expressing that a transfer action calls a debit action in the `from` account and a credit action in the `to` account. This means that all three actions must occur simultanously. ∎

Interaction axioms are not restricted to this simple kind of *event calling* although these describe many cases of interest. It is conceivable to have liveness axioms like $\odot\,\alpha \Rightarrow \mathsf{F}\odot\beta$, safety axioms like $\odot\,\alpha \Rightarrow \mathsf{G}\neg\triangleright\beta$, exclusion axioms like $\odot\,\alpha \Rightarrow \neg\odot\beta$, etc.

12.6 Object systems

For coping with object systems, it is not practical to work with a sequential model that requires global synchronization of all events. With a concurrent model, however, we must be careful to stay within manageable limits: a model powerful enough to deal with all aspects of concurrency is hard to work with. Our way out is to use linear-time distributed temporal logic for specification, and locally sequential labelled event structures as a denotational model. This is expressive enough to reflect the kind of concurrency that often occurs in practice, for instance in networks of work stations. On the other hand, the model is close enough to sequential classes and temporal logic to allow for a smooth extension of familiar concepts.

Let $\Sigma = (S, \Omega)$ be an object system signature, interpreted by a data and object algebra U. Let $U_r = (Id_r, C_r)$ and $U_s = (Id_s, C_s)$ be object types in U. Any two instances $i.C_r \in U_r$ and $u.C_s \in U_s$ are assumed to operate concurrently subject to given inheritance constraints. For simplicity, we assume that there are no such constraints so that a system consists of independent objects $i.C_r$, $u.C_s$, ... The classes are not necessarily distinct.

12.6.1 A denotational system model

In order to show the principle, we may as well assume that the system consists of just two objects $i.C_r$ and $u.C_s$ operating concurrently. Its behavior is given by $B = C_r \times C_s$, the concurrent product of their classes. We briefly describe the product construction for labelled prime event structures, first treating unlabelled ones and then adding labels. The construction is due to Vaandrager [Vaa89].

Let $E_r = (Ev_r, \rightarrow_r, \#_r)$ and $E_s = (Ev_s, \rightarrow_s, \#_s)$ be event structures. Let $Ev_r^\perp = Ev_r + \{\perp_r\}$, $Ev_s^\perp = Ev_s + \{\perp_s\}$, and $Z = Ev_r^\perp \times Ev_s^\perp - \{(\perp_r, \perp_s)\}$. An element (e_r, e_s) where $e_r \in Ev_r$ and $e_s \in Ev_s$ characterizes a shared event where e_r and e_s happen synchronously. An element $(e_r, \perp_s)$ characterizes an event e_r occurring in isolation in E_r, i.e., concurrently to whatever happens in

E_s. Correspondingly, an element $(\perp_r, e_s)$ characterizes an event e_s occurring in isolation in E_s.

For $e = (e_r, e_s), e' = (e'_r, e'_s) \in Z$, let $e \rhd e'$ iff $e_r \rightarrow_r^* e'_r$ or $e_s \rightarrow_s^* e'_s$. This auxiliary relation keeps track of local causalities but is not a causality relation itself. For a subset $X \subseteq Z$, let $\rhd_X = \rhd \cap (X \times X)$.

Definition 25. A subset $X \subseteq Z$ is called a *preconfiguration* iff (1) $pr_r(X) \in \mathcal{C}(E_r)$ and $pr_s(X) \in \mathcal{C}(E_s)$ and (2) $\rhd_X^+$ is a partial order. X is called a *complete prime* iff it has a unique maximal element with respect to $\rhd_X^+$.

A preconfiguration X characterizes a set of events that has occurred at a given moment, reflected locally by configurations $pr_r(X)$ in E_r and $pr_s(X)$ in E_s. Condition (2) says that the events of the components may occur only once and that both components must agree on the causal relationships between events: causal loops are not allowed.

The following definition applies a standard procedure to define a prime event structure from its finite configurations.

