Logic and Software Engineering, A. Pnueli and H. Lin (eds), World Scientific, Singapore 1996, 43-70

Temporal Specification of Information Systems[*]

Hans-Dieter Ehrich and Peter Hartel

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

Abstract. Information systems are open, reactive, and often distributed systems that maintain persistent data. The TROLL and OMTROLL languages aim at specifying information systems on a high level of abstraction, supported by tools integrated in the TBench. The development is rooted in abstract data types, conceptual modeling, behavior modeling, specification of reactive systems, and concurrency theory. The approach taken is object-oriented. Sequential object behavior is specified using linear-time temporal logic. Specifications are composed by structuring mechanisms like inheritance, generalization and aggregation. For specifying interaction in concurrent aggregations, we suggest an extended logic called distributed temporal logic which is based on n-agent logic. Labelled event structures serve as a denotational behavior model for giving interpretation.

1 Introduction

Information systems are reactive systems with the capability of maintaining and utilizing large amounts of data. We are developing a language and a system for designing information systems, using formal methods based on a sound theory. Ideas and concepts from abstract data types, conceptual modeling, behavior modeling, specification of reactive systems, and concurrency theory are adopted.

The approach is object-oriented: a software system is considered to be a dynamic collection 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 types and classes, facilitating structuring and reuse of software.

Moreover, a practical specification and development technique must offer an appropriate blend of formal *and informal* techniques. The latter should

[*] This work was partly supported by the EU under ESPRIT-III BRA WG 6112 COMPASS (COMPrehensive Algebraic approach to System Specification and development).

preferably be based on a graphical symbolism offering intuitive access to the concepts involved.

Our approach is based on experiences in developing the OBLOG family of languages and their semantic foundations [?]. OBLOG is being developed into a commercial product [?]. In the academic realm, there are two related developments: TROLL [?, ?, ?, ?] and GNOME [?]. The common semantic basis is temporal logic.

In the TROLL project, the TROLL and OMTROLL languages and the TBench system are being developed. The languages are designed for specifying object systems on a high level of abstraction. TROLL is a textual and OMTROLL is a graphical variant of the language. The latter borrows concepts from OMT [?]. The TBench is an integrated software development environment for the specification and animation of TROLL or OMTROLL specifications, respectively.

There are other approaches to developing object specification languages with a sound theoretical basis. The ones most closely related to ours are FOOPS [?, ?, ?] and MAUDE [?]. FOOPS is based on OBJ3 [?] which is in turn based on equational logic. MAUDE is based on rewriting logic that is a uniform model of concurrency [?].

Many other language projects work on related ideas, e.g., OOZE [?], ALBERT [?], LCM [?] and ETOILE [?]. The languages and their underlying models and theories are quite diverse.

The TROLL language project is based on linear temporal logic. This is the simplest logic that can express not only safety but also liveness and fairness properties. Since the pioneering work of Pnueli [?], temporal logic is widely accepted as a suitable formalism for giving axiomatic descriptions of system dynamics on a high level of abstraction. For recent textbooks, see [?, ?].

Temporal logic has also been used as the backbone of a large-scale programming system developed and put into practice by Prof. Tang in China [?].

Among the many temporal logics that have been suggested and investigated, we choose an as simple as possible one, namely a propositional linear-time logic similar to PTL [?] that goes back to [?] where also a sound and complete proof system is given. Interpretation is defined in a general way that goes beyond linear-time models. The purpose is to prepare for an extension towards specifying distributed systems displaying true concurrency.

Distributed reactive systems are more complex and difficult to specify, analyse, implement and reconfigure than other kinds of systems. Their foundations are not well understood. In order to capture true concurrency while staying close to temporal logic specification, we developed an extension of temporal logic called *Distributed Temporal Logic.* The basic idea is to use temporal logic only locally and to include temporal statements about other objects. The logic is based on n-agent logic [?, ?, ?, ?] but we deviate in essential respects: we have more elementary temporal operators, and our interpretations are different.

The paper is organized as follows. Section 2 gives a brief overview of the TROLL and OMTROLL languages and the TBench development tool. Section 3 gives a brief account of the underlying theory. Its goal is to provide a basis for complete semantic description of TROLL and its foreseeable extensions, especially towards full concurrency. An integration of these parts in a complete syntactic and semantic description of TROLL is beyond the scope of this paper.

2 TROLL

TROLL [?, ?] is a formal language for the specification of object systems on a high level of abstraction. The basic ideas and concepts of TROLL can be characterized as follows:

- The basic building blocks of information systems are objects [?].
- Objects encapsulate an internal state that can be modified by means of local events only. Part of the object state is observable through attributes.
- Objects are classified in classes and have unique identities which are independent of the observable state.
- An object is described by a set of attributes and actions. The object evolution is specified by possible traces of occurrences of actions and communications with other objects. A system specification is composed of a set of object class and data type specifications.
- Objects are related in various ways. Roles describe the temporal specialization of objects. Such objects may have additional properties and/or a restricted behavior. Static specialization is considered to be a permanent role. Objects may be components of other objects that themselves are called composite objects. Separately defined objects may be connected through global interactions.

In the past few years, there has been considerable activity in the area of object-oriented analysis. OO analysis methods [?, ?, ?] have been proposed mainly with a background in software development practice. They do not reach the level of formality achieved by formal specification languages. However, they come along with a set of rules and guidelines for the conceptual modeling of systems and they propose graphical representations for object-oriented concepts. This suggests that a marriage between formal and informal approaches to modeling and development could be fruitful. The formal approaches help in discovering and eliminating ambiguity and vagueness in the informal representation techniques. Practical analysis methods may contribute to this marriage with their methodological guidelines and graphical notations. They help to make formal languages fit for use. Finally, a formal specification language combined with a practical analysis method makes powerful techniques like automated theorem proving, animation and consistency checking available for CASE tools used in analysis. We decided to marry the Object Modeling Technique (OMT) [?] and the TROLL language. The resulting analysis approach is named OMTROLL[?, ?].

A set of alternative models have to be elaborated and evaluated in the conceptual modeling phase. The validation of the adequacy of a model is an important task in this phase. Specification execution is a powerful tool for this purpose [**?**]. The required malleability and executability of specifications results in the demand for sophisticated tools which support the developers in the analysis phase. We developed a first prototype of a specification environment, called TBench [**?**].

This section is subdivided into three parts. We will introduce the concepts and features of TROLL in the first subsection. Then we will explain the adaptation of the popular object-oriented method OMT for OMTROLL. The last subsection will present some issues of the specification environment TBench which supports the modelling of complex information systems with OMTROLL and TROLL.

2.1 TROLL by example

We illustrate the TROLL language by an example. We are going to specify some aspects of the library of our computer science department. Our fragment of the library world consists of the librarian and the library itself which is composed of the documents, copies of documents and a file of registered library users. There may be several copies of every document in the library. Users may borrow these copies. The library has to trace the frequency a document is borrowed. Users may keep documents for at most 6 weeks until they have to return it or they renew it for another 6 weeks. However, borrowed copies must be returned sometime. Documents are either books or journals. Anyhow, every document has a title and is described by at least 3 keywords. The library issues a unique number for every document and every copy gets an inventory number. The librarian is responsible for registration of new library users. She has to document whenever users borrow, return or renew documents. For sake of simplicity we reduced the example to this part of the library world.

Example 1. We will start by the detailed specification of document objects.

```
template document
  attributes
    docNo : nat constant;
    title : string;
    kind:{book, journal} constant;
    keywords:listOf(string);
    noOfLendings:nat initial 0;
  actions
    newBook(t:string, k:listOf(string), no:nat, authors:listOf(string))
      birth
      changing docNo:= no, title:=t, kind:=book, keywords:=k;
    borrow
      changing noOfLendings:=noOfLendings+1;
    ...
  constraints length(keywords) ≥ 3;
end template document.                                                    □
```

The local properties of instances are specified in the *template*. Attributes and actions define the local signature of objects.

Attributes are typed and may be further characterized—here the initial value of the attribute noOfLendings is 0 and the attribute docNo is constant. Attributes may be derived from the values of other attributes. An example below will illustrate this feature.

Actions are used to describe the behavior of objects. They represent the possible state transitions of an object. The action newBook is qualified to be the birth action of a document that is a book. The occurrence of a birth action creates a new object. Actions may be parameterized. This allows for the exchange of data between objects during communications. The effects of an action may depend on the actual values of parameters and the local object state. They are specified in the changing section. A valuation expression describes the new attribute value that is observable after the action occurred. An implicit frame rule states that all attributes that are not changed by an action maintain their current value.

