Increasing Reusability in Information Systems Development by Applying Generic Methods

Silke Eckstein, Peter Ahlbrecht, and Karl Neumann

Information Systems Group,
TU Braunschweig
P.O.box 3329, D-38023 Braunschweig
{s.eckstein, p.ahlbrecht,k.neumann}@tu-bs.de

Abstract. Increasing the reuse of parts of the specification and implementation of complex software systems, as for example information systems, may lead to substantial progress in the development process. This paper focuses on reusing parts of specifications with the help of generic methods and explores two aspects: the parameterization concepts of the languages UML and Troll, and how formal parameters in such concepts can be restricted if needed.

1 Introduction

The development of large information systems is by far no trivial task, since the probability of errors grows significantly with increasing complexity. However, approaches are being made to realize large systems on the basis of generic methods (cf. e.g. [5]) and thereby to reduce the complexity of the development process. The term “generic methods” also includes parameterized programming and related approaches at the specification level [9].

Information systems are mostly rather complex programs, which are frequently custom tailored for special applications. Their central part is usually a database system, and additional functionality, especially the user interface, is realized as an extensive application program. In recent years database technology moved from centralized systems to distributed databases and client/server systems [3,40], and the swift acceptance of the World Wide Web led to the situation that nowadays users are expecting to be able to use their familiar web browser as the interface to various information systems [21,2].

In this paper we focus on the specification of such information systems and particularly on parameterization aspects. In general, we propose to apply UML [29,43] together with the formal specification language Troll [27,25] developed in our group, in order to utilize the advantages of both a semi–formal graphical language and a formal one, as discussed for instance in [48] for an older version of Troll and a predecessor of the UML, the Object Modeling Technique (OMT) [42]. Regarding parameterization, both languages do provide such concepts, and in this paper we study how they correspond to each other.

To this end we first give an overview of current research activities related to generic methods, putting special emphasis on parameterization concepts for

K.R. Dittrich, A. Geppert, M.C. Norrie (Eds.): CAiSE 2001, LNCS 2068, pp. 251–266, 2001.
© Springer-Verlag Berlin Heidelberg 2001

describing variants of a system. We then introduce the formal object oriented specification language TROLL before sketching our area of application. In Sect. 5 we introduce and investigate the parameterization concepts of TROLL und UML and focus on one hand on the parameter granularity and on the other hand on parameter constraints. Finally we summarize our paper.

2 Generating Software

As a long-term objective the possibility to describe families of information systems with the support of specification libraries is as desirable as generating concrete runnable systems from such a description. However, while the concept of generators has been in use in some fields of software engineering, for instance as scanner and parser generators in the area of compiler construction or the generation of user interfaces [44], to name a very recent approach, there is so far no uniform theory and methodology on generating information systems. Nevertheless the automatic implementation of such reactive systems is considered possible, not only based on specifications given at a high level of abstraction, but even starting from the results of the requirements analysis [26].

[5] discusses various generators for highly specialized software systems, for example Genesis as a generator for database management systems, Ficus for generating distributed file systems, ADAGE as a generator for avionic software, and SPS for software on signal processing. According to [4], all these generators belong to the so-called GenVoca approach, in which the generator constructs a software system with respect to the following points: a system consists of modules, which can be assigned to different fields of application, and which form distinctively larger units than given by classes or functions. Modules are implemented by components, and different components for the same task provide the same interface. The combination of modules and their adaptation to meet special requirements is achieved by parameterization. In addition to these points the possibility to check configurations for validity should also be given [30].

The different types of generators can be classified according to a variety of criteria. We may, for example, distinguish between whether they are compositional or transformational. While the former compose the desired software out of prefabricated components, the latter actually generate the code themselves. Another classification criterion is the question of whether the software is generated statically or dynamically, i.e. of whether a generated complete system may be reconfigured at runtime or not. Regarding parametric languages it is also possible to differentiate between generators that switch from the source language to a different target language, and those which maintain the same language for input and output. Generators of the latter type will replace formal parameters by provided elements during instantiation, thus producing non-parametric code of the same language. Domain-specific generators exist which are capable of creating systems of a special type — examples are scanner and parser generators — and there are also generators for “arbitrary” systems like, for instance, compilers for programming languages.

Under the term *generic methods* we should subsume not only generators and their application for creating software, but also languages which permit a description of families of specification and implementation modules by means of generic elements — so–called formal parameters — in such a way that these descriptions may then be adjusted according to given special requirements by binding the formal parameters to actual values.

In the theory of abstract datatypes the idea of parameterization is a well known concept which has already been investigated some time ago. Here it suffices to mention the frequently used example of a stack for storing elements of an unspecified type. C++, for instance, supports the implementation of such a parameterized abstract datatype with its template construct. In Java, however, a corresponding language element is currently still not available, but attempts to add such a parameterization concept are being made (cf. e.g. [38,7]). Compared to C++ these approaches also provide the possibility to define (syntactical) properties of the parameters. In the literature this possibility is also termed *bounded parametric polymorphism* [1] or *constrained genericity* [36].

