

Component Generalization with Formal Methods

Samira Sadaoui

Dept of Math & Computer Science, 4401 University Drive
Lethbridge, T1K 3M4, AB, Canada
sadaoui@cs.uleth.ca

Abstract. Generalization is a powerful tool because it allows the building of libraries of reusable software components. In this paper we show the importance of formal methods for software development and reuse. The use of formal methods allows providing abstract and rigorous descriptions of software components. Reliability is increased when components are formally specified and are correct with respect to their specifications. We also present the characteristics of our assistance tool for the development and reuse of software components.

1 Introduction

Reuse is a major goal of modern software engineering because it is considered the key to improving the quality of software and productivity. To achieve these benefits, it is necessary to build software with components that are as reusable as possible. However, developing reusable components that are easy to identify, use and adapt, requires additional effort estimated from five to ten times the effort required to build non-reusable components [10]. This effort can be amortized only by the repeated use of the components. Most of the impact of reuse research has been via function libraries, black-box components, scripting languages and object oriented programming and frameworks. Reuse is most effective when applied at early phases of the life-cycle. Thus, we must learn to incorporate reuse into the analysis, specification and design phases whereas the majority of the previous work has focused on the reuse of code. In this paper, our interest is in formal specifications which offer characteristics that are useful for component-based software development; most important ones are not only the underlying theoretical concepts but more pragmatic issues such as a good level of abstraction, great description power, support for structuring, prototyping capabilities and the possibilities of visual aids.

The reuse process involves four steps: (1) find components, (2) select a component, (3) understand the component selected and if required (4) adapt the component. Enough techniques for finding and selecting components have received attention [8,17] but we do not see much focus on understanding and adapting components. Reuse will be facilitated when the development history of the reusable components can be tracked back [3]. This trace is important for

the comprehension and evolution of components. It is also important to anticipate reuse with methods for building reusable components and adapting existing components. These methods are non-existent in the literature.

We believe that one method supporting reuse is component generalization that can greatly enhance the reusability of existing components [16, 14]. The cost of developing generic components is amortized by the repeated use (by instantiation) of these components and by reducing the risk of introducing errors in the instantiated components. In [14], we have defined a generalization method (by parameterization) that consists of abstracting an auxiliary part of an algebraic specification to a more general parameter. Our generalization method creates reusable specification components from already developed ones in opposition to the standard approach that creates components explicitly for a future reuse.

The premise of our work is to support more effective reuse, guidelines and tools that are needed to assist the developers when applying the reuse methods. By assistance, we mean the set of activities allowing the planning, automation and control of a reuse process. We have improved an existing model called Proplane [7] in order to promote the reuse of formal specification components [15]. With Proplane, we use the notion of “workplan” to describe the different steps of the construction of a specification component and the notion of “operators” to define different reuse methods and to assist in their applications in specific contexts.

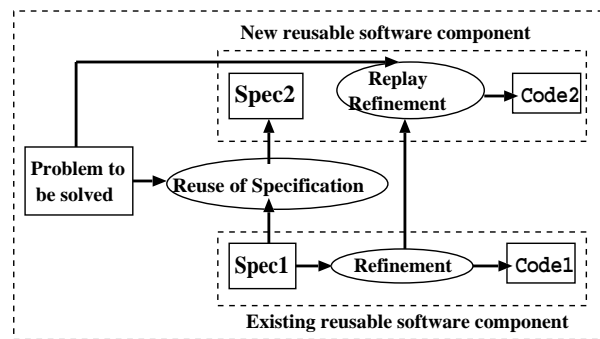
The paper is organized as follows: Section 2 shows the importance of formal methods for component-based software development. Section 3 presents briefly our generalization operation. In section 4, we describe our assistance tool for the specification reuse. Concluding remarks on and future directions of our work are presented in Section 4.

2 Component-based development process with formal methods

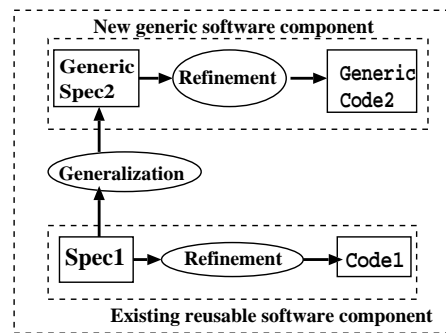
For the formal development of software components, we need techniques and tools for specification, refinement and code generation. We define a software component as a triple: $\langle \textit{specification}, \textit{refinement}, \textit{code} \rangle$. The formal methods offer characteristics that are useful for reuse, including:

- a good level of abstraction that allows the exploitation of the component’s properties rather than its details of implementation. The specification gives the meaning of the operations that are implemented in the code part. It is not practical to enter the code to get the meaning of the component [2];
- modularity that facilitates the modification of components defined as a hierarchy of modules; some of these modules can be replaced by new ones in the new context [5];

- simple and well defined semantics that support rigorous reasoning about components. Tools that manipulate code can easily deal with syntactic features of components (e.g, renaming operations) but usually have difficulties when it comes to semantics;
- the possibility of prototyping with executable specifications;
- refinement and implementation of the specifications in the library [13,11]. The last refinement is intended to allow the code to be automatically generated;
- mathematical foundations that prove the correctness of components in the different steps of the reuse process. They are also used to prove the correctness of the implementation of specifications and the links between the different levels of abstraction.



