The TROLL Approach to Conceptual Modelling: Syntax, Semantics, and Tools[⋆]

Antonio Grau, Juliana Küster Filipe, Mojgan Kowsari,
Silke Eckstein, Ralf Pinger, and Hans-Dieter Ehrich

Technische Universität Braunschweig, Informatik, Abt. Datenbanken,
Postfach 3329, D–38023 Braunschweig, Germany
{grau,filipe,kowsari,eckstein,pinger,ehrich}@idb.cs.tu-bs.de
http://www.cs.tu-bs.de/idb/welcome_e.html

Abstract. In this paper, we present the use of TROLL for the conceptual modelling of distributed information systems. TROLL offers both textual and graphical notations. TROLL has been used in practice to model an industrial information system. We use an extract of this case study to describe briefly the syntax and underlying semantics of the language. We also show a set of software tools that are being developed to support the modelling with TROLL. These tools include editors, checkers as well as an animator for validating TROLL specifications. We report on the experiences we gained by applying the language to the industrial project. Finally, a short description on further work is given.

1 Introduction

One main issue in the conceptual modelling of information systems is that a model has to describe both structural and behavioural aspects of a system. Traditionally, the modelling of both aspects has been treated separately. In the last years the application of the object–oriented approach to this area has contributed enormously to integrate these aspects in only one formalism. The object–oriented approach includes within an object both structural and behavioural properties. Numerous object-oriented techniques for conceptual modelling have emerged. Informal object-oriented methodologies like UML [10] are very popular. Regarding formal methods either new languages like LCM [8], ALBERT [3] and OASIS [19] or extensions to well-known languages as in VDM++ [4], Z++ [16], and LOTOS [1] have been developed [20].TROLL, the language dealt with in this paper, belongs to the group of new formal languages developed for supporting the object-oriented paradigm. TROLL is based on OBLOG [22] and takes ideas and concepts from abstract data types, behaviour modelling, specification of reactive systems and concurrency theory. In this paper we want to present TROLL in its third version [2]. Previous versions have been published in [13,12] among

⋆ Work reported here was partially supported by the EU under ESPRIT-IV WG 22704 ASPIRE, the DFG under Eh 75/11-1, the Spanish Ministry of Education and Culture and the Technical Federal Board, lab 3.41, project CATC.

others. In contrast to older versions, Troll version 3.0 is devoted to modelling distribution issues and has also been designed with the aim of executability.
In this paper we do not want to compare Troll with other approaches, but give a description of its concepts and features. Our approach combines informal and formal techniques taking advantages from both of them. For giving a first overview of the system and as a means of communication between users and developers, a graphical notation called OmTroll is used. From OmTroll a frame of a textual Troll specification can be derived. The textual specification has to be refined to a complete system specification. The formal nature of Troll helps to achieve concise and unambiguous system specifications as well as to develop better CASE tools. We are currently developing a CASE environment for modelling and validating Troll specifications.
For giving Troll a formal semantics, a special kind of temporal logic was needed to describe properties of distributed objects. This logic is called *Distributed Temporal Logic* (Dtl) [6] and is based on n-agent logics [17]. Each object has a local logic allowing it to make assertions about system properties through communication with other objects. Objects are represented by a set of Dtl formulae interpreted over labelled prime event structures [18]. Interaction between concurrent objects is done by synchronous message passing. The system model is obtained from its object models, whereby object interaction is represented by shared events.
We are using Troll to develop an information system in an industrial environment. Our experiences with the use of Troll in the project are very positive. The objective of the project is to develop a large information system supporting the activities of different user groups. The complexity of the organisation and the system that is supposed to support this organisation is rather high. Besides, the system has to integrate already existing applications and respecified ones. In order to achieve a deep understanding of the organisational structures we have to build an abstract model of the organisation which has to cover all structural aspects and organisational activities.
The paper is organised as follows. In Sect. 2 we introduce the case study. In Sects. 3 and 4 we make use of a simplified extract of the case study for illustrating the syntax and semantics of Troll, respectively. Section 5 gives a short description of the CASE environment. We report on the experiences we gained by applying Troll to the case study in Sect. 6. Finally, Sect. 7 summarises the paper and outlines further work.

2 Case Study

Our case study describes the activity of Group 3.4 settled in the PTB, which stands for Physikalisch-Technische Bundesanstalt[1]. The PTB is a government institute that cooperates in national and international committees, establishing rules for the fields of safety technology and standardisation among others. The group 3.4 is concerned with testing and certifying electrical equipment, e.g.,

[1] Federal institute of weights and measures.

motors, switches, heaters, which are to be used in explosion-hazardous areas. The focus is, for example, to avoid causing an explosion by turning on the lights with a switch in a chemical factory. Therefore, the switch has to be tested before it may be used in such areas. The testing procedure takes between 3-6 months and is worked out by approximately 100 certifying officers individually and manually according to the European Standards. On average, 1000 certificates per year are issued. It is important that all informations in connection with a certificate are available and reusable at any time. In 1994, the PTB started a cooperation with the database group of the Technical University Braunschweig to computerise the testing procedure. The system is called CATC (Computer Aided Testing and Certifying) and includes different subsystems based on the certifying process [11]. The certifying process for the electrical equipments is composed of the following parts:

– *administration management* includes the preliminary screening of the documents for the company producing or providing the electrical equipment. Such documents contain, e.g., name and address(es) of the company, name of the company responsable as well as the technical description of the equipment. This information is essential to setup the application form and has to be permanently available. Later, the staff prepares the settlement of the account and certifies documents depending on the test results.
– *design approval* includes the assessment of design papers for the equipment based on descriptions and its accordance with the European Standards. Here the staff creates a check list including the necessary experiments that have to be done by the operators.
– *experimental tests* are performed by the operators in the test lab and store all relevant data. There are different tests depending on the kind of the electrical equipment (for example temperature tests for heaters, pressure test for motors). The tests will be done on samples sent by the company. After the tests are done, the staff reads the results and decides whether more tests are necessary. Otherwise, the certification may be issued.

As mentioned above the testing and certifying process is a cooperation between staff and operators. The communication is an important factor to run the process correctly and more quickly. Both of them have to access the test results. The operators produce and store the test results while the staff reads and interprets them. The test results belonging to a certain experiment can be preselected and accessed by the staff. The operators start the experiment under a certain pressure and during a valid time. Due to European Standards, every application should be tested by more than one experiment to avoid any mistake. While the experiments for one application are running, the staff may set up the next experiment.
In the next section we will illustrate TROLL and OMTROLL using the communication depicted above. Because it would go beyond the scope of this paper, we exclude details from our system description and just consider a simplified extract of it.

3 Troll

Troll is a formal object–oriented language for the specification of distributed information systems at a high level of abstraction [2]. The textual language Troll has a graphical variant called OmTroll which borrows concepts from OMT [21]. In the following we first give a brief survey of the underlying ideas and then explain the main features of Troll and OmTroll by means of a simplified part of the case study from the previous section.
In our approach, an object–oriented system is considered to be a concurrent community of interacting objects. Objects are units of structure and behaviour. Similar objects are described in terms of object classes. They may be composed to complex objects through aggregation and arranged in inheritance hierarchies through specialisation. Objects defined separately may communicate through global interactions. Figure 1 shows the so called community diagram of our ex-

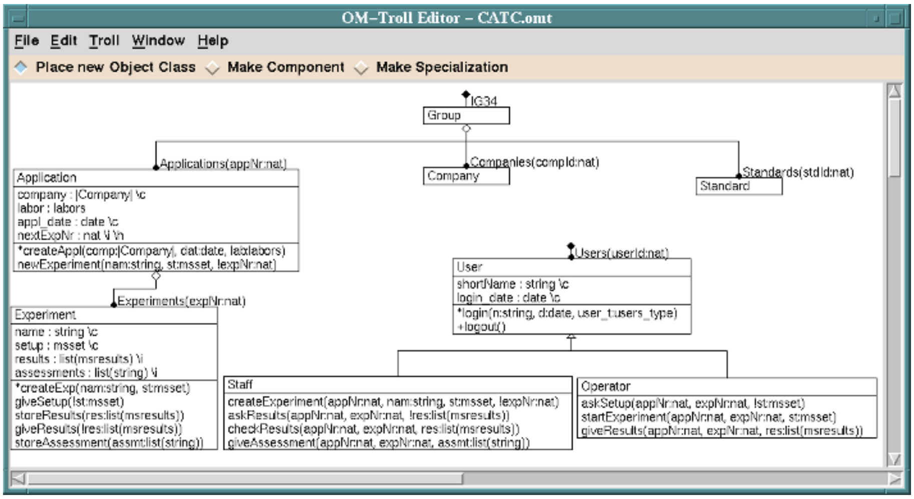

OM–Troll Editor – CATC.omt
File Edit Troll Window Help
Place new Object Class
Make Component
Make Specialization
IG34
Group
Applications(appNr:nat)
Companies(compId:nat)
Company
Standards(stdId:nat)
Standard
Application
company : |Company| \c
labor : labors
appl_date : date \c
nextExpNr : nat \i \h
*createAppl(comp:|Company|, dat:date, lab:labors)
newExperiment(nam:string, st:msset, !expNr:nat)
Experiments(expNr:nat)
Experiment
name : string \c
setup : msset \c
results : list(msresults) \i
assessments : list(string) \i
*createExp(nam:string, st:msset)
giveSetup(!st:msset)
storeResults(res:list(msresults))
giveResults(!res:list(msresults))
storeAssessment(assmt:list(string))
Users(userId:nat)
User
shortName : string \c
login_date : date \c
*login(n:string, d:date, user_t:users_type)
+logout()
Staff
createExperiment(appNr:nat, nam:string, st:msset, !expNr:nat)
askResults(appNr:nat, expNr:nat, !res:list(msresults))
checkResults(appNr:nat, expNr:nat, res:list(msresults))
giveAssessment(appNr:nat, expNr:nat, assmt:list(string))
Operator
askSetup(appNr:nat, expNr:nat, !st:msset)
startExperiment(appNr:nat, expNr:nat, st:msset)
giveResults(appNr:nat, expNr:nat, res:list(msresults))