Parameterization concepts have also been investigated and formalized in the field of algebraic specification of abstract datatypes (cf. e.g. [19,35]). Here, it is also possible to define semantic properties by providing axioms which the actual parameters have to meet, in addition to being able to describe the signature of the parameters. [20] transferred these results on to larger units, namely modules, which may themselves have other modules as parameters. Quite a few specification languages have been developed in this area, which we will not discuss in further detail, but refer this to [49], which provides a useful overview.

We only mention OBJ [23], as this specification language had a major impact on the development of LILEANNA [22], even though the latter also incorporates implementation aspects in addition to specification concerns and the formal semantics of the languages differ. LILEANNA, too, facilitates having entire modules as parameters and has been used during the already mentioned ADAGE project for the generation of avionic software. Furthermore, the language distinguishes between a vertical and horizontal composition and, accordingly, also between horizontal and vertical parameters. Vertical composition permits the description of a system in separate layers, while horizontal composition supports structuring of the single layers [24].

In all the approaches which we have discussed so far the structuring elements like classes or abstract datatypes may be generic. The structure which they describe, however, is and remains unchangeable. In contrast to this the *collaboration–based design* [46] assumes that the collaborations among different classes, i.e. class–spanning algorithms, represent the reusable units, which should be adaptable to various concrete class structures. In [37] these collaborations are described relatively to an abstract class graph, which again represents the interface to a concrete class graph. This setup is termed *structure generic components.*

Irrespective of the type of components, classes or modules, they have to be stored in libraries to be available for reuse. Here, powerful tool support is needed (cf. e.g. [10,31,11]) to make effective reuse possible.

3 The TROLL–Approach

The object oriented language TROLL [27,25] was developed for specifying information systems at a high level of abstraction. It has been successfully utilized in an industrial environment to develop an information system in the area of computer aided testing and certifying of electrical devices [32,28].

In this approach informations systems are regarded as being communities of concurrent, interacting (complex) objects. Objects are units of structure and behavior. They have their own state spaces and life cycles, which are sequences of events. A TROLL specification is called *object system* and consists of a set of datatype and a set of object class specifications, a number of object declarations and of global interaction relationships.

Object classes describe the potential objects of the system by means of structure, interface, and behavior specifications. Attributes are used to model the state spaces of the objects. Together with the actions they form the objects' local signatures. Attributes have datatypes and may be declared as hidden, i.e. only locally visible, optional or constant. They can be derived, meaning their values are calculated from the values of other attributes, or initialized.

The second part of the signature determines the actions of the objects. Each action has a unique name and an optional parameter list. Input and output values are distinguished and each parameter has a certain datatype. The visibility of actions can be restricted in such a way that they do not belong to the object class' interface but can be used internally only. Actions that create objects of a class, so called birth actions, and actions that destroy objects, so called death actions, are explicitly marked.

While the signature part of an object class determines the interface of the objects, the behavior part constitutes the reactions of the objects to calls of the actions declared in the interface. The admissible behavior of the objects can be restricted by means of initialization constraints and invariants.

Complex objects may be built using aggregation and with the consequence that the component objects can exists only in the context of the complex one and that the superior object can restrict the behavior of its components. By means of specialization hierarchies base classes may be extended with further aspects.

The set of potential instances is determined by object declarations at the object system level. At runtime concrete instances can be created through the calling of birth actions of the respective object classes. All instances of a system are concurrent to each other and synchronize when they interact. Interactions are global behavior rules that together with the local ones describe the behavior of the system.

Semantics is given to TROLL specifications using different techniques: the static structure of an object system is semantically described with algebraic methods, and to describe properties of distributed objects a special kind of temporal logic has been developed. This logic is called *Distributed Temporal Logic* (DTL) [17,16] and is based on n-agent logics. 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 [39]. 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. An exhaustive description of the model theory is given in [18,15].

Presently, an extension of TROLL with module concepts is under investigation, aiming on one hand at providing more sophisticated structuring concepts and on the other hand at supporting reuse of specifications by means of concepts for parameterization [12,13]. Regarding theoretical foundations, module theory is being addressed e.g. in [33,34], where in particular DTL has been extended to MDTL *(Module Distributed Temporal Logic).*

4 Area of Application

The starting point for our investigations on generic information systems are web-based information systems which can be generated in order to facilitate the administration of tutorials. In a tutorial, students are supervised in small groups which are guided by a tutor. The students have to complete exercises handed out by one of the lecturer's assistants in teams of two. If they achieve a certain percentage of the total of points assigned to the exercises, they obtain a certificate for the course at the end of term.

Looking from an organizational point of view this means that the students have to form teams of two, and register for the tutorial providing certain personal information (name, registration number, etc.). During term time the tutors keep account of the points which the teams and students of their group receive, so that by the end of term the certificates can be printed and signed by the lecturer.

