Recent Trends in Data Type Specification (ADT'92)
H. Ehrig, F. Orejas (Eds.), Springer, Berlin, LNCS 785, pp. 173-187 (1994)

# Translating TROLL *light* Concepts to Maude

G. Denker, M. Gogolla*

Technische Universität Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, D–38023 Braunschweig, Germany
e–mail: {denker|gogolla}@idb.cs.tu-bs.de

**Abstract.** The specification language TROLL *light* is designed for the conceptual modeling of information systems. Maude is a logic programming language, which unifies the two paradigms of functional and concurrent object-oriented programming. Because of the very similar features offered by both languages, we present a translation from TROLL *light* concepts into the Maude language in order to compare the languages. Apart from presenting the translation, the languages are briefly described and illustrated by examples.

## 1 Introduction

Various approaches for the specification of complex software systems have appeared recently, for example: Specification of functions (VDM, Z [Jon86, BHL90]), abstract data types [EM85, EGL89, EM90, Wir90], predicate logic and extensions like temporal and modal logic [MP91], semantic data models [HK87], and process specification (CCS [Mil80], CSP [Hoa85], petri nets [Rei85]).

Our main concern is the conceptual modeling of information systems, and it seems that in isolation all these approaches are not appropriate for this task. The result of the conceptual modeling process should be a first precise characterization of the part of the world to be modeled, the so-called Universe of Discourse (UoD). If we look at an example UoD we find out that it consists of complex structured entities with time-varying behavior. Such entities usually exist concurrently and interact with each other. This is where the idea of object-oriented specification gives rise to an approach in order to represent the real-world entities as formal objects in a specification language. We work with the language TROLL *light* [CGH92, GCH93] which is a dialect of TROLL [JSHS91]. Both languages are based on recent results for semantic foundations of object-oriented specification [SE91, EGS92, ESS92, EDS93]. Structure as well as behavior of real-world entities can be characterized in TROLL *light*. Other languages for the specification of objects are for example OBLOG [SSG+91], CMSL [Wie91], and GLIDER [DBRW91, CJO92], OS [Bre91], FOOPS [GM87], MONDEL [BBE+90].

* Work reported here was partially supported by CEC under ESPRIT-II Basic Research Working Group No. 6112 COMPASS, under ESPRIT BRA WG 6071 IS-CORE, and by the German Ministry for Research and Technology (BMFT) under Grant No. ITS 9002 D/ 01 IS 203 D KORSO.

Maude [Mes92b] is a logic programming language, which unifies the two paradigms of functional and concurrent object-oriented programming. As a language feature Maude offers object-oriented modules through which classes can be formalized with their structural and behavioral characteristics.

Because of the very similar features offered by both languages, we present a translation from TROLL *light* concepts into the Maude language with examples and informally give a general translation schema. In some cases there exists only a small difference in the expressiveness of the concepts, whereas in other cases the difference results from special design decisions. Apart from presenting the translation, the languages are briefly described and illustrated by examples.

The structure of our paper is as follows. In Sect. 2 the concepts of our specification language TROLL *light* are described in more detail. In Sect. 3 we explain the basic features of the language Maude and give new representations for object configurations and in Sect. 4 for identifiers. Moreover we illustrate by examples the main ideas of translating TROLL *light* into Maude in Sect. 4 and give a general translation schema. Sect. 5 closes the paper by giving some concluding remarks.

## 2 Concepts of TROLL *light*

TROLL *light* is a language for the specification of objects. In general, TROLL *light* objects are associated with a possibly infinite number of attribute names, a possibly infinite number of sub-object names, and a finite number of object machine states which control object behavior. In a given state, a finite number of attribute names will take defined values, a finite number of sub-object names will be connected with defined sub-objects, and the object machine (o-machine in the sequel) will be in a certain state.

The possible attribute names, the possible sub-object names, and the o-machine states are fixed by object descriptions, which are called templates in TROLL *light*. Besides that a template also determines how state transitions between object states are to be carried out, i.e., templates comprise object behavior. State transitions are caused by event occurrences. Here the events which may occur during the life of an object have to be fixed by a finite number of event generators within a template.