Definition 26. The *concurrent product of E_r and E_s* is defined by $E_r \times E_s = (Ev, \rightarrow^*, \#)$ where $Ev = \{X \mid X \text{ is a complete prime}\}$, $X \rightarrow^* Y$ iff $X \subseteq Y$, and $X \# Y$ iff $X \cup Y$ is not a preconfiguration.

A life cycle in $E_r \times E_s$ is a union of life cycles in E_r and E_s, sharing events at synchronization points. If E_r and E_s are sequential, then the local life cycles are linear traces. Thus, a global system life cycle consists of local traces that are glued together at shared events.

In order to make the product construction complete, we have to add labels. We restrict our attention to classes where we have the definitions available. Let $C_r = (G_r, \lambda_r, \Gamma_r)$ and $C_s = (G_s, \lambda_s, \Gamma_s)$ be classes. Of course, the concurrent product of event groves is defined to be that of the prime event structures they present. Let $\lambda' \colon Z \rightarrow [\Gamma_r + \Gamma_s]$ be defined by $\lambda'(e_r, \perp_s) = \lambda_r(e_r)$, $\lambda'(\perp_r, e_s) = \lambda_s(e_s)$, and $\lambda'(e_r, e_s)(a) = \lambda_r(e_r)(a)$ if $a \in \Gamma_r$, and $= \lambda_s(e_s)(a)$ if $a \in \Gamma_s$. Note that Γ_r and Γ_s are disjoint in our context.

Definition 27. The *concurrent product of C_r and C_s* is defined by $C_r \times C_s = (G_r \times G_s, \lambda, \Gamma_r + \Gamma_s)$ where $\lambda(X) = \lambda'(x)$ such that x is the unique maximal element in X with respect to $\rhd_X^+$.

Isolated events retain their local labels, and synchronized events obtain their labels in the same way as in aggregated objects, cf. Subsection 12.5.2. Note that concurrent events always have distinct labels, there is no "auto-concurrency". A constructive definition of event structure product is given in [LG91].

12.6.2 Distributed temporal specification

An object system specification consists of an object instance signature Υ and a set family of object axioms Φ, cf. Definition 10. Our specification logic is based on a version of the *n-agent logic*, cf. Section 12.7 for hints to the literature. An agent corresponds to an object that may be thought of as a site in a distributed system. Temporal descriptions are given locally from the viewpoints of agents. Interaction comes in by incorporating statements about other agents. For example, agent i may prescribe that, whenever i sends a message to agent u, the latter will eventually acknowledge receipt. The actions of agent i sending and agent u receiving may be modeled to occur concurrently. Acknowledgement, however, requires interaction, it may be modeled as a synchronized joint event where both talk to each other: i says "I sent something to you" while u says "I received something from you".

We introduce our version of n-agent logic called DTL. Let $Id = \{i, u, \dots\}$ be a given set of agent identities, also called *localities*.

Definition 28. The *syntax* of DTL, i.e., its set of well-formed formulae Φ, is given by

$$\text{DTL} ::= \text{ATOM} \mid (\text{DTL} \Rightarrow \text{DTL}) \mid (\text{DTL}\ \mathcal{U}_{Id}\ \text{DTL}) \mid (\text{DTL}\ \mathcal{S}_{Id}\ \text{DTL}).$$

where ATOM is defined as in Definition 19.

The only difference with respect to PTL is that we have temporal operators $\mathcal{U}_i$ and $\mathcal{S}_i$ indexed by localities. The intended meaning of $\varphi\,\mathcal{U}_i\,\psi$ is that $\varphi\,\mathcal{U}\,\psi$ holds locally at agent i, and correspondingly for $\mathcal{S}_i$. Like for PTL, we may introduce abbreviations X_i, $\mathsf{X}_i^?$, F_i, G_i, $\mathcal{U}_i^\circ$ and $\mathcal{P}_i^+$ as well as Y_i, $\mathsf{Y}_i^?$, P_i, H_i, $\mathcal{S}_i^\circ$ and $\mathcal{P}_i^-$.

