Towards a Module Concept for Object Oriented Specification Languages

Silke Eckstein

Technische Universität Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, D–38023 Braunschweig, Germany
e–mail: S.Eckstein@tu-bs.de

Abstract

The general aim of our work is the specification of distributed information systems, that means reactive systems consisting of one or more databases and application programs. In this context the formal object oriented specification language TROLL was developed and is now going to be extended by module concepts. In this paper we give a short introduction in TROLL and a survey over related work done not only in the object oriented setting but also in the fields of specification languages in general, theory of abstract data types and parameterized programming. When presenting our own module concepts we explicitly distinguish between providing the possibility to structure a large system on one hand and supporting the reuse of (parts of) specifications on the other hand.

1 Introduction

Investigations concerning module concepts, modular languages and modularization of software systems have been done since in 1972 Parnas introduced the notion of modules [19, 20]. Today it is widely accepted that a modular design of software decreases its complexity, i. e. makes it easier to construct, validate, maintain, and reuse software. In the past ten years a special kind of module has been developed: the concept of object classes. Nowadays it turns out that additional concepts are needed in the object oriented setting to support reuse of specifications as well as implementations and structuring of large systems.

Some of the popular methods and languages for object oriented design, like for example OMT [22] and UML [2], already provide module concepts, which allow to structure a system into manageable parts. Unfortunately, these modules (also called packages, subsystems etc.) are presented rather informally, i.e. they have neither explicit interfaces nor can they be parameterized and module composition operations do also not exist. [25] and [23] argue that object classes are too fine-grained to structure a system and to be effectively reused. Furthermore, [25] remarks that there may be a need to express invariants associated with more than one class. More emphasis on reuse is layed in [8], who distinguishes between black box, glass box und white box reuse. Another approach is taken in publications about design patterns [21, 9] which may also be regarded as special kinds of modules.

As the main interest of our work is the specification of distributed information systems, that means reactive systems consisting of one or more databases and application programs, the special requirements of both areas, databases and programming languages, have to be considered and integrated. In this context the formal object oriented specification language TROLL [12, 3] was developed and is now going to be extended by a module concept.

The paper is organized as follows: In the next section we give a brief presentation of TROLL. Related work is discussed in section 3 and in section 4 our own module concepts are introduced. After all we give an outlook on our further work in the 5. section.

2 TROLL

Information systems are distributed reactive systems consisting of one or more databases and application programs. The object oriented language TROLL [12, 3] was developed for specifying information systems at a high level of abstraction. The current version 3 is based on earlier investigations as described in [13, 15]. The textual language TROLL was supplemented by the graphical variant OMTROLL [16] which borrows concepts from OMT [22]. In the following we will give a brief survey of the main concepts of the TROLL–language. A more detailed description can be found in [12, 3].

A software system is considered to be a concurrent community of interacting objects. Objects are units of structure and behaviour and are classified in classes. Structuring mechanisms are aggregation, resulting in compound objects, and specialization with inheritance. Objects defined separately may be connected through global interactions, which are specified in the behavior part of an object system. We will illustrate this with an example.

Example 1 (airport information system): In the following, we use a small part of an airport information system to illustrate some of the language constructs. In figure 1 we define two object classes, `Passenger` and `Airline_Company`. Objects of class `Passenger` have got two attributes: a name, which is constant troughout the object live, and an address of data types `string` and `adr`, respectively. One action, `book_flight`, is specified. It has an input parameter `nr`, which stands for the number of the flight to be booked. For objects of class `Airline_Company` we specified an attribute to store the office address and an action `book`, that is used to book a seat in a flight offered by this company.

Two objects, `pass` and `comp`, exist in the airport system. `pass` is an object of class `Passenger` and `comp` is an object of class `Airline_company`. The interaction between these objects is specified in the behavior part of the object system whenever the passenger `pass` books a flight, the action `book` of airline company `comp` is called.