Object behavior is described not only by the changing operations but also in terms of allowed object life cycles. We may restrict the permitted life cycles by defining state *constraints*, e.g., the number of keywords for every document has always to be three or more. Using future tense temporal logic [?] we can define more complex constraints as the following example will show. Another way to restrict the occurrence of an action is by defining an *enabling condition.* An action may occur if its enabling condition is fulfilled. We may use predicate logic and past tense temporal logic [?] to specify enabling conditions of actions.

Example 2. Use of temporal logic in the specification of object behavior:

```
template copy
  attributes
    noOfCopy : nat;
    isCopyOf : |Document|;
  actions
    variables u:|User|, d:date;
    enter(copyOf:|Document|, no:nat) birth
      changing noOfCopy:=no; isCopyOf:=copyOf;
    borrow(usr:|User|, d:date);
    return
      enabled sometime(occurred(borrow(u,d)));
    ...
  constraints
    occurred(borrow(u,d)) implies henceforth(occurred(return));
end template copy.                                              □
```

Besides the commonly used past tense predicates *sometime*, *always*, *previous* and their future tense counterparts, we introduced a special predicate *occurred*. It has as argument an action term. *occurred*(act) is true in the states reached by the action act.

Attributes may refer to other objects. For this purpose every object class introduces a special data type. We specified the attribute isCopyOf in order to state that every copy is a copy of a certain document. The data type `|Document|` stands for an object identity type, i.e., the identifiers for objects of class Document.

The template defines the properties of an object instance. To come from a template to an *object class* we have to associate an *identification mechanism* to it. Documents are identified by their numbers, i.e., the value of the docNo attribute.

Example 3. Consider the following specification of the object class Document.

```
object class Document
  identification ByNo(docNo:nat);
  template documents
end object class Document.                                    □
```

For modeling complex object systems it is not sufficient to specify single objects and to classify them into object classes. Objects of different classes may be closely related and form a *complex object.* A complex object incorporates other objects as parts. Thus, the library is composed of document, copy and user objects.

Example 4. The specification of a complex object class Library.

```
object class Library
  single
  attributes
    nextDocNo:nat;
    ...
  components
    books(no:nat):document;
    copies(inventoryNo:nat):copy;
    users(name:string):user;
  actions
    borrow(invNo:nat, usr:|User|, d:date)
      calling copies(invNo).borrow(usr, d)
    newBook(t:string, k:listOf(string), no:nat, authors:listOf(string))
      changing nextDocNo:= nextDocNo+1;
      calling books(nextDocNo).newBook(t,k,nextDocNo,authors);
     ...
end object class Library.                                     □
```

The specification of components implies the inclusion of the component objects into the complex object. This means that the local signature of the latter is enriched with the signature of the former. We may use attribute and action symbols of both objects in the composing object. In our example, a library object calls the borrow action of a specific copy. Calling represents asymmetric, synchronous communication between the aggregating object and its components or between the components in composition.

A very important mechanism to reflect taxonomic relationships is the *role concept.* A role defines a possible *temporary specialization* of an object (which is then said to play the role). It may reveal more properties in addition to the already known ones or may restrict the possible behavior further. As an example, we introduce BorrCopy for copy objects that have been borrowed by a user. A borrowed copy has a special attribute that refers to the library user who borrowed that copy and may perform the action of renewal. A copy becomes a BorrCopy when the copy is borrowed and it ends this role in the moment the copy is returned. However, with the death of the role, the object does continue to exist as a copy.

Example 5. Borrowed copies are a special role played by copies.

```
object class BorrCopy
  role of Copy
  attributes
    by : |User|;
    until : date;
  actions
    borrow(usr:|User|, today:date) birth
      changing by:=usr, until:=addWeeks(today, 6);
    renewal(d:date)
      changing until:=addWeeks(until, 6);
    return death;
end object class BorrCopy.
```

The birth action borrow and the death action return are inherited from Copy. □

A *static specialization* is regarded to be a special role, played for the whole lifetime of an object. The classification must depend on a constant property. Consider books as an example. They are special documents that may have an ISBN code and a list of authors. We specialize the base class Documents to Books.

Example 6. Books are special documents.

```
object class Book
  role of Document derived Document.kind = book
  attributes
    isbn : string;
    authors : listOf(string);
  actions
    newBook(t:string, k:listOf(string), no:nat, a:listOf(string)) birth
      changing authors:=a; isbn:="";
    setISBN(code:string)
      changing isbn:=code;
    ...
end object class Book.
```

□

As a specialized object, a book inherits all properties of its base object. It may also add additional properties. A book has the two new attributes authors and isbn.

Object composition allows one to relate different objects closely. TROLL allows one to loosely connect different objects via communication channels. Consider for example that we want to model the fact that a librarian wants to enter a new book. For this purpose, the librarian has to call an action of the library, which issues a new document number for that book and creates a new book instance as a component of the library. To allow for the specification of such communication channels we introduced the *relationship* construct. In this way, communication does not get buried inside the specifications, and reuse of template specifications is enabled.

Example 7. This example illustrates the communication between a customer and the bank.

```
relationship LibrarianWithLibrary between Librarian l, Library lib
  variables t:string, k:listOf(string), n:nat, a:listOf(string);
  interaction
      l.newBook(t,k,n,a)  >>  lib.newBook(t,k,n,a)
end relationship  LibrarianWithLibrary.
```
□

The data flow between the caller and the called object is defined together with the calling relation.

In this section we gave a survey of the concepts and features of the TROLL language. A complete definition of TROLL is given in [**?**].

2.2 OMTROLL

We will start by giving a brief overview of the Object Modeling Technique. For a detailed discussion of OMT the reader is referred to [**?**].

An OMT system model is threefold. It consists of an object model, a dynamic model, and a functional model. The *object model* represents the structure of the objects in the system. The diagram technique used is a variation and extension of ER representation techniques. The *dynamic model* represents aspects of the system concerned with control, including time, sequencing of operations, and interaction of objects. For each class whose instances have interesting behavior, a state chart [**?**] is made that describes all or part of the behavior of one object of a given class. OMT supports several mechanisms to model interactions between objects. However, OMT does only know asynchronous communication. The *functional model* defines the meaning of operations by showing how values are transformed by the system. The representation technique used is the data flow diagram (DFD).

The most important observation is that the three OMT models are poorly integrated. The functional model takes a global view of the systems and is not object centered. It is orthogonal to the two other models and does not really suit for a truly object-oriented approach.

Due to the popularity of OMT we tried to save as much as possible of the OMT notation in the definition of omTroll. Where necessary, we improved notations or even introduced new notations. Graphical notations are regarded to be auxiliary notations to make Troll specifications better manageable. This implies that omTroll specifications are used for the modeling of the systems structure, relationships between the system components and behavioral aspects. The specification of details, like constraints, preconditions, and updates remain textual.

omTroll is threefold like OMT. It consists of three models, an *object model*, a *behavior model* and a *communication model*. The communication model substitutes the functional model and eliminates one major drawback of OMT.

The omTroll notation for the *object model* is not very different from the OMT notation. A feature that we added is the *interaction association* which is expressed by a flash between the interacting object classes.

Figure 1 shows the overall view of the object model for the library world.