As a first basic structuring of the administration system three layers can be identified (cf. also [41]): one for presenting information, one for storing it, and the third to facilitate the exchange of data between the former two. These tasks are taken over by the packages *Presentation* and *Storage* and the class *Controller*, respectively, which are components of the package *TutorialAdministrationSystem*, which again represents the entire system. In the following a more detailed discussion will be given only for the package modeling the user interface. Figure 1 shows the corresponding class diagram for the package *Presentation.*

In this figure, the possibilities to access the single web pages are modeled by compositions. The parts cannot be created before and die at the latest with the death of the composite object. A *StandardPage*, for example, can be viewed only after a successful log-on on a *StartPage*. The specialization of the *StandardPage* into the *Assistant–*, *Tutor–* and *StudentPage* states that access to the *Tutorial–*

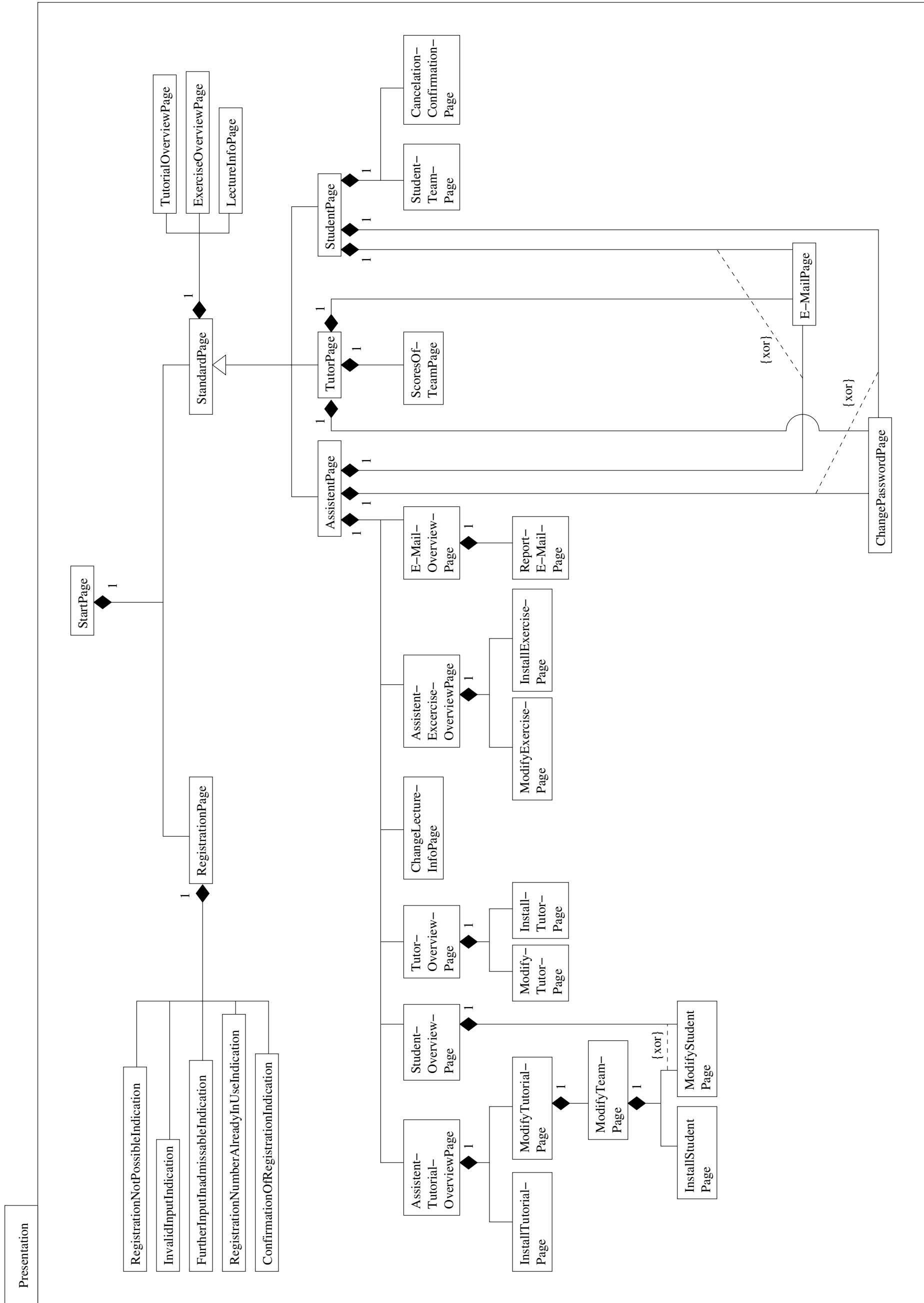

Fig. 1. Detailed class diagram of package *Presentation*

and *ExerciseOverviewPage* as well as to the *LectureInfoPage* is also possible from the specialized pages. In contrast to this, changing the password or sending e-mail can only be done from the specialized pages.