Additionally, we introduce the abbreviation $@\,u$ for $\mathsf{X}_u^?\,\mathit{true}$, saying that there is interaction with u at this point in time, i.e., $@\,u$ holds precisely at events shared with u.

Specification in DTL is bound to localities. That means that the formulae consist of a set family $\Phi = \{\Phi\}_{i \in Id}$ of *local axioms*. The notation $i : \varphi$ means that $\varphi \in \Phi_i$ where $i \in Id$.

Interaction is specified by referring to another locality by using its local temporal operators. We illustrate the idea by examples.

Example 13. In order to give the examples a personal touch, we read "I" for i and "you" for u.

$i : \mathsf{P}_u\,\varphi$ I hear that φ was once valid for you

$i : \mathsf{G}_i(\,@\,u \;\Rightarrow\; \mathsf{X}_i\,\varphi)$ whenever I talk to you, I have φ the next day

$i : \varphi \Rightarrow \mathsf{X}_i\,@\,u$ if φ holds, then I talk to you the next day

$i : \mathsf{X}_u\,@\,i$ you tell me that you will contact me tomorrow

$i : \varphi \Rightarrow \mathsf{X}_u\,\psi$ if φ holds, then you tell me that ψ will hold for you tomorrow

$i : \mathsf{Y}_i\, \mathsf{F}_u\, \varphi$ you told me yesterday that φ will hold for you some time

$i : (i. \odot \mathrm{send}(x) \Rightarrow \mathsf{F}_i\ \mathsf{P}_u\, u. \odot \mathrm{receive}(x))$ if I send x, then I will eventually obtain an acknowledgement from you that you received x

■

Interpretation is given at an event e in a life cycle L in a distributed system $B = \prod_{i \in Id} C_i$. L is a web of local linear traces $L_i, L_u, \ldots$ sharing events. Satisfaction is defined locally. Let $i \in Id$ be a locality, and let $e \in L$. $B, L@e \models_i \varphi$ means that φ holds locally at event e in life cycle L_i in system B.

Here the generality of Definition 20 pays off: we may replace C by B, $\models$ by $\models_i$, $\mathcal{U}$ by $\mathcal{U}_i$, $\mathcal{S}$ by $\mathcal{S}_i$, $\rightarrow$ by $\rightarrow_i$, etc. and add the following two rules in order to obtain the definition of local validity.

$B, L@e \models_i (\varphi\, \mathcal{U}_u\, \psi)$ iff $e \in L_i, e \in L_u,$ and $B, L@e \models_u (\varphi\, \mathcal{U}_u\, \psi)$ hold;
$B, L@e \models_i (\psi\, \mathcal{S}_u\, \varphi)$ iff $e \in L_i, e \in L_u,$ and $B, L@e \models_u (\psi\, \mathcal{S}_u\, \varphi)$ hold.

The life cycles consist of n local traces that may share events. These shared events are points of interaction. Note that the local state $\downarrow e$ of an event $e \in Ev_i$ may contain events of other objects, for instance an event e'' that is causal in u for an interaction event e' with u that is in turn causal for e, i.e., we have $e'' \rightarrow_u e' \rightarrow_i e$. In this sense, agents obtain full historic information about others they have talked to – and about those the others have talked to, etc.

Example 14. For illustration, we refer to Example 2 and give a concurrent version of the `Bank` class.

```
class Bank;
   uses money, acct#, Person, Account;
   attributes owner: Person;
   components Acc(acct#): Account;
   actions    *establish(Person),
              transfer(Account,Account,Bank,money);
              receive(Account,Bank,Account,money);
              +liquidate;
   axioms var   p: Person, from, to, to2: acct#,
                b,b2: Bank, m, n: money;
      ⊙establish(p) ⇒ owner=p,
      ⊙transfer(from,to,b,m) ⇒ X ⊙Acc(from).debit(m),
      ⊙transfer(from,to,b,m) ⇒ ¬ ⊙transfer(from,to,b,m)
                U° P_b b.⊙receive(from,self,to,m),
      ⊙transfer(from,to,b,m) ∧ ⊙transfer(from,to2,b2,m)
                ⇒ to=to2 ∧ b=b2,
      ⊙receive(from,b,to,m) ⇒
                ⊙Acc(to).credit(m) P⁺ ⊙Acc(to).debit(n),
      ⊙receive(from,b,to,m) ⇒ F ⊙Acc(to).credit(m)
end
```