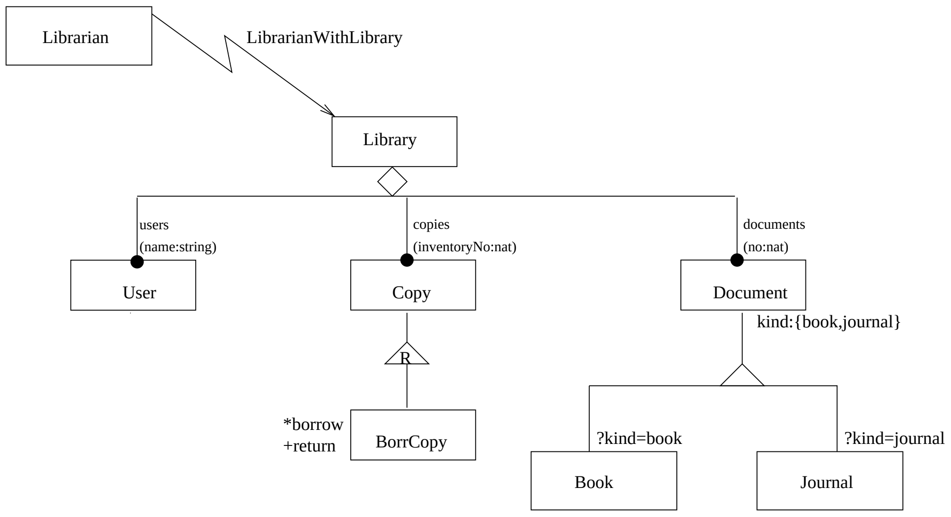


Fig. 1. Object model of the library world

Object classes are depicted by boxes, like in OMT we have composite classes (like the class Library) and specialization classes to define taxonomic relationships between object classes (like Document, Book, Journal). Associations can have cardinalities – here, a black dot denotes that there are many links. An extension is the fact that Troll supports roles. In Troll roles are the general case of taxonomic relationships – thus, roles are represented by a triangle containing an R (like BorrCopy).

Each object class definition is specified according to the OMT notation. A class definition defines the signature for the attributes and the events (or operations as they are called in OMT). Figure 2 gives an example for the declarations of two object classes in the example object model.

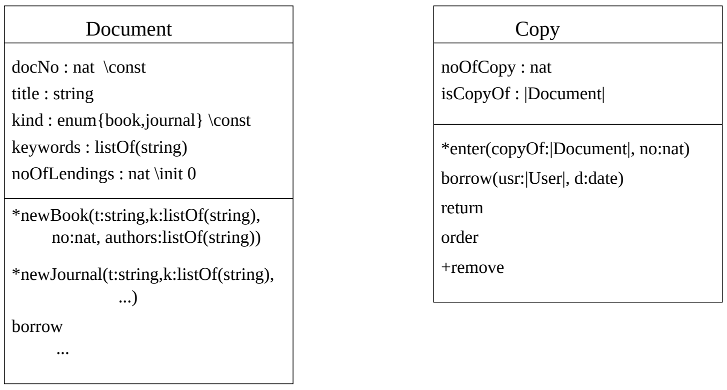

Document
docNo : nat \const
title : string
kind : enum{book,journal} \const
keywords : listOf(string)
noOfLendings : nat \init 0
*newBook(t:string,k:listOf(string), no:nat, authors:listOf(string))
*newJournal(t:string,k:listOf(string), ...)
borrow
...
Copy
noOfCopy : nat
isCopyOf : |Document|
*enter(copyOf:|Document|, no:nat)
borrow(usr:|User|, d:date)
return
order
+remove

Fig. 2. Document Class, Copy Class

In the *dynamic model*, the behavior over time of objects is defined. In OMT, state diagrams are used for this purpose (see figure 3). TROLL does not support the explicit definition of states. Thus, in OMTROLL states are used for auxiliary purposes – of more importance are the state transitions in OMTROLL. The states in the state diagram constrain the possible next transitions. A dot depicts an initial state, a circled dot depicts the postmortem state, which means that the object has died. In OMTROLL we may furthermore define additional enabling conditions for actions (i.e., possible transitions). They are noted in textual form according to the TROLL syntax.

The dynamic model (see figure 3) depicts the behavior of copies. Before a copy may be borrowed, it has to be ordered at the librarian desk. Afterwards, the copy may be borrowed by the user. However, an order may be canceled. Copies that are damaged or that are never requested may be removed from the library.

Since OMT does not support the explicit specification of interactions between objects, we have introduced a new type of diagram in OMTROLL to capture the *communication model*. We call those diagrams *communications diagrams* (see figure 4). The boxes represent object classes. Arrows between the classes represent communication. The label on an arrow pointing to a class indicates the action that is called. There may be more than one arrow originating in a class box. Preconditions may be attached to connections and auxiliary

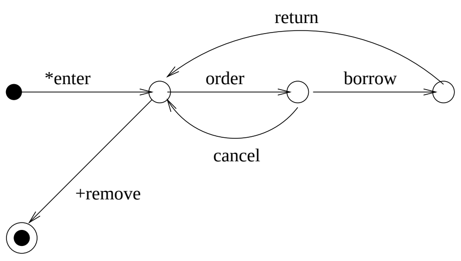


Fig. 3. Dynamic model of Copy

variables may be introduced.

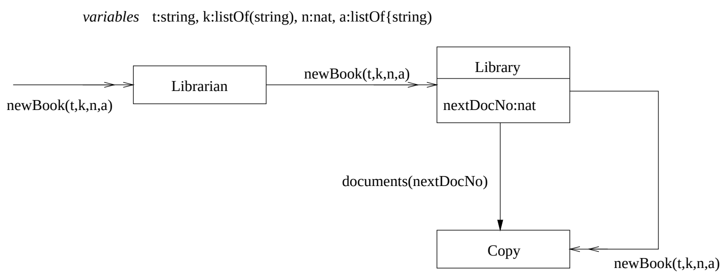


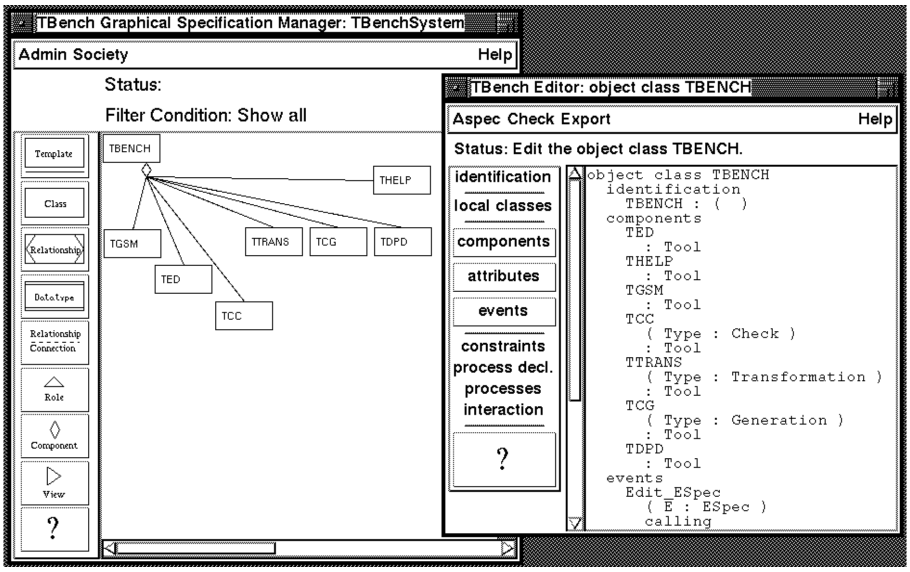
Fig. 4. Communication model

2.3 TBench

The TBench is an integrated software development environment for the specification and animation of TROLL specifications. It consists of several, loosely coupled tools, each supporting a specific task in the phase of conceptual modeling. The set of tools can be divided into two functional groups. One for the management and the editing of large specification documents and another one for the prototyping of TROLL specifications, i.e., the generation of executables from specifications.

The tools which deal with the manipulation of specification documents are:

– The *TGSM* (TBench Graphical Society Manager) that supports the graphical representation of OMTROLL(see figure 5).
– The *TED* (TBench Editor) for the textual specification with TROLL (see figure 5).
– The *TCC* (TBench Consistency Checker) that checks the consistency of specification documents.



```
object class TBENCH
  identification
    TBENCH : (  )
  components
    TED
      : Tool
    THELP
      : Tool
    TGSM
      : Tool
    TCC
      ( Type : Check )
      : Tool
    TTRANS
      ( Type : Transformation )
      : Tool
    TCG
      ( Type : Generation )
      : Tool
    TDPD
      : Tool
  events
    Edit_ESpec
      ( E : ESpec )
      calling
```

Fig. 5. TGSM and TED

The integration of the TGSM and the TED allows for a smooth transition from one presentation to another while the specification environment controls the integrity of the textual and graphical specifications.

The TCC checks the consistency of the specification documents, e.g., it checks the existence of classes and properties of classes which are referred, it analyses possible conflicts in the interactions between objects, etc.

The prototyping of TROLL specifications is supported by two tools:

- The *TTRANS* (TBench Transformator) that maps high level TROLL concepts which are not per se executable into an operational language TROLL-kernel.
- The *TCG* (TBench Code Generator) that maps TROLL-kernel specifications to specific implementation platforms, e.g., C++, relational database, TCL/TK etc.