Access to the system is possible via a *StartPage* only. There, a user can choose to register for the tutorials or retrieve general information on them. Someone already registered with the system may, in accordance with his user status, retrieve further information or perform additional operations. Students for example may change their address and retrieve the score of their team. Identification is accomplished by prompting for a login and password at the *StartPage*. An unregistered student may enroll himself and others using the *RegistrationPage* as many times as admissible to enter the data of the team members. A detailed description of the registration procedure using activity diagrams can be found in [14]; due to space restrictions we refrain from elaborating on it here.

On the one hand, the static part of the *Presentation* package thus describes the information which the system has to provide for the different groups of users. On the other hand it also models the possibilities for navigating between the user interfaces, as e.g. [45] recommends for the design of web sites.

The requirements which have been outlined here so far apply to many tutorials with different deviations. For example, the size of the teams and the required minimum score may vary. The tutorials may be held several times per week or less, and even completely irregularly. Furthermore, they do not have to take place at the same time, which should be reflected in the application procedure: With different contact hours of the tutors it would be nice to offer the students the possibility to choose a time which suits their timetable, requiring a completely different algorithm from the one which merely distributes them equally over the available tutorials in the case of simultaneous contact hours. Finally, the number of teams per group may be restricted, for instance for practicals which use equipment that is only available in a limited number.

5 Specifying Generic Information Systems

The application area introduced in Sect. 4 has been a starting point for a case study addressing certain aspects from the subject “generating information systems”. We studied parameterization concepts at the specification level or, more precisely, parameterization concepts of the UML, which can be used to describe variants of a system, and implemented a generator program, which produces runnable systems from prefabricated components.

In this paper we concentrate on the specification aspects. In particular, we present the parameterization concepts of Troll, compare them to those of the UML and discuss some results of our case study.

The UML not only allows to parameterize classes, as e.g. C++ does, but also arbitrary model elements. Such parameterized classes, collaborations, packages etc. are called templates or template classes, template packages and so forth. Troll provides exactly one parameterizable model element, the so–called module, which can, however, consist of one or more classes together with their struc-

tural and communication relationships, or even entire subsystems. The latter are comparable to UML–packages.

There are different types the formal parameters of a UML–template can belong to: If actual arguments of a parameter are supposed to be values of a certain datatype, the parameter is specified in the form *name: type*. If, on the other hand, the actual argument is supposed to be a class or a datatype itself, it suffices to state the formal parameter's name. In this case we also talk about datatype parameters. Furthermore parameters may even represent operations. To represent a template, a small dashed rectangle containing the formal parameter is superimposed on the upper–right corner of the respective model element. In TROLL formal parameters always have to be declared together with a type. Valid types for formal parameters are datatypes or classes themselves, the statement "type", if the actual value is supposed to be a datatype or a class, or the statement "module", if the actual value is supposed to be a bunch of classes.

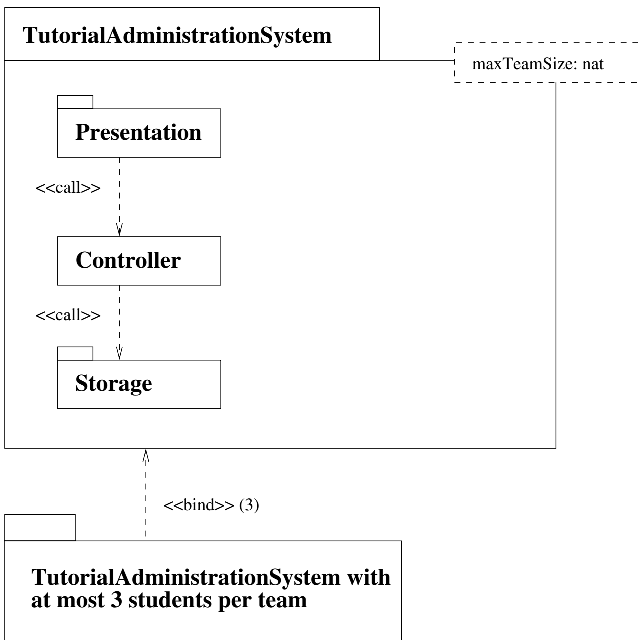

Fig. 2. Parameterized *TutorialAdministrationSystem*

Let us start examining these concepts by means of a small example from our application area. We stated in Sect. 4 that the students should complete exercises in teams of two. It may be desirable to leave the exact team size open at first and generalize the *tutorial administration system* by specifying the maximal number of students per team (*maxTeamSize*) as a parameter. In order to obtain a concrete or actual model element the parameters have to be instantiated.