Comparing with Example 2, we introduce one more parameter in the transfer action, namely the bank `b` to which the money is to be transferred. We also add an action `receive(from,b,to,m)` of receiving money `from` an account of another bank `b`. The standard variable `self` denotes "this" bank, it is used when there is a need to refer to the hidden local object variable (cf. Example 1).

Introducing `Bank` variables like `b`, we can talk locally about transfers to and from another bank. Remember that local variables and localities are omitted. For instance, the full version of the third axiom is

$$\texttt{self: self.}\odot\texttt{transfer(from,to,b,m)} \Rightarrow \neg\texttt{self.}\odot\texttt{transfer(from,to,b,m)}\ \mathcal{U}^{\circ}_{\texttt{self}}\ \mathcal{P}^{+}_{\texttt{b}}\ \texttt{b.}\odot\texttt{receive(from,self,to,m)}$$

The axioms should be understandable without further explanation.

The following liveness property for bank i is a consequence of the third axiom.

$$\texttt{i.}\odot\texttt{transfer(from,to,b,m)} \Rightarrow \mathsf{F}_{\texttt{i}}\,\mathsf{P}_{\texttt{b}}\,\texttt{b.}\odot\text{receive(from,i,to,m).}$$

The property says that transfers are eventually acknowledged as having been received. ∎

Let $\mathit{Sspec} = (\Upsilon, \Phi)$ be an object system specification with a set $\mathit{Id} = \{i, u, \dots\}$ of identities. Its models are distributed life cycles in a systems $B = \prod_{i \in \mathit{Id}} C_i$ satisfying Φ, i.e., $B, L@e \models_i \Phi_i$ for every identity $i \in \mathit{Id}$ and every event $e \in L_i$. The structure of the model category is subject to further study. For a related but slightly different setting, an initiality result has been proved in [ES95].

12.7 Related work

There are a number of methods for object-oriented analysis, modeling and design. These methods are informal or semiformal at best. However, they come along with methodological guidelines and graphical notations. They help to make formal languages fit for use, so they do have their benefits in early modeling and design stages. Recently, there has been considerable activity to unify these methods: the Booch [Boo94], OMT [RBP+91] and OOSE methods [Jac92] merged into the universal modeling language UML [FS97] that has been submitted to the Object Management Group to be considered as a standard. Another successful OO analysis method is Fusion [CAB+94].

However, these methods are too unprecise and ambiguous when it comes to animation, verification and forecasting of system properties, and when it comes to generating test cases or even implementations from specifications.

Among the logic-based formal methods, the work reported here is based on experiences with developing the OBLOG family of languages and their

semantic foundations. OBLOG is being developed into a commercial product [ESD93]. In the academic realm, there are several related developments: TROLL [SJH93,SHJE94,JSHS96,HJ95,EH96,Har97,DH97], GNOME [SR94], LCM [FW93] and ALBERT [DDPW94].

There are other approaches to formal object specification with a sound theoretical basis. The ones most closely related to ours are FOOPS [GM87, RS92,GS95] and MAUDE [Mes93]. FOOPS is based on OBJ3 [GW88] which is in turn based on equational logic. MAUDE is based on rewriting logic that is a uniform model of concurrency [Mes92]. Other language projects working on related ideas are OOZE [AG91] and ETOILE [AB95]. We also acknowledge inspiration by other work, e.g., [AZ95,Bee95,FM92].

In the OBLOG family, TROLL3 [EH96,Har97] is the first to address problems of concurrency and communication, and to integrate benefits from the informal methods mentioned above. So there is a graphical notation for TROLL3 called OMTROLL that adopts elements from OMT. [EH96] gives a brief overview of TROLL3 and OMTROLL and their logic foundations.