The TTRANS works semi-automatically. The developer has to guide the transformation process from a TROLL-specification into a TROLL-kernel specification. Concepts which have to be transformed are temporal logic formulae, global integrity constraints and global object communications. TROLL-kernel is an executable subset of TROLL and contains only concepts which are coherent with common programming languages.

The TCG works automatically. The result of the generation process is an executable prototype of the specified system.

The integration of the different tools of the TBench is managed by the *TIM* (TBench Integration Manager) and the *TDPD* (TBench Development Process Driver). All tools and integration aspects of the TBench are themselves specified in Troll. Their specifications are contained in the TBench system and allow for an easy customization of the development environment.

The TBench is a prototype of a software development environment and not a commercial CASE tool. Therefore, we did not pay attention to aspects like multi-user support or versioning of specification documents. The goal of the development of the TBench was a feasibility study of a prototyping environment based on Troll and omTroll.

3 Fundamentals

In the Troll project, we have taken great care to work on a sound theoretical basis. There is a nearly complete formal semantic description of an earlier Troll version [?] using OSL [?], an object specification logic based on linear temporal logic.

While language and implementation evolve, the foundations are being developed as well. The goal is not only to provide a basis for complete semantic description of Troll as it is now or is currently being developed. The goal is to provide a broad and general enough theory for foreseeable Troll extensions.

One such extension is the incorporation of distribution issues. Distribution implies full concurrency that cannot be specified in temporal logic, and the simple life cycle model underlying Troll is not adequate any more.

In what follows, we give a short account of the state of the theoretical work in the Troll project. Its application to describing Troll semantics is outside the scope of this paper, but the experienced reader will easily be able to relate some of the language issues described above with some of the theoretical issues described below.

3.1 Signatures and interpretation

Signatures. A *signature* is a pair $\Sigma = (S, \Omega)$ where S is a set of *sorts*, and $\Omega = \{\Omega_{x,s}\}_{x \in S^*, s \in S}$ is an $S^* \times S$-indexed family of sets of *operation symbols.*

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..

Data and objects are distinguished by assuming that the sorts S are subdivided into disjoint subsets of *data sorts* S_D and *object sorts* S_O, i.e., $S = S_D + S_O$.

Interpretation. Interpretation is given in a Σ-algebra U. The intended interpretation of a data sort $d \in S_D$ in U is a set d_U of data elements that is the carrier set of d in U.

The intended interpretation of an object sort $s \in S_O$ in U is a pair $s_U = s_U^{id}.s_U^{bv}$ where s_U^{id} is a set of *identities* and s_U^{bv} is the *behavior* of objects of sort s. s_U represents a set of *object instances* $i.s_U^{bv}$, each given by an object identity $i \in s_U^{id}$.

The operators $\omega : x \to s$ are interpreted as usual by operations $\omega_U : x_U \to s_U$ where $x_U = s_{1U} \times \ldots \times s_{nU}$ if $x = s_1 \ldots s_n \in S^*$.

Data and object types. A *data type* is a carrier set s_U of a data sort $s \in S_D$ within a Σ-algebra U, together with the operations defined on s_U. Correspondingly, an *object type* is a carrier set s_U of an object sort $s \in S_O$, together with the operations defined on s_U.

Formally, all instances of an object type have the same behavior. It is possible, however, that different parts of this behavior are relevant for different objects of the type, introducing some heterogeneity.

For simplicity, we assume that there are no operations on object types. Thus, object types are just carrier sets $r_U, s_U, \ldots$ in U where $r, s, \ldots \in S_O$.

Object types are related by *object type morphisms* preserving structure and behavior. Preserving behavior means that there is a *behavior morphism* between related behaviors, cf. definition 8. Behaviors and their morphisms form a category **bhv** of behaviors. The intuition is that a behavior morphism reflects behavior inheritance. As [**?**] points out, there are two kinds of inheritance relationships that go in opposite directions: *is-a* inheritance and *as-a* inheritance (cf. subsection 3.3). Correspondingly, we define object type morphisms in two versions that are also useful for expressing other relationships like hiding and synchronization (cf. subsection 3.3) and relationships between various kinds of composite object types and their constituent types (cf. subsection 3.3).

Definition 1. Let Σ be a signature, U a Σ-algebra, and r_U, s_U object types. Let **bhv** be a category of behaviors.

(1) A *(direct) object type morphism* $\mu : r_U \overset{\to}{\to} s_U$ is a pair $\mu = (\mu^{id}, \mu^{bv})$ where $\mu^{id} : r_U^{id} \to s_U^{id}$ is a total map, and $\mu^{bv} : r_U^{bv} \to s_U^{bv}$ is a behavior morphism in **bhv**.

(2) A *reverse object type morphism* $\mu : r_U \overset{\to}{\leftarrow} s_U$ is a pair $\mu = (\mu^{id}, \mu^{bv})$ where $\mu^{id} : r_U^{id} \to s_U^{id}$ is a total map, and $\mu^{bv} : s_U^{bv} \to r_U^{bv}$ is a behavior morphism in **bhv**.

The category of object types and direct object type morphisms based on a given behavior category **bhv** is denoted by **ot(bhv)**. Essentially, the properties of **ot(bhv)** are determined by those of **bhv** which we comment on below.

Object behavior. As a behavior model for objects, we use labelled event structures.

Definition 2. A *(discrete prime) event structure* is a triple $E = (Ev, \to^*, \#)$ where Ev is a set of events and $\to^*, \# \subseteq Ev \times Ev$ are binary relations called *causality* and *conflict*, respectively. Causality $\rightarrow^*$ is a 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 *concurrent*, $e\, co\, e'$ iff $\neg(e \rightarrow^* e' \vee e' \rightarrow^* e \vee e \# e')$.

The standard notation for causality is $\leq$ but we want to emphasize that causality is the reflexive and transitive closure of a base relation $\rightarrow$ of "step causality". In the sequel, order-theoretic notions refer to causality. The finiteness condition is a temporal reachability condition: only events $e \in Ev$ that may happen within finite time after system start are taken into consideration.

Definition 3. Let $E = (Ev, \rightarrow^*, \#)$ be an event structure. A *configuration* in E is a downward-closed conflict-free set of events $C \subseteq Ev$, i.e., the following conditions hold true. (1) $\forall e \in C : \downarrow e \subseteq C$, and (2) $\forall e_1, e_2 \in C : \neg(e_1 \# e_2)$. A *life cycle* in E is a maximal configuration in E. The sets of configurations and life cycles in E are denoted by $\mathcal{C}(E)$ and $\mathcal{L}(E)$, respectively.

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. If E has just one life cycle, i.e., E is conflict-free, then we call E itself a life cycle. 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 4. For $i = 1, 2$, let $E_i = (Ev_i, \rightarrow_i^*, \#_i)$ be event structures. An *event structure morphism* $g : E_1 \rightarrow E_2$ is a partial function $g : Ev_1 \rightharpoondown Ev_2$ such that, for every configuration $C \in \mathcal{C}(E_1)$, the following conditions hold true. (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 $g : E_1 \rightarrow E_2$ is often used to express how behavior in E_1 coincides 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.

Remark. The category **ev** of event structures is complete and has coproducts. Products are useful for modeling concurrent aggregation, and coproducts are useful for modeling generalization [**?**].

Sequential event structures. We are especially interested in event structures having no concurrency at all.

Definition 5. An event structure $E = (Ev, \rightarrow^*, \#)$ is called *sequential* iff there is no concurrency in E, i.e., its concurrency relation co is empty.

A sequential event structure admits a timing function $\tau : Ev \to \mathsf{N}$ defined by $\tau(e) = 0$ iff e is minimal, and $\tau(e') = \tau(e) + 1$ iff $e \to e'$ holds.

A *sequential* morphism $g : E_1 \to E_2$ between sequential event structures is a total event structure morphism. Sequential event structure morphisms map traces in a bijective way. Another way to put this is that sequential morphisms preserve time in the sense that $\tau(e) = \tau(g(e))$ for every event $e \in Ev_1$.

Remark. The category **evs** of sequential event structures with sequential morphisms is complete and cocomplete. Colimits in **evs** coincide with those in **ev**. Products in **evs** do not coincide with those in **ev**, they are useful for modeling sequential aggregation.