In a given o-machine state an event occurrence is admissible only if a corresponding o-machine state transition rule is provided within a template. A template always contains a finite number of such o-machine state transitions. Hence an o-machine can be viewed as a finite event-driven sequential automaton. The effect of event occurrences on attributes are fixed by valuation rules, and the effect of event occurrences on other objects is fixed by interaction rules.

Further features potentially addressed within a template include means for the description of (static) integrity constraints, derived attributes, and preconditions for o-machine state transitions or for the application of valuation rules and interaction rules.

All in all a template has the following structure:

```
TEMPLATE
  DATA TYPES    decl. of data types used in current template
  TEMPLATES     decl. of other templates used in current template
  SUBOBJECTS    decl. of sub-object names
  ATTRIBUTES    decl. of attribute names
  EVENTS        decl. of event generators
  CONSTRAINTS   conditions on object states
  VALUATION     effect of event occurrences on attributes
  DERIVATION    derivation rules for derived attributes
  INTERACTION   synchronization of events in different objects
  BEHAVIOR      specification of allowed life cycles
END TEMPLATE
```

In the sequel we show by a small example how information systems may be specified by writing templates.

Let us assume that an information system is to be described, which shall contain data about author objects. For every author the name, the date of birth, and the number of books sold by year have to be stored. An author may change the name only once in the life. An appropriate TROLL *light* specification would be:

```
TEMPLATE Author
  DATA TYPES   String, Date, Nat;
  ATTRIBUTES   Name:string; DateOfBirth:date;
               SoldBooks(Year:nat):nat;
  EVENTS       BIRTH create(Name:string, DateOfBirth:date);
                     changeName(NewName:string);
                     storeSoldBooks(Year:nat, Number:nat);
               DEATH destroy;
  VALUATION    [create(S,D)] Name=S, DateOfBirth=D;
               [changeName(S)] Name=S;
               [storeSoldBooks(Y,N)] SoldBooks(Y)=N;
  BEHAVIOR     PROCESS authorlife1 =
                 ( storeSoldBooks -> authorlife1 |
                   changeName -> authorlife2 |
                   destroy -> POSTMORTEM );
               PROCESS authorlife2 =
                 ( storesSoldBooks -> authorlife2 |
                   destroy -> POSTMORTEM );
               ( create -> authorlife1 );
END TEMPLATE;
```

Attributes reflect observable properties of author objects, and events denote events which can happen during the life of an author. The effect of events on attributes is specified by valuation rules. Possible life cycles of authors are fixed by process descriptions. The corresponding o-machine is visualized in Fig. 1.

We may assume that the information system will be populated by lots of authors

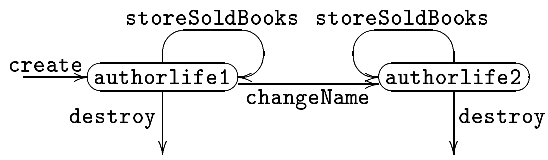


**Fig. 1.** Behavior of authors

so that these authors must be managed by a higher-level object. To this end usually classes are introduced in many object-oriented approaches. TROLL *light* does not support an explicit class concept. Classes are viewed as composite objects instead, and therefore classes have to be described by templates again.

```
TEMPLATE Authorclass
  DATA TYPES    String, Date, Nat;
  TEMPLATES     Author;
  SUBOBJECTS    Authors(No:nat):author;
  ATTRIBUTES    NumberOfAuthors:nat;
  EVENTS        BIRTH createClass;
                      addObject(No:nat,Name:string,
                                DateOfBirth:date);
                      removeObject(No:nat);
                DEATH destroyClass;
  VALUATION     [createClass] NumberOfAuthors=0;
                [addObject] NumberOfAuthors=NumberOfAuthors+1;
                [removeObject] NumberOfAuthors=NumberOfAuthors-1;
  INTERACTION   addObject(N,S,D) >> Authors(N).create(S,D);
                removeObject(N) >> Authors(N).destroy;
END TEMPLATE;
```

An object of type `Authorclass` will hold many author objects as private components or sub-objects. In this example a natural number serves as a key for the identification of sub-objects. Please note that in contrast to many other data models, in TROLL *light* keys need not be related to attributes of objects they identify. Adding an object to an author class means to create a new sub-object, and removing an object from an author class means to destroy the sub-object.