The denotational object and system models adopted in this chapter are based on prime event structures. These are abstract and elegant, and powerful enough to model full concurrency. Their relationship to other models of concurrency like labelled transition systems and Petri nets is well investigated, cf. [NPW81,NRT92,WN95]. And they have been used as a semantic basis for extending the logic to deal with concurrency, see below.

As for the many logics that might be useful for class and system specification, we refer to [AGM92] for a comprehensive overview. The specification logic defined in this paper is based on temporal logic, cf. [Pnu77] for the pioneering paper and [MP92,MP95] for more recent textbooks. Temporal logic has also been used as the backbone of a large-scale programming system [Tan94]. Our logic is influenced by OSL [SSC95] and the logic of [Aba89] that goes back to [GPSS80] where also a sound and complete proof system is given.

The extension of PTL towards concurrency introduced in Section 12.6 is based on n-agent logic that was introduced and developed in [LT87,LMRT91, LRT92,MT92,Thi94,Ram96]. A distinguishing feature is that the assumption of an omnipresent observer of the entire system under consideration is dropped and replaced by a local causal perspective. n-agent logics can explicitly distinguish sequential agents (localities) in the system, refer to the local viewpoint of each agent, and express communication between agents [LT87, LRT92,Ram96]. The latter is the major feature of distribution.

These ideas have been addressed in the context of object orientation [ES95] and used to axiomatize a significant subset of the GNOME language [Cal96]. In [ECSD98] , two distributed logics are defined, an operational one with basic communication facilities, and a fancier one that can talk about communication in an elegant implicit way. The main result is that there is a sound and complete translation from the latter to the former.

12.8 Concluding remarks

The theory outlined here is taking shape but is not complete. It has to be elaborated and refined in several respects.

In particular, the details of class and system composition have to be worked out. For instance, laws for object type construction terms like $T+T \simeq T$, $(T_1+T_2) \bowtie T_3 \simeq (T_1 \bowtie T_3)+(T_2 \bowtie T_3)$, $(T_1+T_2)\times T_3 \simeq (T_1\times T_3)+(T_2\times T_3)$, etc. have to be set up and proved, based on a suitable equivalence relation $\simeq$. This is the basis for an in-the-large algebraic treatment and optimization of concurrent system construction.

Further research will focus on interaction and modularization concepts in truly concurrent models. DTL describes synchronous and symmetric interaction from a local point of view. For the TROLL language, a richer spectrum of interaction concepts is envisaged, including modes of asynchronous directed interaction. These can be explained on DTL grounds.

As for modularization, we envisage a module concept that reflects generic building blocks of software. In particular, instantiation as well as horizontal and vertical composition of modules must be supported. For the latter, a module must incorporate a reification step between an external interface on a higher level of abstraction and an internal interface on a lower level of abstraction [Den96b,DE95,Den96a].

Another issue of practical importance is real time. Real-time constraints set limits to when an action may or must occur or terminate, and how long it may take from a triggering event to the corresponding reaction.

On the longer range, deductive capabilities and default handling must be better understood and eventually incorporated. The role of deduction is to predict the effect of a design before it is implemented. Defaults enhance modularity by allowing assertions to be made in a local object, even when the vocabulary needed to specify their exceptions is unavailable.

Along with these theoretical developments, languages and systems have to be developed, supported by tools and tested in application case and field studies. The TROLL project has proceeded in this direction, and its experiences are encouraging.

Acknowledgments

The author is grateful for help and inspiration from many sides. Amílcar and Cristina Sernadas have been faithful partners in developing the theory. The other partners in the ISCORE, COMPASS and ASPIRE projects have given so manyfold inputs that it is impossible to acknowledge in detail. Grit Denker and Juliana Küster Filipe have contributed essentially to internal discussions on the subjects of the chapter. The practical work on TROLL has provided important input, especially by Gunter Saake, Thorsten Hartmann, Jan Kusch and Peter Hartel. Mojgan Kowsari puts TROLL to practice and has given useful feedback. Grit Denker, Juliana Küster Filipe, Michaela Huhn and Narciso

Martí-Oliet have reviewed earlier versions of the paper and given valuable hints. All these contributions are gratefully acknowledged. Naturally, only the author is responsible for all remaining deficiencies.