Here, the formal parameter is bound to the value “3”. The corresponding UML specification is shown in Fig. 2 and in TROLL this looks like the following:

```
module TutorialAdministration
  parameterized by maxTeamSize: nat;
  subsystem Presentation ... end_subsystem:
  object class Controller ... end;
  subsystem Storage ... end_subsystem:
end-module;
```

Instantiation is expressed as follows:

```
instantiate module TutorialAdministration
  as TutorialAdministrationSystem_with_at_most_3_students_per_team;
  bind maxTeamSize to 3;
```

In the course of the case study it turned out to be beneficial to use lager units as parameters than those provided by the UML. The reason for this is that the number of parameters may increase rapidly and consequently the specification becomes unintelligible. Using UML as the specification language, we chose to employ the concept of packages and allow them to be used as parameters. As mentioned earlier, in TROLL modules are allowed to be used as parameters for other modules and hence parameters can be as large and as complex as needed.

In [14] these results are illustrated by means of a somewhat more complex part of our application area. There, two different procedures to registrate for the tutorials are discussed. For the first one it is assumed that all tutorials take place at the same time. Consequently, the students can be distributed on the tutorials automatically, whereby a level partition is guaranteed. In the second procedure the students are allowed to choose their groups on their own. In [14] three variants are discussed of how to parameterize the specification in such a way that the different registration procedures are taken into account. In the following we give an overview of these three variants, which differ in the granularity of the employed parameter types. Due to space limitations we refrain from presenting the respective translation to TROLL.

Allowing packages as types for parameters we can model the sketched scenario as follows (Fig. 3): The *registrationPage* and all its depending classes are comprised in a package called *registration*, which represents a formal parameter. This parameter can be instantiated with packages that contain the classes needed for the respective variants of registration. The controller is modelled as a formal parameter of type class. The expected actual arguments are classes, which offer operations needed for the interactions between the presentation and the storage component and which differ in the registration procedure. By instantiating the parameters *registration* and *controller* with suitable packages and classes unparameterized specifications result, which describe systems realizing the respective variant of registration.

A variant to model the requested scenario without using packages would look like Fig. 3 with the *Registration: package* being removed. Essentially, actual

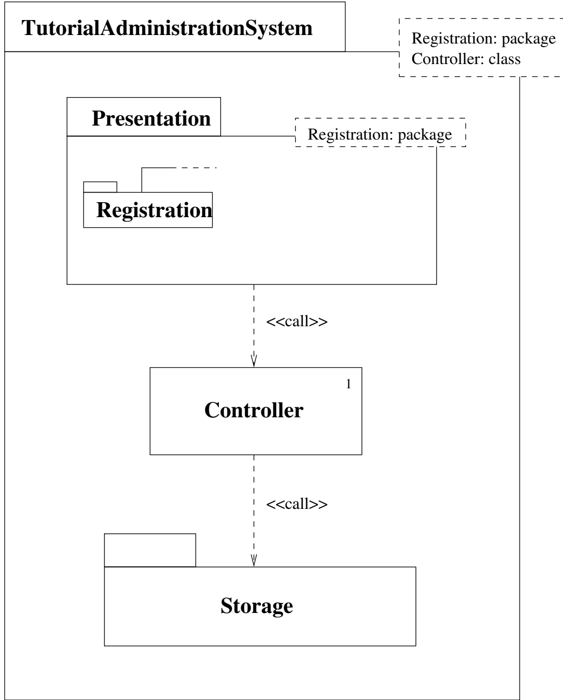

Fig. 3. Variant 1: Packages and classes as parameters

arguments for this parameter are equal to the ones sketched above, but the specification of the presentation package becomes somewhat more complex. As in this variant no packages are allowed to be used as parameters, it is not possible to group all required classes into one unit and exchange them together. Accordingly, this part of the specification has to be so general that it can deal with all actual controller classes which the formal parameter may be instantiated with and thus becomes more complex and a number of additional integrity constraints arises.

Figure 4 shows the last variant we are going to present here. Only data values and operations are used as actual arguments and thus the parameterization exhibits a much finer granularity than before. For instantiating the template two numerical values and an operation have to be provided. The numerical values determine the multiplicity values *maxTeamSize* and *maxNumberOfTeamsPerTutorial*, while the operation determines the core algorithm of the registration procedure.

For this variant, too, the above remarks with respect to the complexity of the presentation package hold, as only the specification of the controller has changed. Comparing the three parameterization variants one can see that in the third one the list of formal parameters may quickly become long and unintelligible while at the same time one can easier determine which parts of the specification are actually variable.