Object behaviors. In order to model object behavior, events are labelled by local class propositions that are formulae describing the current values of attributes, actions enabled, actions occurred, etc.

For a given object sort $s \in S_O$, we assume a *local class logic* $P_{\Delta,s}$, or P for short if Δ and s are clear from context. Δ is a *local class signature.*

Example 8. An example of a local class signature and logic are the following. $\Delta = (At, Ac)$ where $At = \{At_s\}_{s \in S}$ is an S-indexed set family of attribute symbols, and Ac is a given set of action symbols. The syntax of P is given by

$$P \ ::= \ At = T_\Sigma \mid \odot\, Ac \mid \triangleright Ac \mid P \wedge P$$

where $At = T_\Sigma$ is the set $\{at = t \mid at \in At_s\, ,\, t \in T_{\Sigma,s}, s \in S\}$ of well-sorted attribute=value equations, $\odot\, Ac = \{\odot\, ac \mid ac \in Ac\}$ is the set of action occurrence predicates, and $\triangleright Ac = \{\triangleright ac \mid ac \in Ac\}$ is the set of action enabling predicates.

Each local class formula is a finite conjunction of one or more attribute = value, action occurrence and action enabling predicates. Of course, there are general constraints as to which conjunctions are permissible. For instance, no attribute may have more than one value.

Other class logics can be adopted, for instance Horn logic or (conditional) equational logic. This way, relationships to deductive databases or dynamic algebras, respectively, can be established.

Definition 6. Let $E = (Ev, \to^*, \#)$ be an event structure. A *labelling* for E is a total function $\pi : Ev \to P$ where P is a class logic.

Depending on P, a labelling has to satisfy general constraints, for instance that actions have to be enabled when they occur. We do not go into detail here.

Definition 7. A *behavior* is a triple $B = (E, \pi, P)$ where $E = (Ev, \to^*, \#)$ is an event structure, and $\pi : Ev \to P$ is a labelling for E. B is called *sequential* iff E is. B is called a *life cycle* iff E is one, i.e., E is conflict-free.

The labelling of a sequential behavior is called *definite* iff, for all $e_1, e_2 \in Ev$ such that $e_1 \neq e_2$, we have $\pi(e_1) \neq \pi(e_2)$ whenever there is an event $e_3 \in Ev$ such that $e_3 \to e_1$ and $e_3 \to e_2$. That means that successor events are distinguishable by their labels. Definite sequential behaviors can be uniquely characterized by their label sequences. This is why much of the earlier work on sequential object theory is based on label sequences rather than event structures (cf., e.g., [**?**, **?**]).

Behavior morphisms. Behaviors are related by behavior morphisms which are event structure morphisms preserving labels in a canonical way.

Definition 8. Let $B_i = (E_1, \pi_1, P_1)$ and $B_i = (E_2, \pi_2, P_2)$ be behaviors. A *behavior morphism* $f : B_1 \to B_2$ is pair $f = (g, h)$ where $g : E_1 \to E_2$ is an event structure morphism and $h : P_1 \to P_2$ is a class logic morphism, such that $h \circ \pi_1 = \pi_2 \circ g$. f is called *sequential (total, surjective, ...)* iff g is.

Class logic morphisms are structure preserving maps between class logics. We do not go into detail but explain the concept by example.

Example 9. Referring to example 8, let P_1 and P_2 be class logics over class signatures Δ_1 and Δ_2, respectively. Then a *class logic morphism* $h : P_1 \to P_2$ is given by a class signature morphism $\sigma : \Delta_2 \to \Delta_1$ that is a total function $\sigma_{ac} : Ac_2 \to Ac_1$ on action symbols, and an S-indexed family of total functions $\sigma_{at} : At_2 \to At_1$ on attribute symbols. σ is extended to class formulae in the usual way, yielding a syntactic map $\sigma : P_2 \to P_1$.

Now $h : P_1 \to P_2$ is defined by letting $h(p)$ be the strongest condition q such that $p \models_o \sigma(q)$ where $\models_o$ denotes entailment in the class logic. Intuitively, h denotes the "projection along σ" of class formulae.

A helpful intuition for behavior morphisms $f : B_1 \to B_2$ is that f extracts B_2 as a kind of "partial view" from B_1, with labelling reduced to the relevant class propositions.

Remark. With mild assumptions about the underlying category of class signatures and logics, limits and colimits in **ev** and **evs** carry over to the corresponding behavior categories **bhv** of behaviors and behavior morphisms, and **seq** of sequential behaviors and sequential morphisms. This is utilized in subsection 3.3 for modeling object composition.

3.2 Specification and semantics

Temporal logic and interpretation. Let Δ be a local class signature. For temporal specification, we use propositional linear-time temporal logic $\text{PTL}(\Delta)$ including past- and future-directed temporal operators [**?**]. For simplicity, we concentrate on the future-directed fragment $\text{PTL}^+(\Delta)$.

Definition 9. The *syntax* of $\mathrm{PTL}^+(\Delta)$, i.e., its set of well-formed formulae Φ, is given by

$$\Phi ::= \hbar(P) \mid \mathit{false} \mid (\Phi \Rightarrow \Phi) \mid (\Phi\, \mathcal{U}\, \Phi).$$

A proposition $\hbar(p)$ is intended to mean that class proposition p holds. These propositions 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, and $\mathcal{U}$ is the temporal *until* operator.

$\varphi\, \mathcal{U}\, \psi$ means that, from the next moment on, φ will always be true 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.

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 must happen eventually.

Furthermore, we apply the usual rules for omitting brackets.

Formal interpretation is described in terms of *possible worlds* cast in event structure terminology. If $B = (E, \pi, P)$ is a behavior where $E = (Ev, \rightarrow^*, \#)$ is an event structure, then Ev is the set of possible worlds, $\rightarrow$ is an accessibility relation corresponding to *next*, $\rightarrow^*$ is an accessibility relation corresponding to *eventually*, and $\pi(e) \models_o p$ corresponds to the satisfaction of class formulae at worlds.

Interpretation is given in arbitrary behaviors, i.e., we allow for nonstandard interpretations [**?**]. This generality is useful because it also captures the distributed temporal logic described in subsection 3.3.

In this subsection, however, we concentrate on standard interpretations which are linear life cycles in sequential behaviors.

Let $B = (E, \pi, P)$ be a behavior, and let $e \in Ev$. Satisfaction $B@e \models \varphi$ means that the temporal formula φ is valid at an event e in behavior B. $\rightarrow^+$ denotes the irreflexive transitive closure of $\rightarrow$.

Definition 10. The satisfaction relation $\models$ is inductively defined as follows.

$B@e \models \hbar(p)$	holds iff $\pi(e) \models_o p$ holds,
$B@e \models \mathit{false}$	does not hold,
$B@e \models (\varphi \Rightarrow \psi)$	holds iff $B@e \models \varphi$ implies $B@e \models \psi$,
$B@e \models (\varphi\, \mathcal{U}\, \psi)$	holds iff, in every life cycle $L \in \mathcal{L}(E)$ containing e,

there is a future event $e' \in L$, $e \to^+ e'$, where
$B@e' \models \psi$ holds, and $B@e'' \models \varphi$ holds for
every event $e'' \in L$ such that $e \to^+ e'' \to^+ e'$.

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

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

Specification in the small. An in-the-small behavior specification is a temporal logic specification based on a given local class logic. Let Δ be a local class signature.

Definition 11. A *behavior specification* is a pair *Bspec* $= (\Delta, \Phi)$ where Φ is a set of formulae in $\text{PTL}^+(\Delta)$.

Only sequential behavior can be specified with temporal logic. Therefore, semantics is defined in the subcategory Δ**-seq** $\subseteq$ **seq** where the class signature Δ is fixed and, for every morphism $f = (g, h)$, h is the identity.

Δ**-seq** has final elements[2]. A final element in Δ**-seq** is definite, it represents the most liberal definite behavior. Intuitively, it consists of all traces that can be formed over Δ and satisfy its general constraints.

A corresponding result holds for model categories of specifications.

Let *Bspec* $= (\Delta, \Phi)$ be a specification. Let *Bspec*-**seq** be the full subcategory of Δ**-seq** consisting of all behaviors over Δ satisfying Φ.

Bspec-**seq** has final elements. A final element is obtained as the subbehavior of a final element in Δ**-seq** determined by all life cycles satisfying Φ. Consequently, final elements in *Bspec*-**seq** are definite.