Though it is not specified in the example, one can imagine, that the object class `Airline_company` may have a number of flights as components. Then the action `book` would reserve a place in the flight with the given number. □

The formal semantics of TROLL is defined in different ways. Linear-time temporal

```
object class Passenger                    object class Airline_Company
  attributes                                attributes
    name: ;                                   office_addr: addr;
    address: addr;                          actions
  actions                                     book(nr: nat);
    book_flight(nr: nat);                   behavior
  behavior                                    ...
    ...                                   end;
end;

                object system Airport
                  objects pass: Passenger;
                  objects comp: Airline_Company;
                  behavior
                    pass.book_flight(nr)
                      do comp.book(nr) od;
                end_subsystem;
```

Figure 1: Airport_System

logic is used for describing sequential object behaviour. For specifying interactions between concurrent objects a distributed temporal logic is used, which is based on n–agent logic. An interpretation through a denotational behaviour model is given by labelled event structures. The theoretical foundations are outlined in [4, 5, 6].

3 Related Work

Module concepts and similar approaches have been investigated in various fields such as programming and specification languages, entity-relationship approach, theory of abstract data types and parameterized programming.

In the area of algebraic theory of abstract data types Ehrig and Mahr [7] developed a module concept, where a module specification consists of four components, which are given by algebraic specifications and combined through specification morphisms. In the import interface, resources are specified which are to be provided by other modules while the export interface contains the resources provided by this module. As a common part of import and export the parameter describes the generic resources to be instantiated in a concrete environment. The construction of the exported resources from the imported is done in the body. Various forms of interconnections are provided, for example composition, actualisation, union, realization and refinement. They are done by specification morphisms. A similar approach can be found in [18]. Though this is an elaborate theory which provides some useful concepts, not all aspects of our approach are covered. For instance, we have to deal with concurrent objects communicating with each other through action calling.

In [11] a set theoretic approach to module composition is given, using the notions of tuple sets, partial signature and institution. Two kinds of specification modules are distinguished, theories for describing properties of other modules and packages for specifying what has to be implemented. The operations on modules are renaming, hiding, enriching, composition of horizontal and vertical modules, parameterization and instantiation. Views are used to bind specification modules to theories. Similar to the algebraic approach described above, this one also provides some useful concepts which will help us, but it doesn't cover all aspects we are concerned in.

There already exist some object oriented specification languages which provide module concepts. FOOPS [24, 10] supports horizontal and vertical composition, abstract classes, parameterized modules, instantiation, import etc., but distribution, concurrency and communication are not considered. Similar concepts are offered by OOZE [1] which is syntactically based on Z.

A quite different approach was taken in the area of protocol specification for distributed systems. In this context the specification language Estelle [14] was developed, which is based on extended finite state automata. In Estelle, modules represent the (concurrent) components of a system. They may contain submodules that may be connected sequentially or concurrently. Modules communicate with each other through channels using an asynchronous communication mechanism. Though object orientation is not considered, the concurrency and communication issues are of interest for our work.

4 Module Concepts

By examining the presented approaches and requirements on module concepts more closely, one notices that they may be divided into two almost independent domains. One aspect is the necessity to structure large systems to make them easier to extend and better to understand. The other aspect is the reusability of already existing components and the construction of libraries for those. To explicitly distinguish between both aspects, we call the structuring units `subsystems` instead of modules and discuss them in the following subsection, whereas units of reuse retain the name `module` and are presented in subsection 4.2

4.1 Structuring large systems

To lower the effort of changes it is a good idea to divide a system into subsystems which contain closely related features, are as independent of each other as possible, have insuperable boundaries and precisely defined interfaces. To keep the independence as large as possible, the subsystems should be connected to each other only through communication as it is done in case of concurrent objects (cf. section 2). That means no relationships like "component_of", "object_valued_attribute_of", or "aspect_of" should be allowed.

In our approach, the interfaces of subsystems are built using objects called OFFER and REQUEST, which offer actions to other subsystems or need to get some information from other subsystems, respectively. Each action of an REQUEST object has to be connected to an action of an OFFER object from another subsystem. This is explicitly done in a communication section of an object system specification. Inside a subsystem these two special objects participate in communication relationships with the internal objects and establish in this way the connection between the interface and the module body.