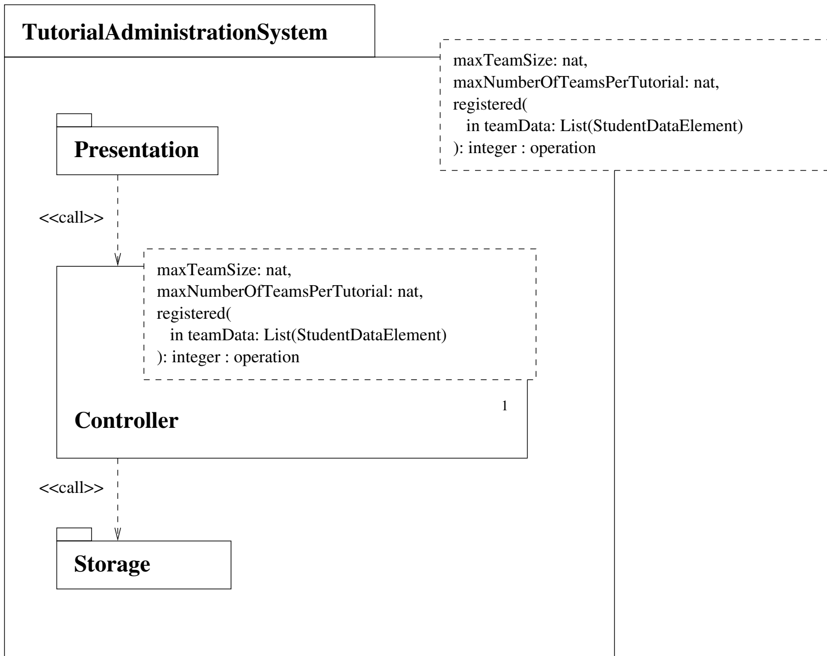

Fig. 4. Variant 3: Values and operations as parameters

It is true that variants one and two are easier to understand, but they require more redundancy between the different actual arguments a certain formal parameter may be bound to. For example considerable parts of the controller specification would be identical for each actual argument, resulting in problems with the maintenance of the specification. Presumably best results will be reached with a gradually applied parameterization of mixed granularity.

Independent from the (permissible) parameter granularity it is often necessary to restrict the set of potential actual arguments by means of additional rules. Such rules can for instance determine which combinations of actual arguments for different formal parameters can be used together, or which requirements the actual arguments have to fulfill in general in order to produce a correct non–parameterized specification. Depending on the type of parameter, different kinds of rules may be applied.

Five types of parameters can be distinguished. Table 1 gives an overview of the possibilities to restrict the respective types of parameters. The simplest case are value parameters as, for example, the maximum team size mentioned above. These parameters are roughly described by their datatype and can be made more concrete by a further restriction of their range.

All other types of parameters are settled at a higher level of abstraction, where datatypes make no sense. Instead, the expected signature can be specified. For classes, operations and packages it may be useful to fix the complete

Table 1. Possible parameter restrictions

	value parameter	datatype parameter		operation	package/
		basis datatype	class		module
type specification	×				
signature			×	×	×
signature part		×	×		×
range restriction	×			×	
pre– and postcond.				×	
choice			×	×	×

signature. In the case of operations this would be the operation name as well as the names and datatypes of the input and output parameters. In the case of classes this would be the operations the respective class is expected to provide, and in the case of packages it would be the classes with their signatures.

Besides specifying the complete signature, one may also want to fix only that part of it that is relevant in the respective template. For example, it could be sufficient to state that the required datatype has to provide a compare operation for its elements. For classes it can be useful to specify a base class where the class given as an argument has to be derived from. While it does not make much sense to specify an incomplete signature for an operation, it sometimes makes sense to restrict the range of its return values. One may for instance think about an operation to produce random numbers, where the interval of the output values shall be restricted to a certain range. Furthermore, operations can be characterized by the specification of pre– and postconditions. All restrictions that can be constituted for operations that are parameters in their own right can also be constituted for operations as parts of class or package parameters.

Other types of restrictions are needed in the case that the actual arguments a parameter can be bound to should not be any arguments which fulfill the restrictions, but only such arguments which are provided by a library. Here, rules regarding permisible combinations of actual arguments are of interest. Referring to our case study one may think of the situation where all possible registration algorithms and all variants of the controller class belonging to them are provided by a library. In the case that for instance algorithm x works together with controller classes a and b only, while algorithm y needs controller class c to cooperate with, this should be stated as a parameter rule.

The UML does not explicitly provide language constructs for the specification of parameter rules in templates. As also mentioned in [6], which develops a classification for stereotypes similar to the one given here for types of formal parameters, with the OCL (Object Constraint Language) being part of the UML [47], we have a formalism at hand with which such rules can in principle be stated. However, if we employ UML together with TROLL in order to utilize the advantages of both, we could shift the specification of parameter constraints to TROLL and use the power of a formal specification language for this purpose.

In our example from above such a parameter rule would be specified as follows:

```
module TutorialAdministration
  parameterized by maxTeamSize: nat;
  parameter constraint 1 < maxTeamSize < 6;
  subsystem Presentation ... end_subsystem:
  object class Controller ... end;
  subsystem Storage ... end_subsystem:
end-module;
```

Here, the range of `maxTeamSize` is restricted. In general all kinds of rules that have been sketched above can be specified.