References

[AB95] M. Aiguier and G. Bernot. Algebraic specification of object type specifications. In R.J. Wieringa and R.B. Feenstra, editors, *Information Systems—Correctness and Reusability, Selected Papers from the ISCORE Workshop'94*, pages 16–32. World Scientific Publishers, 1995.

[Aba89] M. Abadi. The power of temporal proofs. *Theoretical Computer Science*, 65:35–83, 1989.

[AG91] A.J. Alencar and J.A. Goguen. OOZE: An object-oriented *z* environment. In P. America, editor, *Proc. ECOOP'91*, pages 180–199. Springer, 1991.

[AGM92] S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum. *Handbook of Logic in Computer Science, Vols. 1–6.* Clarendon Press, Oxford, 1992.

[AZ95] E. Astesiano and E. Zucca. D-oids: a model for dynamic data types. *Mathematical Structures in Computer Science*, 5:257–282, 1995.

[Bee95] C. Beeri. Recent trends in data type specification. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Bulk Types and Query Language Design*, volume 906 of *Lecture Notes in Computer Science*. Springer, 1995.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 2nd edition, 1994.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, S. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes, editors. *Object-oriented Development – The Fusion Method*. Prentice Hall, 1994.

[Cal96] C. Caleiro. Distributed object communities. Master's thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, Av. Rovisco Pais, 1096 Lisboa Codex, Portugal, 1996. Supervised by A. Sernadas. In Portuguese.

[CSS94] J.F. Costa, A. Sernadas, and C. Sernadas. Object inheritance beyond subtyping. *Acta Informatica*, 31:5–26, 1994.

[Dav65] M. Davis, editor. *The Undecidable*. Raven Press, New York, 1965.

[DDPW94] E. Dubois, Ph. Du Bois, M. Petit, and S. Wu. ALBERT: A formal agent-oriented requirements language for distributed composite systems. In E. Dubois, P. Hartel, and G. Saake, editors, *Proc. Workshop on Formal Methods for Information System Dynamics (CAiSE'94-Workshop)*, pages 25–39. University of Twente, 1994. Technical Report.

[DE95] G. Denker and H.-D. Ehrich. Action reification in object-oriented specification. In R.J. Wieringa and R.B. Feenstra, editors, *Information Systems—Correctness and Reusability, Selected Papers from the ISCORE Workshop'94*, pages 103–118. World Scientific, 1995.

[Den96a] G. Denker. Reification—changing viewpoint but preserving truth. In O.-J. Dahl, M. Haveraan, and O. Owe, editors, *Recent Trends in Data Types Specification, Proc. 11th Workshop on Specification of Abstract Data Types joint with the 8th General COMPASS Meeting*, volume 1130 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 1996.

[Den96b] G. Denker, editor. *Verfeinerung in objektorientierten Spezifikationen: Von Aktionen zu Transaktionen*. Reihe DISDBIS, Band 6. infix-Verlag, Sankt Augustin, 1996.

[DH97] G. Denker and P. Hartel. TROLL—an object oriented formal method for distributed information system design: Syntax and pragmatics. Informatik-Berichte 97-03, Technische Universität Braunschweig, 1997.

[DMN67] O.-J. Dahl, B. Myrhaug, and K. Nygaard, editors. SIMULA *67, Common Base Language*. Norwegian Computer Center, Oslo, 1967.

[ECSD98] H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for specifying concurrent information systems. In G. Saake and J. Chomicki, editors, *Logics for Databases and Information Systems*. Kluwer Publ. Comp., 1998. To appear.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing systems as object communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT'93)*, volume 668 of *Lecture Notes in Computer Science*, pages 453–467. Springer, 1993.

[EGS90] H.-D. Ehrich, J. A. Goguen, and A. Sernadas. A categorial theory of objects as observed processes. In W.P. de Roever J.W.̃de Bakker and G. Rozenberg, editors, *Proc. REX/FOOL Workshop*, volume 489 of *Lecture Notes in Computer Science*, pages 203–228. Springer, 1990.