An object of type `Authorclass` may be sub-object of a higher object again, and so on. For example, a template `Libraryworld` may be specified which contains a class of authors, a class of books, and two libraries as sub-objects (see [CGH92]).

## 3 The logic programming language Maude

The programming language Maude is a direct result from the requirement to integrate concurrent programming with paradigms like functional and object-oriented programming. Maude's semantic framework is conditional rewriting logic which is a logic to reason about change in a concurrent system. Since rewriting is naturally concurrent, this logic deals adequately with one intrinsic part of object-oriented systems, namely concurrency.

Modules in Maude are theories in rewriting logic so that concurrent object-oriented computation coincides with logical deduction. Deduction is done modulo some structural axioms in order to catch in an abstract way the essential aspects of object-oriented systems. We will illustrate this by examples later on.

Maude contains three kinds of *modules*: **functional modules**, **system modules**, and **object-oriented modules**. The functional sub-language of Maude is similar to OBJ3 [GW88]. The syntax and semantics of object-oriented modules is reducible to that of system modules; they only provide a notational convenience for object-oriented applications by supporting appropriate language features. In contrast to functional modules system modules provide the specification of non-functional behavior. Maude has a parameterization mechanism and also supports multiple class inheritance.

In the sequel we illustrate the main concepts of Maude by examples which will be needed subsequently for the translation. All given modules are specified in order to translate the running example from Sect. 2. Therefore we change original Maude notions and give new representations, such as for the `CONFIGURATION` and `ID` modules below.

First we will have a look at **functional modules**. Consider the following Maude functional module specifying the abstract data type `SETOFTUPLE` (Maude keywords are underlined):

```
fmod SETOFTUPLE is
  protecting TUPLE .
  sort SetOfTuple .
  subsort Tuple < SetOfTuple .
  op _in_: Tuple SetOfTuple -> Bool .
  op _U_: SetOfTuple SetOfTuple -> SetOfTuple [assoc id:0 comm].
  op _\_: SetOfTuple Tuple -> SetOfTuple .
  vars S S': Set .
  vars T T': Tuple .
  sct S U T: SetOfTuple if not T in S .
  eq T in 0 = false .
  eq T in (T' U S)= if T == T' then true else T in S fi .
  eq S\0 = S .
  eq 0\T = 0 .
  eq (S U T')\T = if T == T' then S else T' U (S\T) fi .
endfm
```

The given functional module defines sets of tuples (e.g., of natural numbers) with the union, minus, and element of functions. The union function is declared to be commutative, associative, and has `0` as the identity element. Rewriting is done modulo these structural axioms so that besides the sort constraint sct we need no further equations. We assume that the module `TUPLE` is specified somewhere in the way that it protects the natural numbers and provides at least a constructor function `[ , ]: Nat Nat -> Tuple` and a selector function `first: Tuple -> Nat` returning the first component of a tuple. These operations will be needed in the next example. The line protecting `TUPLE` imports this module as a submodule of `SETOFTUPLE`, and asserts that no new data elements to old sorts (here `Tuple`) will be introduced (no "junk") and that the original equality relation between data elements will not be altered (no "confusion"). Moreover sort declarations, which describe the data sorts manipulated by this module, and subsort declarations, which describe how the sorts are related, are given: sort `SetOfTuple` and subsort `Tuple < SetOfTuple`. Functions with their domain and co-domain sorts provided by the module (here union function, minus function, and element of function for sets) are introduced by the keyword op . Variables used for defining equations are declared with their corresponding sorts in the vars part. The sort constraint sct accomplishes that the addition of a tuple to a set of tuples yields a set (of tuples) if and only if the tuple has not been already element of the original set. Equations eq establish the effect of the operations.