6 Conclusions

Further demands on the already complex task of developing information systems motivate using generic methods, which facilitate and support reusing parts of a specification and implementation. Having outlined this in the first chapter of this paper, we gave an overview of current research activities related to generic methods putting special emphasis on parameterization concepts for describing variants of a system. Chapter 3 provided a brief introduction to the general concepts of the object oriented language Troll, to its building blocks for specifications as well as to its formal semantics. Following this, we sketched an application area for information systems at universities, namely the administration of tutorials, and showed that also in this area use of generic methods is desirable and useful.

With this as the foundation, in chapter 5 we outlined a specification for a tutorial administration systems, introducing and investigating on the parameterization concepts of Troll und UML. Our first main focus here was on the granularity which the formal parameters of parameterized elements of specifications should have, and it turned out that presumable best results with regard to intelligibility and redundancy will be achieved by a gradually applied parameterization of mixed granularity.

The other main focus of chapter 5 was to discuss how restrictions on formal parameters, which are frequently needed with generic methods, can be expressed using rules. We gave an overview of different types of formal parameters together with the possibilities to restrict them, and provided examples of where such restrictions might be desired. Even though the UML provides the sublanguage OCL, which could — after some extensions — be used to express such restrictions, we propose to use Troll instead and in general to use the UML only for the description of the overall architecture, shifting the specification of details to the textual language. To profit significantly from this joint application of a semi–formal graphical language with a formal textual one, it is essential to give a precise definition of their combination [8], which is one of the further steps we plan to do.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer–Verlag, New York, 1996.
2. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, 2000.
3. P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems: Concepts, Languages and Architectures*. McGraw–Hill, London, 1999.
4. D. Batory. Software Generators, Architectures, and Reuse. Tutorial, Department of Computer Science, University of Texas, 1996.
5. D. Batory. Intelligent Components and Software Generators. Invited presentation to the "Software Quality Institute Symposion on Software Reliability", Austin, 1997.
6. S. Berner, M. Glinz, and S. Joos. A Classification of Stereotypes for Object–Oriented Modeling Languages. In *Proc. 2nd Int. Conf. on the UML*, LNCS 1723, pages 249–264. Springer, Berlin, 1999.
7. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proc. Int. Conf. on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 183–200, 1998.
8. S. Brinkkemper, M. Saeki, and F. Harmsen. Assembly Techniques for Method Engineering. In B. Pernici and C. Thanos, editors, *Proc. 10th Int. Conf. on Advanced Information Systems Engineering (CAiSE'98), Pisa*, LNCS 1413, pages 381–400. Springer, Berlin, 1998.
9. K. Czarnecki and U.W. Eisenecker. *Generative Programming — Methods, Tools, and Applications*. Addison–Wesley, Boston, 2000.
10. E. Damiani, M.G. Fugini, and C. Belletini. A hierarchy–aware approach to faceted classification of object–oriented components. *ACM Transactions on Software Engineering and Methodology*, 8(3):215–262, 1999.
11. K.R. Dittrich, D. Tombros, and A. Geppert. Databases in Software Engineering: A RoadMap. In A. Finkelstein, editor, *The Future of Software Engineering (in conjunction with ICSE 2000)*, pages 291–302. ACM Press, 2000.
12. S. Eckstein. Modules for Object Oriented Specification Languages: A Bipartite Approach. In V. Thurner and A. Erni, editors, *Proc. 5th Doctoral Consortium on Advanced Information Systems Engineering (CAiSE'98), Pisa*. ETH Zürich, 1998.
13. S. Eckstein. Towards a Module Concept for Object Oriented Specification Languages. In J. Bārzdiņš, editor, *Proc. of the 3rd Int. Baltic Workshop on Data Bases and Information Systems, Riga*, volume 2, pages 180–188. Institute of Mathematics and Informatics, University of Latvia, Latvian Academic Library, Riga, 1998.
14. S. Eckstein, P. Ahlbrecht, and K. Neumann. From Parameterized Specifications to Generated Information Systems: an Application. (In German). Technical Report 00–05, Technical University Braunschweig, 2000.
15. H.-D. Ehrich. Object Specification. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, chapter 12, pages 435–465. Springer, Berlin, 1999.
16. H.-D. Ehrich and C. Caleiro. Specifying Communication in Distributed Information Systems. *Acta Informatica*, 36:591–616, 2000.
17. 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*, chapter 6, pages 167–198. Kluwer Academic Publishers, Dordrecht, 1998.