Fig. 1. OmTroll Community Diagram

ample. Some of the object classes in there are described not only by their name but also by their signature (attributes and action declarations). We will explain this later in conjunction with the textual part of the language. The community diagram contains the aggregated object class `Group`, which represents group 3.4 of the PTB. It has object classes `Application, Company` and `Standard` as components, whereby `Application` itself has `Experiment` as such. Aggregation is represented by a diamond and the dot at the top of some of the object classes denotes multiple components. For example, each application may own a number of accompanying experiments, locally identified by their experiment number (`expNr`).
The object class `Group` is called a *node* of the system which is concurrent to the

other node shown in Fig. 1, namely User. Object class User is specialised in Staff and Operator. Specialisation is graphically presented by a triangle.
Besides the community diagram OMTROLL provides three more kinds of diagrams: data type diagrams for the declarations of user defined complex data types, behaviour diagrams to specify the allowed behaviour for each object class and communication diagrams to show interaction between object classes. An example for the last one is shown in Fig. 4.
From the OMTROLL diagrams a textual specification fragment can be derived. It contains all object classes with their signature, their component, specialisation and communication relationships as well as all user defined data types. The behaviour parts of the object classes, i.e., the operation definitions and local constraints, have to be extended. For example the object class Application as

```
object class Application
  components
   Experiments(expNr:nat):Experiment;
  attributes
   company : |Company| constant;
   labor : labors;
   appl_date : date constant;
   nextExpNr : nat initialized 1, hidden;
  actions
   *createAppl(comp:|Company|, dat: date, lab:labors);
   newExperiment(nam: string, st:msset, !expNr: nat);
  behavior
   createAppl(comp, dat, lab)
      do
        company:=comp,
        appl_date:=dat,
        labor:=lab
      od;
   newExperiment(nam, st, expNr)
      do
        Experiments(expNr).createExp(nam, st),
        expNr:=nextExpNr,
        nextExpNr:=nextExpNr+1
      od;
  constraints
   nextExpNr<10;
end;
```

Fig. 2. Object Class Application

shown in Fig. 2 has several Experiments as components, each one identified by a unique experiment number. It contains four attributes: company is an object valued attribute, i.e., a link to object class Company. In contrast to components, objects referenced by attributes are not embedded in the classes where the attributes are declared. The second attribute labor has a user defined data type labors, which we do not specify here. appl_date is a constant attribute, that means the date is once set and can never be changed. The attribute nextExpNr is initialised with one and it is hidden, i.e., it is only visible inside of object class Application. It stores the next number used to identify the experiments

belonging to this application. Furthermore, there are two actions: `createAppl` is a birth action that creates an object of class `Application`. It has all attributes of the class as parameters excluding `nextExpNr`, which is automatically initialised. `newExperiment` calls the birth action of the component class and increases `nextExpNr`. Finally, a constraint assures that there can not be more than nine experiments for each application.
After we explained the main language constructs to specify object classes we will now take a look at the specification of a whole object system (see Fig. 3). The keyword `object system` followed by the name of the system is the opening bracket. The specification itself consists of four parts: data type and object class specifications, object declarations and a behaviour part where the global interactions are described. In our case the name of the system is CATC. We

```
object system CATC
  data type ...
  object class Application ... end;
  object class ...
  objects IG34:Group;
  objects Users(userId:nat):User;
  behavior
  Staff(Users(userId)).createExperiment(appNr, nam, st, expNr)
     do
        IG34.Applications(appNr).newExperiment(nam, st, expNr)
     od;
end.
```

Fig. 3. Object System CATC

do not consider any user defined data types, but we already took a close look to object class `Application`. Besides this all other object classes shown in the community diagram (Fig. 1) have to be specified here. We declare two kinds of objects: `IG34` is the name of an instance of object class `Group`. After the occurrence of the accompanying birth actions this object will also contain objects of class `Application` as well as of class `Experiment` as components. The declaration of several objects of class `User` is done by parameterisation, i.e., each instance is identified by an `userId` of type `nat`. As `User` is the base class of an specialisation hierarchy, the objects will either have a special aspect `Staff` or `Operator`. In the behaviour part we give an example for a calling from an object of class `Staff` (identified by the `userId` of the base class) to a certain object of class `Application`, which can be reached via the aggregating object `IG34` through "dot notation". Whenever the action `createExperiment` occurs in `Staff`, the action `newExperiment` in `Application` is called, which in turn calls the birth action of class `Experiment` (see Fig. 2). The other object interactions of our example are shown in Fig. 4, where one can see the following: The communication between `Application` and `Experiment` has already been described just as the communication between `Staff` and `Application`. The four remaining relationships between `Staff` and `Experiment` and between `Operator` and `Experiment` concern the testing and certifying process. First the `Operator` has