Up to now importing modules by means of protecting introduces no semantic modifications. But there exists also the possibility that importing (possibly renamed) modules is accompanied with semantic modifications, i.e., new data elements to old sorts could occur and/or new relationships between old data elements could be defined. This can be done via an using module importation. We will explain this in the following example:

```
fmod MAPNAT2NAT is
  using SETOFTUPLE *( sort SetOfTuple to MapNat2Nat) .
  op U: MapNat2Nat Tuple -> MapNat2Nat .
  ...
endfm
```

The specification *uses* the `SETOFTUPLE` module in such a way that the sort `SetOfTuple` is renamed to `MapNat2Nat`. In order to keep the example small we omit details. The functional dependencies forcing sets of tuples of natural numbers to become a map between natural numbers have to be specified via suitable equations. Through these new equations semantic modifications are introduced.

As another possibility we might want to specify modules which do not behave in a functional way. In fact non-functional behavior is intrinsic to object-oriented systems. For example, several reachable configurations (or say system states) are possible if we start from an initial state. This kind of behavior is supported by system and object-oriented modules.

As a first example we just want to specify such system configurations. The state of a concurrent system is represented as a multi-set of objects and messages, where objects are described as terms giving their class membership and their attribute values. A system configuration made up of concurrently existing objects and messages is formalized via a **system module**:

```
mod CONFIGURATION is
  protecting ATTRIBUTES ID.
  sorts Configuration Object Msg .
  subsorts Object Msg < Configuration .
  op __: Configuration Configuration -> Configuration
                                         [assoc id: 0 comm] .
  op <_:_|_>: OId CId Attributes -> Object .
  var IDSET : IdSet .
  var OID : OId .
  var CID : CId .
  var ATTS : Attributes .
  sct <OID:CID|ATTS> : Object if not OID in IDSET .
  rl <OID:CID|ATTS> => <OID:CID|ATTS> addId(OID,IDSET) .
endm
```

Two modules are imported by `CONFIGURATION`: we assume the `ATTRIBUTES` module to be specified somewhere else and specify the `ID` module in the following section. The `__` operator, which is commutative, associative, and has `0` (representing a system without objects and messages) as its identity element, structures the state of a concurrent object-oriented system as a multi-set of objects and messages. Deduction, that is to say computation, is done modulo these given axioms. The "object-generating" operation `<_:_|_>` demands an object identifier, a class identifier, and the specific attributes (name-value pairs) of the new object. Object and class identifiers are subsorts of `Id` as specified later on. The sort constraint in combination with the specified rule assures that new objects are only created if their identifier is unique in the sense that before the creation it is not in the set of current object identifiers and afterwards it is added to this set. To assure that `IDSET` always contains all identifiers of currently existing objects this rule has to be applied each time new objects are created. The `addId` operation is specified in the `ID` module as we will see in the following. Here we changed the original notion of Meseguer (cf Sect. 4 in [Mes92a]) in order to deal with our identification mechanism more adequately. Meseguer proposes a system-wide counter as identification mechanism what is not appropriate for our approach since we use logical names as object identifiers. More details are given in the next section where the module `ID` is defined.

**Object-oriented modules** only provide a notational convenience compared with system modules. They support some common object-oriented notions, like object, message, class, etc., but their syntax and semantics can be reduced to that of system modules (for details see [Mes92a], Sect. 4.2). By means of object-oriented modules, classes can be defined, in which the structural and behavioral aspects of class members are specified via attributes and messages, respectively. The effects of messages (with or without interaction) on objects are determined by conditional term rewriting rules. Object-oriented modules look like the following:

```
omod name is
  extending module names
  protecting (renamed) module names
  class class name | attribute decl.
  subclass class names
  msgs message decl.
  vars variable decl.
  rl rule decl.
endom
```

where all italic items have to be specified by the user. The protecting part of object-oriented modules corresponds to that of functional modules. This means no semantic modifications are introduced. The `extending` mechanism in combination with rules can be used to express inheritance aspects. More exhaustive examples for object-oriented modules will follow later on.

Maude offers a lot of other language facilities. For example modules can be parameterized and class inheritance is directly supported by its order-sorted type structure in object-oriented modules. We will not go into details since we do not need these facilities for our purpose.

## 4 TROLL *light* to Maude – Translation by example

The translation of TROLL *light* concepts to Maude is relevant to implementation aspects. After modeling the UoD as a system of concurrently existing and interacting objects formalized with TROLL *light*, we yield a collection of TROLL *light* templates. Based on this abstract description steps towards more concrete descriptions yielding an implementation should be done. It is conceivable to take Maude [Mes92a] as such an implementation basis. Programming languages used as an implementation basis for TROLL *light* should both be able to deal with the concepts of object-orientation and be efficiently implementable. Maude among other languages seems to fulfil these requirements.