Hence, if one looks upon the whole system, he or she can see a number of subsystems connected through communication relationships between their OFFER and REQUEST objects. If the point of view is changed to the internals of a subsystem, one can see data type and object class definitions as well as a number of communicating objects among them the objects OFFER and REQUEST.

Every subsystem may itself contain other subsystems, which are communicating through their OFFER and REQUEST objects. We will illustrate our language constructs with an example.

Example 2 (airport information system): In the following, we continue the example from section 2 by focussing on two subsystems, one for managing data about flights, i.e. connections, flight_schedules, delays etc., and another one for managing maintenance of planes, i.e. date of last maintenance, complaints, authorized supervisors etc.

```
subsystem FlightData
  object OFFER
    actions
      flight_schedule(date: date, !schedule: list(record(...)));
      destination (flight#: nat, !destination: );
      ...
  end;
  object REQUEST
    actions
      is_maintained (plane#: nat, !info: record(...));
  end;
  data type ...
  object class ...
  objects ...
  behavior ...
end_subsystem;
```

Figure 2: Subsystem FlightData

The subsystem `FlightData` (see Fig. 2) offers information about flight schedules for given dates and about destinations for given flight numbers. It needs the information wether a given plane is maintained, that means wether it is allowed to fly. The answer to this question is offered by the subsystem `PlaneMaintanance` (Fig. 3). The connection of the two subsystems is done in the communication part of the object system (Fig. 4). □

If a new subsystem should be added to an existing system, it has to be declared and all actions from its REQUEST–object have to be connected to actions from OFFER–objects from the already existing subsystems.

4.2 Reusing specifications

Initialy, one idea of the object oriented approach was to increase reusability with help of concepts like inheritance. But it turned out that object classes are too small

```
subsystem PlaneMaintanance
  object OFFER
    actions
      is_maintained (plane#: nat, !info: record(...));
  end;
  ...
end_subsystem;
```

Figure 3: Subsystem PlaneMaintanance

```
object system Airport
  subsystems
    FlightData
    PlaneMaintanance
  communication
    FlightData.REQUEST.is_maintained
      do PlaneMaintanance.OFFER.is_maintained od;
  ...
end
```

Figure 4: Object System Airport

to support effective reuse and that it would be advantageous to reuse a set of classes together with their relationships. These structures should be available in libraries from where they may be imported into other specifications. In contrast to conventional languages it is not enough to make access to routines or data types possible, but the object structures should be integrated into existing ones, e.g. it should be possible to build "component_of" and "aspect_of" relationships. Furthermore, these reusable modules have to be made suitable for the current context. Here, module operations as presented in [7, 11, 10] are of interest for our work. We will illustrate these ideas and our language constructs with some small examples which may be a little contrived but sufficient to explain our approach.

Example 3 (airline company): Suppose the airline company from example 1 should besides other things have its planes as components and should also have its employees as such. Let in library `A` be a module `Person/Staff` which consist of an object class `Person` and an specialized object class `StaffMember`. As shown in Fig. 5, the module `Person/Staff` should be imported and become a component of `AirlineCompany`. The corresponding TROLL statements are shown in Fig. 6. □

In general, imported modules are included in the current specification and their object classes may be used to build complex object classes or nodes. Though, in our opinion, whole subsystems as described in section 4.1 will often be too large and hence too special to be reused, this should be possible in principle. To do so, the module containing the subsystem has to be imported as shown in the previous example and the REQUEST actions of the subsystem have to be connected to OFFER actions of other subsystems as described in section 4.1.

As already mentioned, there should be some operations to adapt the library modules to the current context. The provided operations are renaming, adding and hiding.