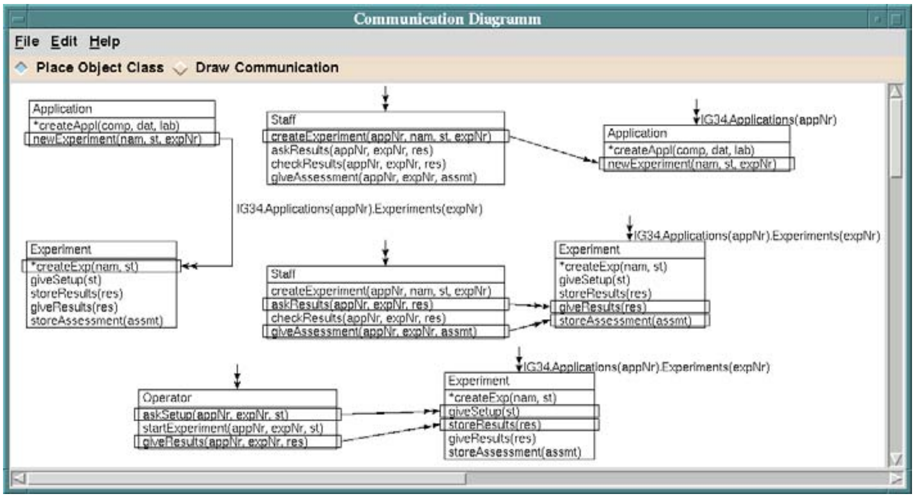

Communication Diagramm
File Edit Help
Place Object Class
Draw Communication
Application
*createAppl(comp, dat, lab)
newExperiment(nam, st, expNr)
Staff
createExperiment(appNr, nam, st, expNr)
askResults(appNr, expNr, res)
checkResults(appNr, expNr, res)
giveAssessment(appNr, expNr, assmt)
IG34.Applications(appNr)
Application
*createAppl(comp, dat, lab)
newExperiment(nam, st, expNr)
IG34.Applications(appNr).Experiments(expNr)
Experiment
*createExp(nam, st)
giveSetup(st)
storeResults(res)
giveResults(res)
storeAssessment(assmt)
Staff
createExperiment(appNr, nam, st, expNr)
askResults(appNr, expNr, res)
checkResults(appNr, expNr, res)
giveAssessment(appNr, expNr, assmt)
IG34.Applications(appNr).Experiments(expNr)
Experiment
*createExp(nam, st)
giveSetup(st)
storeResults(res)
giveResults(res)
storeAssessment(assmt)
IG34.Applications(appNr).Experiments(expNr)
Operator
askSetup(appNr, expNr, st)
startExperiment(appNr, expNr, st)
giveResults(appNr, expNr, res)
Experiment
*createExp(nam, st)
giveSetup(st)
storeResults(res)
giveResults(res)
storeAssessment(assmt)

Fig. 4. OMTROLL Communication Diagram

to know under which pressure the test has to be done, he `asks` for the `setup` values and gets the answer from action `giveSetup` in `Experiment`. After carrying out the tests he `stores` the `results` in `Experiment`. After this the `Staff` can `ask` for the `results`, `check` them and `store` an `assessment`.

4 TROLL Semantics

In this section we explain by means of the previously introduced example some aspects of the semantics underlying the TROLL language. Due to space limitations, we focus on inter-object communication. For more details on other semantical features of TROLL, like object relationships, see for instance [7].
From a system specification in TROLL we can derive both information on its structural and dynamic parts. The system structure is given by the system object instances together with their attributes and actions as described in the corresponding object classes. The structural part of the system is captured by a system signature. The system dynamics is given by the global interactions among the objects, constraints, initialisation declarations, action effects on the attributes, action preconditions, etcetera. The dynamic aspect of a system can be expressed in logical formulae. We use a distributed temporal logic called DTL [6]. Formulae written in DTL are interpreted in labelled prime event structures. A system specification is given by a *system signature* together with a set of DTL formulae, the *axioms* of the system.
In what follows we consider $\Sigma = (S, \Omega)$ a signature where S is a set of data and object sorts, i.e., $S = S_D \cup S_O$, and Ω is an $S^* \times S$-indexed family of sets of operation symbols.
A *system signature* is given by $\Sigma_I = (Id, At, Ac)$ where Id is a set of object identifiers (the system object instances), At is an $Id \times S$-indexed family of sets $At = \{At_{i,s}\}_{i \in Id, s \in S}$ of attribute symbols, and Ac is an $Id \times S^*$-indexed family

of sets $Ac = \{Ac_{i,x}\}_{i \in Id, x \in S^*}$ of action symbols. For an arbitrary $a \in At_{i,s}$, a is an attribute of object i and type s. For an arbitrary $b \in Ac_{i,s_1 \ldots s_n}$, b is an action of object i with n parameters of sorts $s_1 \ldots s_n$.
We illustrate these concepts with our example. In what follows, let $\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots$ be user numbers, $\mathtt{100}, \mathtt{200}, \ldots$ be application numbers and $\mathtt{j}$ an arbitrary user number. The CATC system signature is given by $\Sigma_{CATC} = (Id, At, Ac)$ where for instance

$$\begin{aligned}
&Id = \{\mathtt{IG34}, \mathtt{Users(0)}, \mathtt{Users(1)}, \mathtt{Users(2)}, \ldots\}\\
&At_{\mathtt{IG34},nat} = \{\mathtt{IG34.Applications(100).nextExpNr}, \ldots\}\\
&At_{\mathtt{IG34},\mathtt{msset}} = \{\mathtt{IG34.Applications(100).Experiments(7).setup}, \ldots\}\\
&Ac_{\mathtt{IG34},string\ \mathtt{msset}\ nat} = \{\mathtt{IG34.Applications(100).newExperiment}, \ldots\}\\
&Ac_{\mathtt{Users(j)},nat\ nat\ \mathtt{msset}} = \{\mathtt{Users(j).giveResults}\}
\end{aligned}$$