Now we explain our ideas of translating TROLL *light* concepts to Maude with the examples given in Sect. 2. This means we will work with `Authorclass`, `Author` as the given template collection where `Authorclass` is the schema template. Afterwards we will mention some general aspects of the translation.

We already mentioned the identifier problem in the previous section. The `CONFIGURATION` module deals with different kinds of identifiers, and since object-oriented modules define classes an identification mechanism for class members is required. On the other hand we set up a specification in TROLL *light* as a collection of templates, where the identification mechanism via logical names is user-definable. Since our aim is to translate TROLL *light* templates to object-oriented modules in Maude we have to deal with the identifier problem adequately. At first sight this seems to be difficult since there exists no global identifier sort in TROLL *light*. But all possible object identifier, or logical names in our notion, are defined in the `SUBOBJECTS` part. As a solution we suggest the introduction of a module `ID` which provides among other things an object identifier sort `OId` and a class identifier sort `CId`. Possible sub-object names are presented via suitable operations with co-domain sort `OId`. We illustrate this with an example.

The specified object sorts `Authorclass` and `Author` will correspond to class types in Maude as we will see in a while. The possible object identities `Authors(No:nat)` as established in the `SUBOBJECTS` part will be reflected by a module `ID` which makes object identifiers available. In the example we yield the following functional module:

```
fmod ID is
  protecting SET*( sort Set to IdSet, sort Elt to Id,
                   op _U_ to addId) .
  sorts Id OId CId AuthorclassId AuthorId IdSet .
  subsorts OId CId < Id .
  subsort AuthorId AuthorclassId < OId .
  op INIT: -> AuthorclassId .
  op Authors: AuthorclassId Nat -> AuthorId .
endfm
```

Since in TROLL *light* a given template collection represents a hirarchy of templates fixed through sub-object relationships there is always one distinguished template, the so-called schema template, of which all other templates are direct or indirect sub-templates. `INIT` is a special operation to generate an instance of the given schema template, in our example the `Authorclass` template. Class identifiers are automatically defined by object-oriented modules as we will see later on.

O-machine states (e.g., `authorlife1` etc.) are essential for the description of object behavior. Maude does not provide a special feature to express constraints on objects behavior as we are used to it in TROLL *light*. In order to give a correct translation we make o-machine states explicit (by defining a sort `OMState` plus adequate generator functions) and use them later in rules in object-oriented Maude modules to express the dependencies between o-machine states and possible object state transitions. In the given example `authorlife1`, `authorlife2`, `authorclasslife`, and `POSTMORTEM` are defined as possible object-machine states. They are also reflected via a functional module:

```
fmod OMSTATE is
  sorts OMState AuthorState AuthorclassState .
  subsorts AuthorState AuthorclassState < OMState .
  op authorlife1 : -> AuthorState .
  op authorlife2 : -> AuthorState .
  op authorclasslife : -> AuthorclassState .
  op POSTMORTEM : -> OMState .
endfm
```

The above translation, yielding the modules `CONFIGURATION`, `ID`, and `OMSTATE`, is necessary in general for the rest of the work. In the following each TROLL *light* template will be translated to an object-oriented Maude module and each Maude module received in this way imports at least the first two modules. `OMSTATE` only has to be imported if o-machine states are specified.