[EH96] H.-D. Ehrich and P. Hartel. Temporal specification of information systems. In A. Pnueli and H. Lin, editors, *Proc. Intl. Workshop in Honor of Chih-Sung Tang*, pages 43–70, Singapore, 1996. World Scientific.

[EJD93] H.-D. Ehrich, R. Jungclaus, and G. Denker. Object roles and phases. In U.W. Lipeck and G. Koschorreck, editors, *Proc. Intern. Workshop on Information Systems – Correctness and Reusability IS-CORE'93*, pages 114–121. University of Hannover, 1993. Technical Report No. 01/93.

[EJDS94] H.-D. Ehrich, R. Jungclaus, G. Denker, and A. Sernadas. Object-oriented design of information systems: Theoretical foundations. In J. Paredaens and L. Tenenbaum, editors, *Advances in Database Systems, Implementations and Applications*, CISM Courses and Lectures No. 347, pages 201–218. Springer, 1994.

[ES90] H.-D. Ehrich and A. Sernadas. Algebraic implementation of objects over objects. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness"*, volume 430 of *Lecture Notes in Computer Science*, pages 239–266. Springer, 1990.

[ES95] H.-D. Ehrich and A. Sernadas. Local specification of distributed families of sequential objects. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop*, volume 906 of *Lecture Notes in Computer Science*, pages 218–235. Springer, 1995.

[ESD93] Espírito Santo Data Informática (ESDI), Lisbon. OBLOG CASE V1.0 – *The User's Guide*, 1993.

[FM92] J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing*, 4:239–272, 1992.

[FS97] M. Fowler and K. Scott. UML *Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, New York, 1997.

[FW93] R.B. Feenstra and R. Wieringa. LCM 3.0: A language for describing conceptual models – syntax definition. Rapport IR-344, Vrije Universiteit Amsterdam, 1993.

[GG94] Dov Gabbay and Franz Guenthner, editors. *What is a Logical System?* Oxford University Press, 1994.

[GM87] J. A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B.S̃hriver anf P.W̃egner, editor, *Research Direction in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. The temporal analysis of fairness. In *Proc. 7th ACM Symp. on Principles of Programming Languages*, pages 163–173, 1980.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[GS95] J.A. Goguen and A. Socorro. Module composition and system design for the object paradigm. *Journal of Object Oriented Programming*, 7(14):47–55, 1995.

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Research Report SRI-CSL-88-9, SRI International, 1988.

[Har97] P. Hartel. *Konzeptionelle Modellierung von Informationssystemen als verteilte Objektsysteme*. Reihe DISDBIS, Band 6. infix-Verlag, Sankt Augustin, 1997.

[HJ95] P. Hartel and R. Jungclaus. Modeling business processes over objects. *Intl. Journal of Intelligent and Cooperative Information Systems*, 4:165–188, 1995.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering*. Addison-Wesley, Reading, MA, 1992.

[JSHS96] Ralf Jungclaus, Gunter Saake, Thorsten Hartmann, and Cristina Sernadas. TROLL—A language for object-oriented specification of information systems. *ACM Transactions on Information Systems*, 14:175–211, 1996.

[KG76] A. Kay and A. Goldberg. *Smalltalk-76 Instruction Manual*. Xerox PARC, 1976.

[LcP90] Huimin Lin and Man chi Pong. Modelling multiple inheritance with colimits. *Formal Aspects of Computing*, 2:301–311, 1990.

[LG91] R. Loogen and U. Goltz. Modelling nondeterministic concurrent processes with event structures. *Fundamenta Informaticae*, 14:39–74, 1991.

[LMRT91] K. Lodaya, M. Mukund, R. Ramanujam, and P.S. Thiagarajan. Models and logics for true concurrency. In P.S. Thiagarajan, editor, *Some Models and Logics for Concurrency. Advanced School on the Algebraic, Logical and Categorical Foundations of Concurrency, Gargnano del Garda*, 1991.