Before showing some CATC system axioms, we ought to give a brief understanding on DTL. DTL is an extension of linear temporal logic based on n-agent logics [17]. Each object $i \in Id$ has a local logic DTL$_i$. A local logic DTL$_i$ allows the object i to make assertions about system properties from its local viewpoint. Such system properties can be made based on the communication with other objects. DTL may be defined as follows:

$$\begin{aligned}
&\text{DTL} ::= \{\text{DTL}_i\}_{i \in Id}\\
&\text{DTL}_i ::= i.H_i \mid i.C_i\\
&H_i ::= \text{ATOM} \mid (H_i \Rightarrow H_i) \mid (H_i\ \mathcal{U}\ H_i) \mid (H_i\ \mathcal{S}\ H_i)\\
&\text{ATOM} ::= false \mid T_\Sigma\ \theta\ T_\Sigma \mid AT_\Sigma\ \theta\ T_\Sigma \mid \rhd AC_\Sigma \mid \odot AC_\Sigma\\
&C_i ::= \ \mathcal{C}_i \ \Rightarrow j.\mathcal{C}_j \qquad\qquad \text{for some } j \in Id, i \neq j
\end{aligned}$$

The local logic of object i is split into a local *home* logic H_i and a *communication* logic C_i. H_i is a first order linear temporal logic. T_Σ, AT_Σ and AC_Σ represent data, attribute and action terms respectively. An atomic formula of H_i can be the boolean constant $false$; the predicate θ applied to two data terms or to an attribute and data term, where θ is a comparison predicate (e.g., $=, \leq, \ldots$); the predicate $\rhd$ (enabling) applied to an action term; or the predicate $\odot$ (occurrence) applied to an action term. A formula in H_i can be obtained by applying successively the connective $\Rightarrow$ and the temporal operators $\mathcal{U}$ (until) and $\mathcal{S}$ (since) to atomic formulae. The other temporal operators like *next* X, *sometime in the future* F and *always* G can be derived from these (cf. [6] for details). The communication logic C_i allows to express communication among several objects from the local viewpoint of object i. Formulae in $\mathcal{C}_i$ contain occurrences of actions for object i like, e.g., $\odot i.a \wedge \rhd i.b$.
We now give some examples of CATC system axioms. To increase their readability we will omit the local object identity. The next formula denotes the global interaction among a staff instance and the instance IG34, and it corresponds mainly to a one to one translation of the system specification (cf. Fig. 3). It is a communication formula from object Staff(Users(0)).

$$\begin{aligned}
&\odot\ \mathtt{Staff(Users(0)).createExperiment(100,nam,st,expNr)} \Rightarrow\\
&\quad \odot\ \mathtt{IG34.Applications(100).newExperiment(nam,st,expNr)}
\end{aligned}$$

If the above global interaction occurs it causes a further communication within the node IG34. This is captured by the next formula (cf. Fig. 2).

$\odot$ `IG34.Applications(100).newExperiment(nam,st,expNr)` $\Rightarrow$
$\odot$ `IG34.Applications(100).Experiments(expNr).createExp(nam,st)` $\wedge$
    `expNr = IG34.Applications(100).nextExpNr` $\wedge$
    X (`IG34.Applications(100).nextExpNr = expNr + 1`)

DTL is interpreted over labelled prime event structures. Prime event structures were first introduced by Winskel (for more details cf., e.g., [18]) and allow a nice and simple description of distributed computations as event occurrences together with a causal and a conflict relation between them. The causal relation implies a (partial) order among event occurrences, while the conflict relation allows to express choice. Events in conflict cannot belong to the same system or object run. Formally, a prime event structure is a triple $E = (Ev, \rightarrow^*, \#)$, where Ev is a set of events, $\rightarrow^*$ and $\#$ are binary relations on events called causality (a partial order) and conflict (irreflexive and symmetric) respectively. Two events which are neither in conflict nor related by causality are said to be concurrent. If there are no concurrent events the structure is called sequential. In order to be able to use event structures as a model for our objects, we have to attach to them some more information in the form of event labels. We introduce a labelling function $\mu : Ev \rightarrow \mathcal{P}(Ac)$, i.e., a total function attaching to each event the actions which occurrence it denotes. A prime event structure enriched by a labelling function is called a labelled prime event structure. Labelled prime event structures constitute a non-interleaving model of concurrency and are a fair choice when a denotational semantics is needed.
Objects in TROLL do not have intra-object concurrency, and are therefore modelled by sequential labelled event structures. Let (E_i, μ_i) be a model for the object instance $i \in Id$. A system model is obtained by composing the corresponding instance models. The idea of the composition construction is to identify shared communication events and leave all the other events concurrent. Figure 5 shows part of the CATC system model focusing on the communication between a staff (`Users(0)`) and an operator (`Users(1)`) instances. We represent events by boxes with the corresponding label (set of actions) written inside. Arrows between boxes denote the causality relation among events. The communication, as presented already in the OMTROLL communication diagram of Fig. 4, is done indirectly via experiments.