18. 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*, LNCS 906, pages 219–235. Springer, Berlin, 1995.
19. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer, Berlin, 1985.
20. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints.* Springer, Berlin, 1990.
21. A. Gal, S. Kerr, and J. Mylopoulos. Information Services for the Web: Building and Maintaining Domain Models. *Int. Journal of Cooperative Information Systems (IJCIS)*, 8(4):227–254, 1999.
22. J. Goguen. Parameterized Programming and Software Architecture. In IEEE Computer Society, editor, *Proceedings Fourth International Conference on Software Reuse*, pages 2–11, 1996.
23. J. Goguen and G. Malcolm, editors. *Software Engineering with OBJ: Algebraic Specification in Action.* Kluwer, Boston, 2000.
24. J. Goguen and W. Tracz. An Implementation Oriented Semantics for Module Composition. In G.T. Leavens and M. Sitaraman, editors, *Foundations of Component–Based Systems*, pages 231–263. Cambridge University Press, 2000.
25. A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger, and H.-D. Ehrich. The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools. In T.W. Ling, S. Ram, and M.L. Lee, editors, *Proc. 17th Int. Conf. on Conceptual Modeling (ER'98)*, pages 277–290. Springer, LNCS 1507, 1998.
26. D. Harel. From Play–In Scenarios to Code: An Achievable Dream. In T. Maibaum, editor, *Proc. 3rd Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2000)*, pages 22–34. Springer, LNCS 1783, 2000.
27. P. Hartel. *Conceptual Modelling of Information Systems as Distributed Object Systems. (In German).* Series DISDBIS. Infix–Verlag, Sankt Augustin, 1997.
28. 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.
29. M. Hitz and G. Kappel. *UML@Work.* dpunkt, Heidelberg, 1999.
30. S. Jarzabek and P. Knauber. Synergy between Component-Based and Generative Approaches. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering - ESEC/FSE'99*, pages 429–445. Springer, LNCS 1687, 1999.
31. M. Jeusfeld, M. Jarke, M. Staudt, C. Quix, and T. List. Application Experience with a Repository System for Information Systems Development. In R. Kaschke, editor, *Proc. EMISA (Methods for Developing Information Systems and their Applications)*, pages 147–174. Teubner, 1999.
32. M. Krone, M. Kowsari, P. Hartel, G. Denker, and H.-D. Ehrich. Developing an Information System Using TROLL: an Application Field Study. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Proc. 8th Int. Conf. on Advanced Information Systems Engineering (CAiSE'96)*, LNCS 1080, pages 136–159, Berlin, 1996. Springer.
33. J. Küster Filipe. *Foundations of a Module Concept for Distributed Object Systems.* PhD thesis, Technical University Braunschweig, 2000.
34. J. Küster Filipe. Fundamentals of a Module Logic for Distributed Object Systems. *Journal of Functional and Logic Programming*, 2000(3), March 2000.
35. J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* John Wiley & B. G. Teubner, New York, 1996.

36. B. Meyer. *Object–oriented Software Construction*. Prentice Hall, New York, 1988.
37. M. Mezini and K. Lieberherr. Adaptive Plug–and–Play Components for Evolutionary Software Development. In *Proc. of the 1998 ACM SIGPLAN Conference on Object–Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, volume 33 (10) of *SIGPLAN Notices*, pages 97–116, Vancouver, 1998.
38. A.C. Myers, J.A. Bank, and B. Liskov. Parameterized Types for Java. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages*, pages 132–145, Paris, 1997.
39. M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoretical Computer Science*, 13:85–108, 1981.
40. M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Upper Saddle Rive, 2. edition, 1999.
41. G. Preuner and M. Schrefl. A Three-Level Schema Architecture for the Conceptual Design of Web-Based Information Systems: From Web-Data Management to Integrated Web-Data and Web-Process Management. *World Wide Web Journal, Special Issue on World Wide Web Data Management*, 3(2), 2000.
42. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modeling and Design*. Prentice Hall, New York, 1991.
43. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Guide*. Addison–Wesley, 1999.
44. P. di Silva, T. Griffiths, and N. Paton. Generating User Interface Code in a Model Based User Interface Development Environment. In V. di Gesu, S. Levialdi, and L. Tarantino, editors, *Proc. Advanced Visual Interfaces (AVI 2000)*, pages 155–160. ACM Press, New York, 2000.
45. O. De Troyer. Designing Well–Structured Websites: Lessons to Be Learned from Database Schema Methodology. In T.W. Ling, S. Ram, and M.L. Lee, editors, *Proc. 17th Int. Conf. on Conceptual Modeling (ER'98), Singapore*, pages 51–64. Springer, LNCS 1507, 1998.
46. M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration Based Designs. In *Proc. of the 1996 ACM SIGPLAN Conference on Object–Oriented Programming Systems, Languages and Applications (OOPSLA '96)*, volume 28 (10) of *SIGPLAN Notices*, pages 359–369, San Jose, 1996.
47. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison–Wesley, Reading, 1999.
48. R. Wieringa, R. Jungclaus, P. Hartel, T. Hartmann, and G. Saake. OMTROLL – Object Modeling in TROLL. In U.W. Lipeck and G. Koschorreck, editors, *Proc. Intern. Workshop on Information Systems — Correctness and Reusability IS-CORE '93, Technical Report, University of Hannover No. 01/93*, pages 267–283, 1993.
49. M. Wirsing. Algebraic Specification Languages: An Overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, pages 81–115. Springer, LNCS 906, 1995.