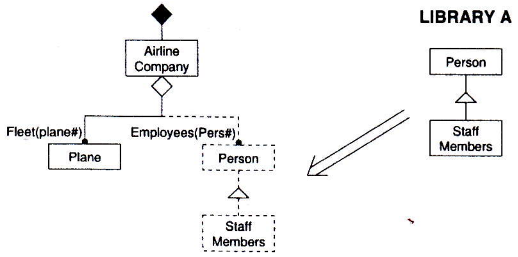

Figure 5: reuse of module Person/Staff

```
object class AirlineCompany
  components
    Fleet (planeNr: int): Plane
    Employees (persNr: int): Person
  attributes
    ...
end;
Import Person/Staff from A
```

Figure 6: TROLL specification of reuse

Example 4 (airline company): Let the object class `Person` of the imported module `Person/Staff` in example 2 have attributes `name`, `address` and `telNr`. Suppose that instead of `name` the attributes `firstName` and `surname` are needed and that the telephonenumber shoud not be visible anymore. Then the import statement in Fig 5 would have to be replaced by

```
Import Person/Staff from A
  as (add attribute firstName to Person,
      rename name to surname in Person,
      hide telNr in Person).
```

□

To increase the reusability of modules, it should be possible to parameterize them. The most common example to explain parameterization is a stack with elements from an unknown type, which later has to be instantiated. The approaches [10, 11] go one step further by allowing even modules to be used as parameters. Doing so, something like a type for modules is needed. We are planning to provide similar concepts, and in the area of semantic foundations some results already exist [17].

5 Concluding Remarks

In this paper we distinguished between two kinds of module concepts. On the one hand we provided subsystems for the structuring of large specifications, which are connected through communication relationships between special objects named

OFFER and REQUEST. And on the other hand we discussed concepts for the reuse of specification operations allowing to adapt them to the current context.

The next steps of our work will be to introduce asynchronous communication between subsystems, to provide parameterization as explained above and to formalize our language constructs.

References

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

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language. User Guide.* Addison–Wesley, 1997.

[3] 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.

[4] H.-D. Ehrich. Object Specification. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *IFIP WG14.3 Book on Algebraic Foundations of Systems Specification.* Springer, 1996. *To appear.*

[5] 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.

[6] H.-D. Ehrich and A. Sernadas. Local Specification of Distributed Families of Sequential Objects. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data 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 219–235. Springer, Berlin, LNCS 906, 1995.

[7] Ehrig and Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints.* Springer, 1990.

[8] Eisenecker. Objektorientierte Software wiederverwendbar entwerfen. In *Softwaretechnik '96*, 1996.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-oriented Software.* Addison-Wesley, Reading, 1996.

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

[11] J.A. Goguen and W. Tracz. An Implementation Oriented Semantics for Module Composition. 1997.

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

[13] 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.

[14] Hogrefe. *Estelle, LOTOS und SDL. Standard-Spezifikationssprachen für verteilte Systeme.* Springer, 1989.

[15] 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.

[16] 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.

[17] J. Küster Filipe. Modelling Parameterisation in Concurrent Object Systems. *IGPL*, 1997. In Conference Report: Workshop on Logic, Language, Information and Computation (WoL LIC)'97, Fortaleza, Ceará August 20-22.*To appear.*

[18] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of abstract data types.* J. Wiley & Sons and B.G.Teubner Publishers, 1996.

[19] D.L. Parnas. A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15:1053–1058, 1972.

[20] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15:330–336, 1972.

[21] W. Pree. *Design Patterns for Object/Oriented Software Development.* Addison-Wesley, ACM Press, Wokingham, 1995.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, New York, 1991.

[23] A. Rüping. Modules in object–oriented systems. In R. Ege, M. Singh, and B. Meyer, editors, *TOOLS 1 — Technology of Object–Oriented Languages and Systems.* Prentice Hall, 1994.

[24] A.J. Socorro Ramos. *Design, Implementation and Evaluation of a Declarative Object–Oriented Programming Language.* PhD thesis, Oxford University Computing Laboratory, Programming Research Group, Oxford, 1993.

[25] C.A. Szyperski. Import is Not Inheritance. Why We Need Both: Modules and Classes. In *ECOOP*, pages 19 - 32, 1992.