5 TROLL-WORKBENCH

The architecture and aims of the TROLL-WORKBENCH were described in [9]. In this paper we only want to give a brief description of its functionalities.

– OMTROLL **Editor**: This editor provides developers with different diagrams for modelling the system in OMTROLL *(community, communication and data type declaration diagrams)*. Figures 1 and 4 showed the specification of our example using the OMTROLL editor. An experience we gained from projects where we used OMTROLL and TROLL, is that specifiers tend to work switching between both notations. For this reason, an automatic translation can be

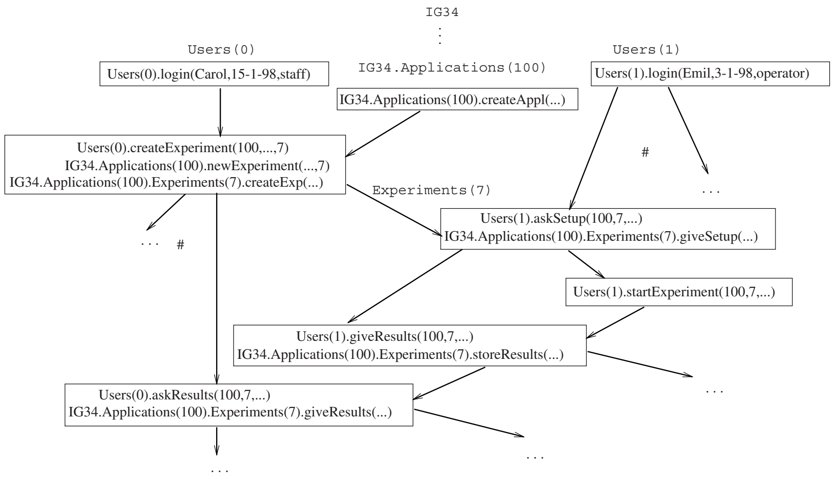
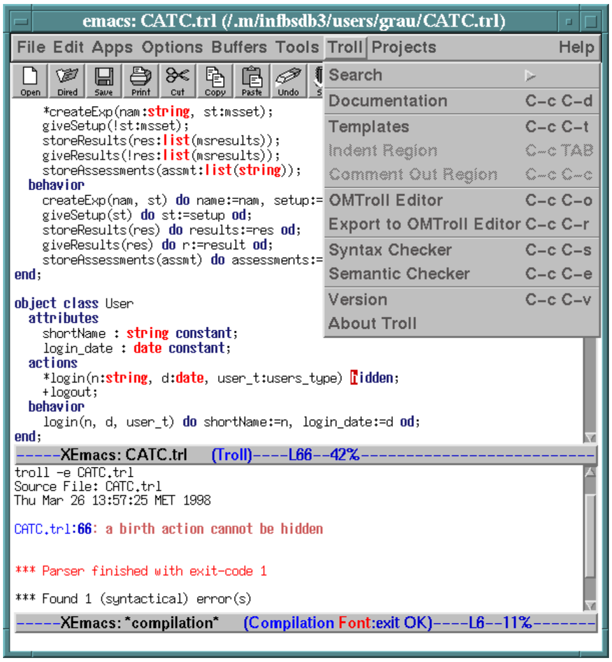

IG34
Users(0)
Users(0).login(Carol,15-1-98,staff)
IG34.Applications(100)
IG34.Applications(100).createAppl(...)
Users(1)
Users(1).login(Emil,3-1-98,operator)
Users(0).createExperiment(100,...,7)
IG34.Applications(100).newExperiment(...,7)
IG34.Applications(100).Experiments(7).createExp(...)
#
...
Experiments(7)
Users(1).askSetup(100,7,...)
IG34.Applications(100).Experiments(7).giveSetup(...)
... #
Users(1).startExperiment(100,7,...)
Users(1).giveResults(100,7,...)
IG34.Applications(100).Experiments(7).storeResults(...)
...
Users(0).askResults(100,7,...)
IG34.Applications(100).Experiments(7).giveResults(...)
...
...

Fig. 5. Modelling communication between a staff and an operator instances.

done between both notations, obtaining not only the freedom in the method, but also the consistency in the notations. At the moment the method does not allow a complete translation because there is not a direct correspondence between the behaviour part of Troll and the object behaviour diagram of OmTroll. Investigations towards a complete translation are future work.