The finality results are contained in [**?**].

We may utilize these results for assigning standard semantics to temporal template specifications in an abstract way, i.e., up to isomorphism. However, we cannot specify nondefinite behavior this way. In order to increase specification power and manageability, we envisage to give direct axiomatic specification only to basic behaviors that are per se definite. Derived and complex behaviors may be specified using structuring mechanisms like those described in 3.3. In particular, nondefiniteness comes with hiding and with generalization. Using concurrent aggregation, also nonsequential behavior can be specified.

Aggregation is usually supplemented by *interaction specifications* describing how components interact. For sequential aggregation, temporal logic is sufficient. For concurrent aggregation, however, an extension is needed.

[2] For obvious reasons, we avoid the term *object* for the elements of a category.

3.3 Specification in the large

For an effective specification method, in-the-large concepts based on structuring mechanisms are indispensible. We express structuring by structured object sorts and corresponding object type morphisms. The exposition is sketchy, we refer to [**?**] for a more detailed account.

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 relations on object sorts. All may be multiple. Composition is expressed by binary order sort constructors for generalization as well as sequential and concurrent aggregation. The latter reflects distribution of components.

Interaction between components of a sequential aggregation can be specified in the framework developed so far, using inheritance. For specifying interaction between distributed objects locally, however, logic and semantics have to be extended.

Let $\Sigma = (S, \Omega)$ be a signature. Let $S = S_D + S_O$ where S_D are data sorts and S_O are object sorts. Let U be a Σ-algebra.

Inheritance describes how an object or class reuses features from another one. On a representation level, the inheriting object or class has the same attributes and actions and possibly additional ones. On an implementation level, the inheriting object or class reuses code, i.e., it implements inherited attributes and actions in the same way as in the original object or class.

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

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.

We distinguish between three kinds of inheritance: *is-a inheritance* $r \underset{\rightarrow}{\leq} s$, *as-a inheritance* $r \underset{\leftarrow}{\leq} s$, and *hiding* $r \sqsubseteq s$. For the first two, we may define *full* versions $r \underset{\rightarrow}{\lneq} s$ and $r \underset{\leftarrow}{\lneq} s$, respectively, that exclude overriding. All kinds of inheritance may be multiple.

Depending on the kind of inheritance considered, it is formalized in our approach by an object type morphism $\mu : r_U \rightrightarrows s_U$ or a reverse object type morphism $\mu : r_U \rightleftarrows s_U$.

Is-a inheritance $r \underset{\rightarrow}{\leq} s$ is formalized by an object type morphism $\mu : r_U \rightrightarrows s_U$ such that $r_U^{id} \subseteq s_U^{id}$. As an example, consider *Customer* $\underset{\rightarrow}{\leq}$ *Person* where a

Customer is-a Person and has a *richer* behavior than a *Person.* The behavior relationship is modeled by a behavior morphism $\mathit{Customer}^{bv} \to \mathit{Person}^{bv}$.

As-a inheritance $r \underset{\leftharpoondown}{\leq} s$ is formalized by an object type morphism $\mu : r_U \rightleftarrows s_U$ such that $r_U^{id} \subseteq s_U^{id}$. As an example, consider $\mathit{Square} \underset{\leftharpoondown}{\leq} \mathit{Rectangle}$ where a *Square is-a Rectangle* and has a *more restricted* behavior than a *Rectangle.* For instance, rectangles may be stretched along the x- and y-axes separately while squares cannot. The behavior relationship is modeled by a behavior morphism $\mathit{Rectangle}^{bv} \to \mathit{Square}^{bv}$.

Hiding $r \sqsubseteq s$ is formalized by an object type morphism $\mu : r_U \rightleftarrows s_U$ such that $r_U^{id} \cap s_U^{id} = \emptyset$. As an example, consider $\mathit{View} \sqsubseteq \mathit{Database}$ where a *View* is a separate object with a more restricted behavior than a *Database.* The behavior relationship is modeled by a behavior morphism $\mathit{Database}^{bv} \to \mathit{View}^{bv}$.

Composition describes how objects are put together to form complex objects.

Let $\Sigma = (S, \Omega)$ be a signature. Let $S = S_D + S_O$ where S_D are data sorts and S_O are object sorts. Let $r, s \in S_O$. Let U be a Σ-algebra.

On the syntactic level, composition is expressed by object sort constructors. We have three kinds of composition: generalization $r \oplus s$, sequential aggregation $r \otimes s$, and concurrent aggregation $r \parallel s$. These constructors are associative, so the same constructor may be iterated any finite number of times.

On the semantic level, generalization and concurrent aggregation are expressed by coproducts or products, respectively, in the category **ot(bhv)** of object types with direct morphisms, built over the behavior category **bhv**. Sequential aggregation is expressed by products in the subcategory **ot(seq)** of object types over the behavior subcategory **seq**.

Generalization. Composition by generalization $r \oplus s$ is intended to express how objects can be put together to reflect alternative choice among its constituents. Generalization does not add any concurrency that was not yet there in the constituents. In particular, if we generalize sequential objects, the generalized object is sequential.

Interpretation is given by $(r \oplus s)_U = r_U + s_U$, the coproduct in **ot(bhv)**. The coproduct in **ot(seq)** coincides with that in **ot(bhv)**. The coproduct morphisms describe how the constituent object types are embedded into the generalization.

As an example, consider $\mathit{Customer} = \mathit{Person} \oplus \mathit{Company}$ saying that each *Customer* is either a *Person* or a *Company.* The identity of a customer is its identity as a person or company, respectively, and the same holds for behaviors.

Sequential aggregation. Composition by sequential aggregation $r \otimes s$ is intended to express how sequential objects can be put together in a synchronized way. Sequential aggregation may only be applied to sequential objects. The result is again sequential. A life cycle of an aggregated object is a step-by-step synchronization of life cycles of its constituents, sharing events at each step.

Interpretation is given by $(r \otimes s)_U = r_U \bowtie s_U$, the product in the category **ot(seq)**. The product morphisms describe how the behaviors of constituents are extracted from the aggregation.

Sequential aggregation forms complex objects whose identities are pairs of component identities, and whose behaviors are fully synchronized compositions of those of their components. As an example, consider object types *Clock* and *Automaton*. Objects of type *Clock* $\otimes$ *Automaton* are clocked automata where every *Automaton* event is synchronized with a *tic* of the *Clock*, and vice versa.

Sequential interaction. Synchronous interaction between components of a sequential aggregation can easily be described by temporal axioms over the class logic of the aggregation. There are no problems since the result of a sequential aggregation is sequential.

For example, we may add the axiom $\odot\, \mathit{tic} \Rightarrow \odot\, \mathit{incr\text{-}counter}$ in a clocked automaton in order to express that every *tic* action must be synchronized with an *incr-counter* action. Technically, for every event in the clocked automaton, the above formula must be entailed by the class information.

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

Concurrent aggregation. Composition by concurrent aggregation $r \parallel s$ is intended to express how objects can be put together in a concurrent way to reflect distribution of its constituents. Concurrent aggregation allows for concurrency between constituent objects. In particular, if we put sequential objects together this way, we obtain a non-sequential object. A life cycle of an aggregated object is a union of life cycles of its constituents, sharing events at synchronization points.

Interpretation is given by $(r \parallel s)_U = r_U \times s_U$, the product in the category **ot(bhv)**. The product morphisms describe how the behaviors of constituents are extracted from the aggregation.

Concurrent aggregation forms complex objects whose identities are pairs of component identities, and whose behaviors are concurrent compositions of those of their components. Behaviorally, components remain independent but may synchronize on specific events. As an example, consider object types *Bank* and *Customer*. Objects of type *Bank* $\parallel$ *Customer* describe a Bank with a Customer and their possibilities to interact.

Concurrent interaction. Interaction between components of a concurrent aggregation cannot be described by temporal logic because temporal logic can only talk about sequential objects. The result of a concurrent aggregation is not sequential in general.

So we need an extension of temporal logic. However, a logic powerful enough to talk about all aspects of concurrency is too difficult to work with. The way out is to restrict the models to *locally sequential* behaviors. These are behaviors of the form $B = B_1 \times B_2 \times \cdots B_n$ where $\times$ denotes product in **bhv**, and B_i is a sequential behavior for each $i \in \{1, \ldots n\}$. B may be seen as the joint behavior of n sequential agents working concurrently.