We proceed with translating `Author` and `Authorclass`, respectively, to object-oriented Maude modules. We will start with the `Author` template which is translated into the following Maude module:

```
omod AUTHOR is
  extending CONFIGURATION .
  protecting ID OMSTATE MAPNAT2NAT STRING NAT DATE .
  class Author | Name: String,
                 DateOfBirth: Date,
                 SoldBooks: MapNat2Nat,
                 omstate: AuthorState .
  msg create : AuthorId String Date -> Msg .
  msg changeName : AuthorId String -> Msg .
  msg storeSoldBooks : AuthorId Nat Nat -> Msg .
  msg destroy : AuthorId -> Msg .
  var OIDSET : OIdSet .
  var AID : AuthorId .
  var MNN : MapNat2Nat .
  vars Y N: Nat .
  var D : Date .
  vars S S': String .
  rl create(AID,S,D) =>
      <AID:Author|Name:S,DateOfBirth:D,omstate:authorlife1> .
  rl changeName(AID,S) <AID:Author|Name:S',omstate:authorlife1>
      => <AID:Author|Name:S,omstate:authorlife2> .
  rl storeSoldBooks(AID,Y,N)
      <AID:Author|SoldBooks:MNN,omstate:authorlife1>
      => <AID:Author|SoldBooks: MNNU[Y,N],omstate:authorlife1> .
  rl storeSoldBooks(AID,Y,N)
      <AID:Author|SoldBooks:MNN,omstate:authorlife2>
      => <AID:Author|SoldBooks: MNNU[Y,N],omstate:authorlife2> .
  rl destroy(AID) <AID:Author|ATTS> => 0.
endom
```

First of all we have to import the `CONFIGURATION` and `ID` modules to each object-oriented module which we receive as a translation from a TROLL *light* template. The essential notions of object, message, and identifier are necessary for each of them. As soon as o-machine states are introduced in the TROLL *light* template we also have to import `OMSTATE`. Furthermore, for the author template we need the sort `MapNat2Nat`, specified in the previous section, in order to express the parameterized attribute `SoldBooks(Year:nat):nat`, which could also be viewed as an attribute `SoldBooks:set(tuple(nat,nat))` with additional constraints expressing the functionality implicit in the former attribute expression. The sort `MapNat2Nat` comprises the required functional dependency. The sorts `String`, `Date`, and `Nat` which are needed for several attribute and message definitions are protected by module importation.

By means of the object-oriented Maude module `AUTHOR` a class `Author` is defined, and instances are described via four attributes: `Name`, `DateOfBirth`, `SoldBooks`, and `omstate`. Obviously we have to explicitly model the o-machine state as an attribute in order to adequately express dependencies between o-machine states and possible state transitions in Maude. In contrast to that these states are inherent in TROLL *light* specifications, but not visible. Though in TROLL *light* the object's state is given through its attribute values, its subobjects, and the current o-machine state only the attribute values and subobjects are directly observable. However, it is possible to deduce the current o-machine state from a given fixed life cycle.

TROLL *light* events are translated more or less one to one to messages in Maude. We only have to follow the rule that each message requires an object identifier, because we describe a class of objects, where single objects are distinguished via their identifiers. This also holds for the special `BIRTH` and `DEATH` events, so that all events set one more parameter: an Maude object identifier.

In Maude all used variables have to be declared in the <u>`vars`</u> section. We do without variable declarations in TROLL *light* templates, because the right sorts always can be determined from the defined attributes and events, and no difficulties are to be feared.

The remaining TROLL *light* parts to translate, i.e., `VALUATION`, `BEHAVIOR`, and, for `Authorclass`, `INTERACTION` must all be modeled by rules in Maude modules. So the first rule connected with the `BIRTH` event not only says that after the occurrence of `create(AID,S,D)` a new object of type `Author` with identifier `AID`, and with attributes `Name=S` and `DateOfBirth=D` is generated (something which is expressed in the valuation part of the TROLL *light* template), but also that afterwards the o-machine state is equal to `authorlife1` (what is stated in the behavior part of the corresponding TROLL *light* template).

Combining the valuation formula for the `changeName` event of the TROLL *light* specification with the stated behavior, i.e., `changeName` can only occur if the o-machine state is equal to `authorlife1`, gives the second rule. Similar considerations lead to the third and fourth rule about the `storeSoldBooks` event. The last rule states that a system consisting of an author `<AID:Author|ATTS>` and an event which destroys this author object proceeds to a system without any object and event. Please note that `0` is the identity element of the `__` operation in `CONFIGURATION`.