– Troll **Editor**: For textual specifications a Troll language mode has been implemented in the *(X)Emacs* editors. It constitutes the front end of the system. Here other system tools can be invoked providing a common interface. Some of the new capabilities added to the editor are: management of specification projects, automatic generation of patterns, search for keywords through the project files, cross-referencing, automatic generation of documentation files, different font styles for reserved words and automatic indentation. The textual editor is depicted in Fig. 6.
– **Syntax Checker**: This tool checks whether the syntax of the specification meets the Troll syntax. The syntax checker creates a syntax graph from the specification which is used as internal structure for all remaining tools.
– **Semantic Checker**: The semantic checker analyses the specification looking for possible type errors and Troll specific semantics inconsistencies. The last task is not an easy one, because of the complex Troll rules regarding attributes and actions visibility in order to maintain object encapsulation principles.
– **Report Generator**: The report generator, including a pretty printer, allows to export the specification to a document. In this way the model and the documentation of the model can be contained in one document.
– **Animator**: For validation purposes, we are implementing an animator for Troll specifications. By animating specifications we mean the process of identifying a set of *scenarios*, where a scenario is a sequence of events which


emacs: CATC.trl (/.m/infbsdb3/users/grau/CATC.trl)
File Edit Apps Options Buffers Tools Troll Projects Help
Open Dired Save Print Cut Copy Paste Undo
Search
Documentation C-c C-d
Templates C-c C-t
Indent Region C-c TAB
Comment Out Region C-c C-c
OMTroll Editor C-c C-o
Export to OMTroll Editor C-c C-r
Syntax Checker C-c C-s
Semantic Checker C-c C-e
Version C-c C-v
About Troll

```
    *createExp(nam:string, st:msset);
    giveSetup(!st:msset);
    storeResults(res:list(msresults));
    giveResults(!res:list(msresults));
    storeAssessments(assmt:list(string));
  behavior
    createExp(nam, st) do name:=nam, setup:=
    giveSetup(st) do st:=setup od;
    storeResults(res) do results:=res od;
    giveResults(res) do r:=result od;
    storeAssessments(assmt) do assessments:=
end;

object class User
  attributes
    shortName : string constant;
    login_date : date constant;
  actions
    *login(n:string, d:date, user_t:users_type) hidden;
    +logout;
  behavior
    login(n, d, user_t) do shortName:=n, login_date:=d od;
end;
-----XEmacs: CATC.trl     (Troll)----L66--42%------------------------
troll -e CATC.trl
Source File: CATC.trl
Thu Mar 26 13:57:25 MET 1998

CATC.trl:66: a birth action cannot be hidden


*** Parser finished with exit-code 1

*** Found 1 (syntactical) error(s)
-----XEmacs: *compilation*     (Compilation Font:exit OK)----L6--11%------
```

Fig. 6. TROLL language mode for XEmacs. The syntax and semantics checkers are included in the editor.

may occur in the domain of the system, and testing them against the specification. By this way, we can see whether the specification meets the intended user requirements involved in these scenarios. In order to test a specification it is necessary to execute it. There are different approaches for executing a specification. We have chosen code generation rather than a TROLL interpretation because of two principal reasons. On the one hand, we have gained positive experiences from the CATC project, about how TROLL objects can be implemented using an object-oriented programming language and on the other hand, generating code could later make possible to use some parts of the animator for evolutionary prototyping purposes.

Implementation Aspects: For the implementation we are only using well–known public domain tools. A prototype of the TROLL-WORKBENCH including editors and checkers can be downloaded from our ftp site.

6 Experiences

The conceptual modelling, design and implementation of the CATC system for one of the three laboratories of the group 3.4 in the PTB has been done and we are reusing this model for the other ones [11]. We gained positive experiences

with using object-oriented formal techniques in our project especially in describing our universe of discourse. During this time, we tried to get an overview in specific sub-problem domains and developed a first rough global architecture. One advantage of using Troll was to achieve first a rather global view before considering details. Changes to the finer grained specification documents did not affect the global view. OmTroll community diagrams help us to improve the communication skills between developers and federal board employees in their understanding of technical and administrative processes. In the analysis phase, the focus is on providing a structured intuitive representation of the aspects that are relevant for the planned information system. This represention has to be unambiguous and precise, structured to support the management of the model complexity and independent of concrete implementations.
Without tool support, respecifying and redoing already developed system parts was over and over disappointing. Tools support turned out to be one key factor in order to avoid frustration when having to change a model again. We had regular meetings with all team members to propagate important design decisions concerning interfaces, but we spent a lot of time answering syntactical and semantical questions by using Troll. The fact that specification is the most important part of the work is also reflected by the amount of time used for modelling and implementation, respectively. On average 70% of the time was spent on specification purposes whereas the rest of the time was spent on implementation including integration. During analysis and design, we spent more than 30% of the time checking manually syntax and semantic mistakes in our specification. This was not always easy within 17000 lines of Troll code.
By refining the specification we also used Troll in the design phase. Our implementation environment was a relational database and had C++ as target language. Therefore, we defined general rules to transform structural aspects of Troll specifications to relational database schemes and to translate information about attributes, and actions into C++. The translation from compact Troll notation into C++ was straightforward. The factor between Troll and C++ lines of code was of 1:6.
Because we have not developed the different subsystems of CATC at the same time, we have started to integrate them using the Troll-WorkBench. Since we are now using tools for modelling CATC in other laboratories of the PTB, we have the possibility to let the developers share the diagrams. This helps them to discover ambiguities by defining interfaces. Overall, we strongly believe that the formal object oriented method has paid off.