Locally sequential behaviors are expressive enough to display the kind of concurrency that often occurs in practice, for instance in a network of sequential work stations. On the other hand, they are close enough to sequential behaviors to allow for talking about them in a smooth extension of temporal logic.

Such a logic is the *n-agent logic* as introduced and developed in [**?**, **?**, **?**, **?**]. An agent corresponds to a local sequential behavior. Temporal descriptions are given locally from the viewpoint of an agent. 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: u says "I received something from you" while i says "what a coincidence, I sent something to you".

We introduce our version of n-agent logic called $\mathrm{D{\scriptstyle TL}}(\Delta)$ where Δ is a given class signature. Here we describe the future-directed fragment $\mathrm{D{\scriptstyle TL}}^+(\Delta)$. Let $I = \{i, u, \ldots\}$ be a given finite set of *localities*. Localities are names of agents.

Definition 12. The *syntax* of $\mathrm{D{\scriptstyle TL}}^+(\Delta)$, i.e., its set of well-formed formulae Φ, is given by

$$\Phi ::= \hbar(P) \mid \mathit{false} \mid (\Phi \Rightarrow \Phi) \mid (\Phi \,\mathcal{U}_I\, \Phi).$$

The only difference to $\mathrm{P{\scriptstyle TL}}^+(\Delta)$ is that we have temporal operators $\mathcal{U}_i$ indexed by localities. The intended meaning of $\varphi \,\mathcal{U}_i\, \psi$ is that $\varphi \,\mathcal{U}\, \psi$ holds locally at agent i. As in the $\mathrm{P{\scriptstyle TL}}^+(\Delta)$ case, we may introduce abbreviations X_i, $\mathsf{X}_i^?$, F_i, G_i, $\mathcal{U}_i^\circ$ and $\mathcal{P}_i^+$. Additionally, we introduce the abbreviation $@u$ for $\mathsf{X}_u^?$ *true*, saying that there is interaction with u at this point in time.

Specification in $\mathrm{D{\scriptstyle TL}}^+(\Delta)$ is bound to localities. The notation $i : \varphi$ says that formula φ is bound to locality i.

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

Example 10. In order to give the example a real-life touch, we read "I" for i and "you" for u.

$i : \mathsf{F}_u\, \varphi$	*I hear that φ will sometime be valid for you*
$i : \mathsf{G}_i(@u \;\Rightarrow\; \mathsf{X}_i\, \varphi)$	*whenever I talk to you, φ holds 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*

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

Interpretation is given in local behaviors $B_i, B_u, \ldots \subseteq B$ in an arbitrary *global* behavior $B = (E, \pi, P)$. Of course, $i, u, \ldots \in I$ are the given localities.

Satisfaction is also defined locally. Let $i \in I$ be a given locality, and let $e \in Ev_i$. $B@e \models_i \varphi$ means that φ holds locally for i in behavior B at event e.

Here the generality of definition 10 pays off: we may replace $\models$ by $\models_i$ and $\mathcal{U}$ by $\mathcal{U}_i$ and add one further rule in order to obtain the definition of local validity. The further rule is

$$B@e \models_i (\varphi\, \mathcal{U}_u\, \psi) \ \text{ iff } \ e \in Ev_u \text{ and } B@e \models_u (\varphi\, \mathcal{U}_u\, \psi) \text{ holds}$$

where $u \in I$.

For the n-agent model, we have $B = B_1 \times \cdots \times B_n$ where B_k is a sequential behavior for each $k = 1, 2, \ldots$. In this case, life cycles are n-tuples of linear sequential life cycles that may share events. These shared events are points of interaction. Please note that the local configuration $\downarrow e$ of an event $e \in B_i, i \in I$, contains – with each point of interaction $e_0 \to^* e$ – all events causal for e_0: in agent i and in the agents i is interacting with at e_0! In this sense, agents obtain full historic knowledge of others at interaction points.

4 Concluding remarks

In this paper, we briefly describe related pieces of practical and theoretical work that are being integrated. Troll and the TBench will be extended towards capturing distribution, based on distributed temporal logic. The underlying theory has to be elaborated and refined, a more detailed account will appear in [**?**]. Special aspects of Dtl and its use for describing semantics of distributed systems can be found in [**?**].

Further research will focus on interaction and modularization concepts. Dtl describes synchronous and symmetric interaction, albeit from a local point of view. On the Troll language level, a richer spectrum of interaction concepts is envisaged, including modes of asynchronous directed interaction. Fortunately, 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 [**?**, **?**].

Another issue of practical importance is real time. Real-time constraints will be investigated and incorporated into Troll that 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.

Acknowledgments

The authors are grateful for help and inspiration from many sides. Amílcar and Cristina Sernadas have been faithful partners in developing the theoretical foundations and discussing all matters of interest. The other partners in the ISCORE and COMPASS projects have given so manyfold inputs that it is impossible to acknowledge in detail. Gunter Saake, Thorsten Hartmann and Jan Kusch were driving forces behind developing TROLL and the TBench. Grit Denker and Juliana Küster Filipe have contributed essentially to internal discussions on the subjects of the paper. Mojgan Kowsari does a great job in putting TROLL to practice.

Last but not least, the first author is very indebted to Prof. Tang for discussions on temporal logic, software development, and Chinese wisdom and philosophy.

References

[Aba89] M. Abadi. The Power of Temporal Proofs. Theoretical Computer Science 65 (1989), 35-83

[AB95] M. Aiguier and G. Bernot. Algebraic Specification of Object Type Specifications. In R.J. Wieringa and R.B. Feenstra (eds.), Information Systems – Correctness and Reusability, Selected Papers from the IS-CORE Workshop'94. World Scientific Publishers, 1995, 16-32

[AGM] S. Abramski, D. Gabbay and T. Maibaum. Handbook of Logic in Computer Science, Vols. 1-5. Oxford Science Publications, Oxford 1992ff

[AG91] A.J. Alencar and J.A. Goguen. OOZE: An object-oriented *Z* environment. In P. America (ed.), ECOOP'91, Springer, Berlin 1991, pages 180–199

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

[CSS94] J.F. Costa, A. Sernadas, and C. Sernadas. Object Inheritance Beyond Subtyping. Acta Informatica 31 (1994), 5-26

[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 und G. Saake, Hrsg., Proc. Workshop on Formal Methods for Information System Dynamics (CAiSE'94-Workshop), pages 25–39. University of Twente, Technical Report, 1994

[DE95] G. Denker and H.-D. Ehrich. Action Reification in Object-Oriented Specification. In R.J. Wieringa and R.B. Feenstra (eds.), Information Systems – Correctness and Reusability, Selected Papers from the IS-CORE Workshop'94. World Scientific Publishers, 1995, 103-118

[Den95] G. Denker. Verfeinerung in objektorientierten Spezifikationen: Von Aktionen zu Transaktionen. PhD Thesis, TU Braunschweig 1995

[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), pages 453–467. Springer, Berlin, LNCS 668, 1993.

[EGL89] H.-D. Ehrich, M. Gogolla, and U. Lipeck. Algebraische Spezifikation Abstrakter Datentypen. Teubner-Verlag, Stuttgart 1989

[EGS90] H.-D. Ehrich, J. A. Goguen and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J.W. deBakker, W.P. deRoever und G. Rozenberg, Hrsg., Proc. REX/FOOL Workshop, pages 203–228, Noordwijkerhood (NL), 1990. LNCS 489, Springer, Berlin

[Ehr96] H.-D. Ehrich. Object Specification. IFIP 14.3 Volume on Foundations of System Specification, Chapter 11. H.-J. Kreowski, ed. *to appear* in Springer LNCS

[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, pages 201–218. Springer Verlag, Wien, CISM Courses and Lectures no. 347, 1994

[EM85] H. Ehrig and B. Mahr. Fundamentals of Algebraic Specification 1. Springer-Verlag, Berlin 1985

[ES90] H.-D. Ehrich and A. Sernadas. Algebraic implementation of objects over objects. In deBakker, J. W., deRoever, W.-P., and Rozenberg, G., editors, Proc. REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness", LNCS 430, pages 239–266. Berlin, 1990. Springer-Verlag

[ES91] H.-D. Ehrich and A. Sernadas. Fundamental Object Concepts and Constructions. Information Systems – Correctness and Reusability, Proc. ISCORE Workshop'91 (G. Saake and A. Sernadas, eds.), Informatik-Berichte 91-03, Techn. Univ. Braunschweig 1991, 1-24

[ES95] Ehrich, H.-D. and Sernadas, A. Local Specification of Distributed Families of Sequential Objects. In Astesiano, E., Reggio, G., and Tarlecki, A., editors, Recent Trends in Data Types Specification, Proc. 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, S.Margherita, Italy, May/June 1994, Selected papers, pages 218-235. Springer, Berlin, LNCS 906, 1995.