Now we translate the `Authorclass` template. New aspects are introduced through the specified sub-objects part in conjunction with the interaction formulas.

```
omod AUTHORCLASS is
  extending CONFIGURATION .
  protecting ID AUTHOR STRING NAT DATE .
  class Authorclass | NumberOfAuthors: Nat .
  msg createClass : AuthorclassId -> Msg .
  msg addObject : AuthorclassId Nat String Date -> Msg .
  msg removeObject : AuthorclassId Nat -> Msg .
  msg destroyClass : AuthorclassId -> Msg .
  var ACID : AuthorclassId .
  ...
  rl createClass(ACID) =>
     <ACID:Authorclass|NumberOfAuthors:0> .
  rl addObject(ACID,N,S,D) create(Authors(ACID,N),S,D)
     <ACID:Authorclass|NumberOfAuthors: N' =>
     <ACID:Authorclass|NumberOfAuthors: N'+1>
     <Authors(ACID,N):Author|Name:S,DateOfBirth:D,
       omstate:authorlife1> .
  rl removeObject(ACID,N) destroy(Authors(ACID,N))
     <ACID:Authorclass|NumberOfAuthors: N'>
     <Authors(ACID,N):Author|ATTS>
     => <ACID:Authorclass|NumberOfAuthors: N'-1>
  rl destroyClass(ACID) <ACID:Authorclass|NumberOfAuthors: N'> => 0.
endom
```

We will focus our interest especially on interaction formulas and how they are translated. If we have a look at the `Authorclass` template, we would translate the valuation formula for `addObject`, corresponding to the considerations already done, like this:

```
rl addObject(ACID,N,S,D) <ACID:Authorclass|NumberOfAuthors:N'> =>
   <ACID:Authorclass|NumberOfAuthors:N'+1> .
```

But because of the given interaction formula

```
addObject(N,S,D) >> Authors(N).create(S,D)
```

we have to adapt this rule to

```
rl addObject(ACID,N,S,D) create(Authors(ACID,N),S,D)
   <ACID:Authorclass | NumberOfAuthors: N'> =>
   <ACID:Authorclass | NumberOfAuthors: N'+1>
   <Authors(ACID,N):Author|Name:S,DateOfBirth:D,omstate:authorlife1>.
```

We will speak in the sequel of "rule adaption" caused by interaction formulas. The rule for the `removeObject` event is produced analogously.

At this point we loose some information of the corresponding TROLL *light* template collection. The given template collection {`Authorclass, Author`} with `Authorclass` as its schema template implies that authors can only be created and destroyed local to an author class. This kind of implicit functionality cannot be expressed with Maude. This seems to be a difference originated by our special design decision that systems are built as a complex object, an instance of the schema template, of which all the other objects are sub-objects in some way.

In summary we can give the following general translation schema:

$$\begin{aligned}
\text{object sorts} &\rightarrow \text{class types}\\
\text{sub-object symbols} &\rightarrow \text{constructor functions for identities}\\
\text{attributes} &\rightarrow \text{attributes}\\
\text{events} &\rightarrow \text{messages (one more parameter)}\\
\text{behavior} &\rightarrow \text{object machine states + rules}\\
\text{interaction} &\rightarrow \text{“rule adaption”}
\end{aligned}$$

Object sorts become class types in Maude since each TROLL *light* template which defines an object sort is translated to an object-oriented Maude module which defines a class. Sub-object symbols provide an identification mechanism for instances via logical names. We reflected this via the `ID` module. It is always possible to determine the correct `ID` module from a given template collection in the following way. Appropriate identifier sorts have to be provided for each template; all these identity sorts will be subsorts of `OId`. Each sub-object symbol corresponds to a constructor operation for identities, where `INIT` has to be added as a constructor function for a schema template instance. TROLL *light* events can be directly translated to Maude messages, and we only have to pay attention that for each event one more parameter is needed to indicate the class instance the event refers to. The translation of TROLL *light* attributes is also easy. They are mapped one to one to Maude attributes, where an additional attribute of sort `OMState` must be included if o-machine states are specified in the behavior part.

Although most TROLL *light* concepts are expressed by Maude features some concepts like `CONSTRAINTS` or `DERIVATION` or even more complex `VALUATION` or `INTERACTION` formulas remain to be analyzed. We comment on the chances to translate full TROLL *light* to Maude in the next section.