7 Summary and Further Work

In this paper we have presented the Troll approach to conceptual modelling of information systems. Troll has been used in modelling an industrial application. A simplified extract of this application has been used herein to describe briefly the syntax and semantics of Troll. We have also shown the functionalities of a CASE environment for Troll. Very positive experiences have been

gained using TROLL in the industrial setting. We have shortly reported on these experiences.
Further research focuses on modularisation concepts. It became obvious that a higher level structuring mechanism than object classes is needed. This is mainly a result of the case study presented above, where 17000 lines of TROLL code were produced. Besides this we are going to provide reusable modules in a catalogue which will be integrated in the TROLL-WORKBENCH. First investigations on the language level are presented in [5]. Work towards a logical and semantical foundation of such concepts has been covered among others in [14,15].
Moreover, we are also considering the support of other kinds of communication mechanisms for TROLL. At the specification level it is convenient to be able to abstract from too many details, therefore in some cases asynchronous communication could be preferable to synchronous one. An advantage of asynchronous communication is also given by the higher level of concurrency that can be achieved and is especially important for distributed systems. Considerations on the logic and denotational semantics for asynchronous communication can be found among others in [15].
A further research direction we are currently considering is the development of a TROLL tool for model checking. Usually verification is done at the end phases of the development process verifying a program or a hardware description. With our tool, we aim at checking the global system behaviour in earlier phases. The expected behaviour of our system can be expressed as a set of DTL formulae. The intention is to check a system model, represented by a state transition system, against this set of formulae.
Finally, since some parts of the CATC project include the specification of real time aspects, we are also considering the extension of TROLL to cope with them.

References

1. E. Cusack, S. Rudkin, and C. Smith. An Object-Oriented Interpretation of LOTOS. In S. Vuong, editor, *Formal Description Techniques II, FORTE'89*, pages 211–226. North–Holland, Amsterdam, 1990.
2. G. Denker and P. Hartel. TROLL – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics. Informatik-Bericht 97–03, Technische Universität Braunschweig, 1997.
3. P. Du Bois. *The Albert-II Language – On the specification and the Use of a Formal Specification Language for Requirements Analysis.* PhD thesis, Facultes Universitaires Notre–Dame de la Paix, Namur, Belgium, 1995.
4. E.H. Dürr and J.v. Katwijk. VDM++, A Formal Specification Language for Object–Oriented Design. In *Proceedings of TOOLS7 (Technology of Object-Oriented Languages and Systems)*. Prentice Hall, 1992.
5. S. Eckstein. Towards a module concept for object oriented specification languages. In J. Bārzdiņš, editor, *Proc. 3rd Int. Baltic Workshop on Databases and Information Systems, Riga, Latvia, April 15-17, 1998*, pages 180–188, 1998.
6. H.-D. Ehrich, C. Caleiro, A Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic, 1998.

7. H.-D. Ehrich and P. Hartel. Temporal Specification of Information Systems. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering, Proc. Int. Workshop in Honor of C.S. Tang,Beijing, 14-15 August 1995*, pages 43–71. World Scientific, 1996.
8. R. Feenstra and R. Wieringa. LCM 3.0: A Language for Describing Conceptual Models - Syntax Definition. Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1993.
9. A. Grau and M. Kowsari. A Validation System for Object-Oriented Specifications of Information Systems. In R. Manthey and V. Wolfengagen, editors, *Proc. of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97), St. Petersburg, 2-5 September.* EWiC, Springer, 1997.
10. Paul Harmon and Mark. Watson. *Understanding UML- The Developer's Guide.* Morgan Kaufmann, San Francisco, California, 1998.
11. P. Hartel, G. Denker, M. Kowsari, M. Krone, and H.-D. Ehrich. Information systems modelling with TROLL formal methods at work. *Information Systems*, 22(2-3):79–99, 1997.
12. T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised Version of the Modelling Language Troll (Version 2.0). Informatik-Bericht 94–03, Technische Universität Braunschweig, 1994.
13. R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Troll – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, April 1996.
14. J. Küster Filipe. Modelling Parameterisation in Concurrent Object Systems. *Logic Journal of the IGPL*, 5(6):877–879, November 1997. In Conference Report: Workshop on Logic, Language, Information and Computation (WoLLIC)'97, Fortaleza, Ceará, Brazil, August 20-22.
15. J. Küster Filipe. On a Distributed Temporal Logic for Modular Object Systems. Technical Report 98-06, Technical University Braunschweig, 1998.
16. K. Lano. Z++: an Object-Oriented Extension to Z. In J. Nicholls, editor, *Z Users Workshop: Proc. 4th Annu. Z User Meeting*, pages 151–172. Workshops in Computing. Springer-Verlag, Berlin, 1991.
17. K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. Temporal Logics for Communicating Sequential Agents. *Int. Journal of Foundations of Computer Science*, 3(2):117–159, 1992.
18. M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, part 1. *Theoretical Computer Science*, 13:85–108, 1981.
19. O. Pastor, E. Insfran, V. Pelechano, J. Romero, and J. Meseguer. OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods. In A. Olive and J. A. Pastor, editors, *9th International Conference on Advanced Information Systems Engineering (CAiSE'97), Barcelona*, pages 145–158, June 1997.
20. A. Ruiz-Delgado, D. Pitt, and C. Smythe. A Review of Object-Oriented Approaches in Formal Methods. *The Computer Journal*, 38(10):777–784, 1995.
21. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modeling and Design.* Prentice Hall, New York, 1991.
22. A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.