[LRT92] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. Temporal logics for communicating sequential agents: I. *International Journal of Foundations of Computer Science*, 3:117–159, 1992.

[LS95] F. Laroussinie and Ph. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148:303–324, 1995.

[LT87] K. Lodaya and P.S. Thiagarajan. A modal logic for a subclass of event structures. In Th. Ottmann, editor, *Proc. 14th Intl. Colloq. on Au-*

tomata, Languages and Programming, volume 267 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 1987.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes93] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[MP92] Z. Manna and A. Pnueli, editors. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[MP95] Z. Manna and A. Pnueli, editors. *Temporal Verification of Reactive Systems—Safety*. Springer, 1995.

[MT92] M. Mukund and P.S. Thiagarajan. A logical characterization of well-branching event structures. *Theoretical Computer Science*, 96:35–72, 1992.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 13:85–108, 1981.

[NRT92] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96:3–33, 1992.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 19th Annual Symposium on Foundations of Computer Science*, pages 46–57, New York, 1977. IEEE.

[Ram96] R. Ramanujam. Locally linear time temporal logic. In *Proc. 11th Annual IEEE Symposium on Logics in Computer Science*, pages 118–127, New York, 1996. IEEE Computer Society.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[RS92] L. Rapanotti and A. Socorro. Introducing FOOPS. Report PRG-TR-28-92, Programming Research Group, Oxford University Computing Laboratory, 1992.

[SFSE88] A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich. Abstract object types: A temporal perspective. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Colloq. on Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 324–350. Springer, 1988.

[SHJE94] G. Saake, T. Hartmann, R. Jungclaus, and H.-D. Ehrich. Object-oriented design of information systems: `TROLL` language features. In J. Paredaens and L. Tenenbaum, editors, *Advances in Database Systems, Implementations and Applications*, CISM Courses and Lectures No. 347, pages 219–245. Springer, 1994.

[SJE92] G. Saake, R. Jungclaus, and H.-D. Ehrich. Object-oriented specification and stepwise refinement. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP Transactions C: Communication Systems, Vol. 1: Proc. Open Distributed Processing*, pages 99–121. North–Holland, 1992.

[SJH93] G. Saake, R. Jungclaus, and T. Hartmann. Application modelling in heterogeneous environments using an object specification language. *Intl. Journal of Intelligent and Cooperative Information Systems*, 2:425–449, 1993.

[SNW93] V. Sassone, M. Nielsen, and G. Winskel. A classification of models for concurrency. In E. Best, editor, *Proc. CONCUR'93*, pages 82–96, 1993.

[SR94] A. Sernadas and J. Ramos. The GNOME language: Syntax, semantics and calculus. Technical report, Instituto Superior Técnico, Lisboa, 1994.

[SSC95] A. Sernadas, C. Sernadas, and J.F. Costa. Object specification logic. *Journal of Logic and Computation*, 5:603–630, 1995.

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-oriented specification of databases: An algebraic approach. In P. Hammerslay, editor, *Proc. 13th Intl. Conference on Very Large Databases (VLDB'87)*, pages 107–116, Palo Alto, 1987. Morgan–Kaufmann.

[Tan94] C.S. Tang. A temporal logic language oriented toward software engineering – an introduction to the XYZ sytem. *Chinese Journal of Advanced Software Research*, 1:1–29, 1994.

[Thi94] P. S. Thiagarajan. A trace based extension of linear time temporal logic. In *Proc. 9th Annual IEEE Symposium on Logic in Computer Science*, pages 438–447. IEEE Computer Science Press, 1994.

[Vaa89] F. Vaandrager. A simple definition for parallel composition of prime event structures. Report CS-R8903, CWI Amsterdam, 1989.

[Weg89] P. Wegner. Learning the language. *Byte*, 14:245–253, 1989.

[Win84] G. Winskel. Synchronization trees. *Theoretical Computer Science*, 34:33–82, 1984.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In [AGM92], volume 4, pages 1–148. 1995.