[ESDI93] OBLOG CASE V1.0 – The User's Guide. Espírito Santo Data Informática (ESDI), Lisbon 1993

[ESS90] H.-D. Ehrich, A. Sernadas, and C. Sernadas. From Data Types to Object Types. J. Inf. Process. Cybern. EIK 26 (1990), 33-48

[ESS92] H.-D. Ehrich, G. Saake, and A. Sernadas. Concepts of object-orientation. In Proc. of the 2nd Workshop of "Informationssysteme und Künstliche Intelligenz: Modellierung", Informatik–Fachberichte 303, pages 1–19, Berlin, 1992. Springer–Verlag

[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

[GB92] J.A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. Journal of the ACM 39 (1992), 95-146

[GM87] J.A. Goguen and J. Meseguer. Unifying functional, Object–oriented and Relational Programming with logical semantics. Research Direction in Object-Oriented Programming, B.Shriver,P.Wegner (eds.), MIT Press 1987, pages 417–477

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. The Temporal Analysis of Fairness. Proc. 7th ACM Symp. on Principles of Programming Languages (1980), 163-173

[GS95] J.A. Goguen and A. Socorro. Module Composition and System Design for the Object Paradigm. Journal of Object oriented Programming Vol. 7, 14, 1995

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

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. Science of Computer Programming, Vol.8, 231–274, 1987.

[HJ95] P. Hartel and R. Jungclaus. Modeling Business Processes over Objects. Int. Journal of Intelligent and Cooperative Information Systems, Vol.4 (1995), 165–188.

[HSJ$^+$94] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel und J. Kusch. Revised Version of the Modelling Language TROLL (Version 2.0). Informatik-Bericht 94–03, Technische Universität Braunschweig, 1994

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

[Jun93] R. Jungclaus. *Modeling of Dynamic Object Systems—A Logic-Based Approach*. Advanced Studies in Computer Science. Vieweg Verlag, Braunschweig/Wiesbaden, 1993.

[JSHS95] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. ACM Transactions on Information Systems, 1995. *To appear*

[JWH$^+$94] R. Jungclaus, R.J. Wieringa, P. Hartel, G. Saake, and T. Hartmann. Combining TROLL with the Object Modeling Technique. In B. Wolfinger, editor, Innovationen bei Rechen- und Kommunikationssystemen. GI-Fachgespräch FG 1: Integration von semi-formalen und formalen Methoden für die Spezifikation von Software, pages 35–42. Springer, Informatik aktuell, 1994.

[KHHS95] J. Kusch, P. Hartel, T. Hartmann, and G. Saake. Gaining a Uniform View of Different Integration Aspects in a Prototyping Environment. In Proc. 6th Int. Conf. on Database and Expert Systems Applications (DEXA'95), pages 38–47. Springer, LNCS 978, 1995.

[Kne89] R. Kneuper. Symbolic Execution as a Tool for Validation of Specifications. Technical Report UMCS-89-7-1, Department of Computer Science, University of Manchester, 1989.

[LEW96] J. Loeckx, H.-D. Ehrich, and M. Wolf. Specification of Abstract Data Types. Wiley/Teubner, 1996. *To appear*

[LMRT91] K. Lodaya, M. Mukund, R. Ramanujam, and P.S. Thiagarajan. Models and Logics for True Concurrency. in P.S. Thiagarajan (ed.): Some Models and Logics for Concurrency. Advanced School on the Algebraic, Logical and Categorical Foundations of Concurrency. Gargnano del Garda, 1991

[LP90] Huimin Lin and Man-chi Pong. Modelling Multiple Inheritance with Colimits. Formal Aspects of Computing 2 (1990), 301-311

[LRT92] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. Temporal Logics for Communicating Sequential Agents: I. International Journal of Foundations of Computer Science, Vol. 3 (1992), 117-159.

[LT87] K. Lodaya and P.S. Thiagarajan. A Modal Logic for a Subclass of Event Structures. Proc. 14th Int. Colloq. on Automata, Languages and Programming, Th. Ottmann (ed.), LNCS 267, Springer-Verlag, Berlin 1987, 290-303.

[Mes92] J. Meseguer. Conditional Rewriting as a Unified Model of Concurrency. Theoretical Computer Science, 96(1): pages 73–156, 1992

[Mes93] J. Meseguer. A Logical Theory of Concurrent Objects and Its Realization in the Maude Language. In G. Agha, P. Wegener und A. Yonezawa, Hrsg., Research Directions in Object-Oriented Programming, pages 314–390. MIT Press, 1993

[MHM95] R. Mitchell, J. Howse, and I. Maung. As-a: a relationship to support code reuse. JOOP 8,4 (1995), 25-55

[MP92] Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, New York 1992

[MP95] Z. Manna and A. Pnueli. Temporal Verification of Reactive Systems – Safety. Springer-Verlag, New York 1995

[MT92] M. Mukund and P.S. Thiagarajan. A Logical Characterization of Well-Branching Event Structures. Theoretical Computer Science 96 (1992), 35-72

[Pnu77] A. Pnueli. The Temporal Logic of Programs. Proc. 19th Annual Symposium on Foundations of Computer Science. IEEE. New York 1977, 46-57

[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. Technical Report Report PRG-TR-28-92, Programming Research Group, Oxford University Computing Laboratory, 1992

[SFSE88] A. Sernadas, J. Fiadeiro, C. Sernadas und H.-D. Ehrich. Abstract Object Types: A Temporal Perspective. In B. Banieqbal, H. Barringer und A. Pnueli, Hrsg., Proc. Colloq. on Temporal Logic in Specification, pages 324–350. LNCS 398, Springer, Berlin, 1988

[SFSE89] A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich. The Basic Building Blocks of Information Systems. In E. Falkenberg and P. Lindgreen, editors, Information System Concepts: An In-Depth Analysis, pages 225–246. North-Holland, Amsterdam, 1989.

[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, pages 219–245. Springer Verlag, Wien, CISM Courses and Lectures no. 347, 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, Berlin 1991, pages 99–121. North–Holland, 1992

[SJH93] G. Saake, R. Jungclaus, and T. Hartmann. Application Modelling in Heterogeneous Environments using an Object Specification Language. Int. Journal of Intelligent and Cooperative Information Systems, Vol.2 (1993), 425–449.

[SR94] A. Sernadas and J. Ramos. The GNOME Language: Syntax, Semantics and Calculus. Tech. Report, Instituto Superior Técnico, Lisboa 1994

[SSC95] A. Sernadas, C. Sernadas, and J.F. Costa. Object Specification Logic. Journal of Logic and Computation 5 (1995), 603-630

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

[Tan94] C.S. Tang. A Temporal Logic Language Oriented Toward Software Engineering – An Introduction To The XYZ Sytem. Chinese Journal of Advanced Software Research Vol. 1 (1994), 1-29

[Thi94] P. S. Thiagarajan. A Trace Based Extension of Linear Time Temporal Logic. Proc. 9th Annual IEEE Symposium on Logic in Computer Science, CNAM, Paris, July 4-7 1994, IEEE Computer Science Press, 438-447

[Win87] G. Winskel. Event Structures. Petri nets: Applications and relationships to other models of concurrency, W. Brauer, W. Reisig and G. Rozenberg (eds.), LNCS 255, Springer-Verlag, Berlin 1987, 325-392

[WJH$^+$93] R. Wieringa, R. Jungclaus, P. Hartel, T. Hartmann, and G. Saake. omTroll – Object Modeling in TROLL. In U.W. Lipeck and G. Koschorreck, editors, Proc. Intern. Workshop on Information Systems – Correctness and

Reusability IS-CORE '93, Technical Report, University of Hannover No. 01/93, pages 267–283, 1993.

[WN94] G. Winskel and M. Nielsen. Models for Concurrency. [?] Vol. 4 (1994), 1-148