## 5 Conclusions

Some concepts of TROLL *light* specifications have not been considered up to now. Let us have a look at `CONSTRAINTS` and try to give a taste what kind of formulas we have to deal with and how they will be translated. Maybe we want to specify that author classes are required to have at least one author. We can do this in TROLL *light* as follows

```
CONSTRAINTS CNT(Authors)>0
```

`CNT` is a predefined aggregate function which counts all elements of a set. At each point of time only a finite number of sub-objects exist and the given formula restricts all possible author class object states in the way that at least one element has to be in the set with all author identifiers. To translate this we have to provide an operation that counts the number of identifiers of sort `AuthorId` in the set `IdSet`. Surely this is manageable if we introduce such an operation in the `ID` module, but perhaps the translation becomes less direct or intuitive. Similar considerations for derivation formulas and more complex interaction or valuation formulas let us presume that Maude's language features are expressible enough to cover all TROLL *light* concepts.

### Acknowledgements

Thanks to our KORSO colleagues Stefan Conrad, Hans–Dieter Ehrich, Rudolf Herzig, and Nikolaos Vlachantonis for fruitful discussions. Futhermore, the comments and questions of our SG1-IS-CORE colleagues helped a lot to improve this paper.

## References

[BBE+90] G. v. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval, and N. Williams. Mondel: An Object-Oriented Specification Language. Département d'Informatique et de Recherche Opérationnelle, Publication 748, Université de Montréal, 1990.

[BHL90] D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors. *VDM'90: VDM and Z — Formal Methods in Software Development*. Springer, LNCS 428, 1990.

[Bre91] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*. Springer, LNCS 562, 1991.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.

[CJO92] S. Clerici, R. Jimenez, and F. Orejas. Semantic Constructions in the Specification Language GLIDER. Workshop on Abstract Data Types, 1992.

[DBRW91] E. Dubois, P. Du Bois, A. Rifaut, and P. Wodan. *GLIDER Manual*. Facultés Universitaires de Namur, Namur (B), 1991. ICARUS Deliverable.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. TAPSOFT'93: Theory and Practice of Software Development*, pages 453–467. LNCS 668, Springer, Berlin, 1993.

[EGL89] H.-D. Ehrich, M. Gogolla, and U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen – Eine Einführung in die Theorie*. Teubner, Stuttgart, 1989.

[EGS92] H.-D. Ehrich, M. Gogolla, and A. Sernadas. Objects and their Specification. In M. Bidoit and C. Choppy, editors, *Proc. 8th Workshop on Abstract Data Types (ADT'91)*, pages 40–65. Springer, Berlin, LNCS 655, 1992.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, Berlin, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Modules and Constraints.* Springer, Berlin, 1990.

[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", Ulm (Germany)*, pages 1–19. Springer IFB 303, 1992.

[GCH93] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, pages 17–32. Springer, Berlin, LNCS 722, 1993.

[GM87] J. A. Goguen and J. Meseguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics . In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

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

[HK87] R. Hull and R. King. Semantic Database Modelling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs (NJ), 1985.

[Jon86] C.B. Jones. *Systematic Software Developing Using VDM.* Prentice-Hall, Englewood Cliffs (NJ), 1986.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[Mes92a] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegener, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency.* MIT Press, 1992. To appear.

[Mes92b] J. Meseguer. Conditional Rewriting as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–156, 1992.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer, Berlin, 1980.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems; Specification.* Springer-Verlag, New York, 1991.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Springer, Berlin, 1985.

[SE91] A. Sernadas and H.-D. Ehrich. What Is an Object, After All? In R. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK))*, pages 39–70, Amsterdam, 1991. North-Holland.

[SSG+91] A. Sernadas, C. Sernadas, P. Gouveia, P. Resende, and J. Gouveia. OBLOG — Object-Oriented Logic: An Informal Introduction. Technical report, INESC, Lisbon, 1991.

[Wie91] R. Wieringa. Equational Specification of Dynamic Objects. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proc. IFIP WG 2.6 Working Conference, Windermere (UK) 1990*, pages 415–438. North-Holland, 1991.

[Wir90] M. Wirsing. Algebraic Specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 677–788. Elsevier, North-Holland, 1990.