(a) Reuse of software components



(b) Construction of generic software components

Fig. 1. Software Reuse, formal specifications and component generalization

Our goal is to integrate formal methods into component-based software development process. The reuse of an existing code component is done via the reuse of its formal specification. To achieve our goal, we have to define methods that allow the:

Development with reuse. The goal is to maximize the reuse of existing reusable software components as shown in figure 1(a). The specification $Spec_2$ (that describes the problem to be solved) is built by reusing, modulo some transformations, an existing specification $Spec_1$ [15]. The corresponding code, $Code_2$, can be built by replaying, modulo some transformations, the refinement process of the reused specification $Spec_1$.

Development for reuse. The goal is to produce generic software components with the highest reuse potential as shown in figure 1(b). The current specification $Spec_2$ can be built by the generalization of an existing specification $Spec_1$ [14].

The generic specification $Spec_2$ is then refined in order to produce the generic code, $Code_2$. In the literature, we notice that there are many works associating the algebraic specifications with the object oriented concepts largely associated with reuse [6, 12]. We hope to use them in the future for the implementation of our specifications in the library.

In our research work, we are interested in:

- defining methods that promote the reuse of existing formal software components such as composition, restriction, extension, promotion, generalization, specialization, adaptation and restructuring. These methods are based on theory in order to prove the correctness of the new component. This consists of proving some formal relationships between the new specification and the reused ones;
- modeling and assisting (planning, automating and controlling) these methods using our tool.

3 Component Generalization

The goal of the generalization operation is to incorporate genericity into the newly developed components. Generalization is dual to the instantiation operation. Generalization is defined as a transformation of a component C into a parameterized component C' , such that :

- C is usable in more contexts than C' ,
- C should be a semantic generalization of C' ,
- and there must exist an instantiation of C' that is isomorphic to C .

In the definition above, the component C can be a formal specification [14] or a code [16]. A parameterized component contains formal parameters and structures that allow it to be systematically specialized into a possibly infinite set of specific components. The main difficulty is to identify the good level of generality for each component. Highly specific ones have small chances of being

reused but on the other hand if a component is too general its reuse will often be useless. In most algebraic specification languages, genericity provides great flexibility on the desired abstraction level and with the formal parameters we can state the most important properties to be preserved after generalization.

The language Act One [4] is used to describe our specifications. To facilitate the reuse, we build an explicit data type TS with two parts:

- an auxiliary part including the importation of types defined in the library Act One, *lib_TS*, and the definition of non predefined types, *data_defs_TS*;
- a local part including the sort of interest *ts*, the local operations of TS, *opns_TS*, and their equations, *eqns_TS*. Equations are equalities between terms. Only positive conditional equalities are allowed.

Generalizing a data type consists of replacing some of its auxiliary types by more general types called formal parameters. The notion of *general* corresponds to the existence of a morphism between the formal parameters and the substituted types. First, we have to decide which sorts and operations should be generalized. Once this signature is defined we can decide on the most important semantic properties of the original specification that we want to preserve. Generalization is done in an incremental way, i.e. by abstracting one auxiliary type at a time. We define the generalization operation with the following arguments:

- C : a specification component containing the definition of the data type TS to be generalized;
- TS : a type to be generalized. TS can be either a parameterized type or not. The generalization of a parameterized type consists of (i) relaxing the constraints of the formal parameters or (ii) abstracting the other non formal auxiliary types;
- IMP : an auxiliary type to be replaced by a more general type;
- PF : a formal parameter PF;
- OPN: local operations of TS to be abstracted. Given a specification, several generalizations are possible. To avoid over-generalization which makes it harder to understand what the new specification accomplishes, the developer has to choose the operations which are appropriate for a good generalization according to their semantics.

We associate with the generalization operation the following conditions:

- (c_1): TS is a data type present in the component C;
- (c_2): TS must contain structures;
- (c_3): IMP must be a type present in the profiles of the constructors of TS. This condition avoids creating overly general types;
- (c_4): the morphism m from PF to IMP must be a homomorphism, i.e. $\forall t_1, t_2 \in PF, PF \vdash t_1 = t_2 \Rightarrow IMP \vdash m(t_1) = m(t_2)$ where t_1 and t_2 are terms of sort *pf*. This consists of checking if IMP is a model of PF by generating proof obligations.

We generalize the chosen operations of TS (see OPN) by using the following order:

- (g_1): replace the auxiliary sort *imp* by the formal sort *pf* in the operation profiles and replace auxiliary operations by formal operations in the equations. All the constructors of TS are generalized according to (g_1). (g_1) is called syntactic generalization since it supports the genericity without constraints (like in programming languages);
- (g_2): associate properties with the formal operations in order to achieve the desired level of generality. (g_2) is called semantic generalization.

If the conditions are satisfied, the generalization operation produces a generic type TC derived from TS and parameterized by PF. We notice that the original sort *ts* is replaced by the generic sort *tc* in the whole specification TS. All possible instantiations of TC should satisfy the properties stated in PF. To guarantee the correctness of the generalization, we have to prove that there exists a specification morphism between the general specification TC and the original one TS. This consists of checking if the class of models of TS is extended after generalization, i.e. no models are lost.

4 Assistance tool for component development and reuse

The aim of our tool is to assist the development of the formal part of a software component. The development is not just a one-shot activity but it involves decomposing a task into a number of subtasks and then recursively locating the appropriate operators applicable to these more specific subtasks. The decomposition is twofold: mastering the complexity of specifications and reusing existing specifications for some sub-problems.

Development operators. To assist our generalization operation we plan five development steps. Generalization is activated with the operator *Generalize Data Type* (see figure 2). This operator is domain-independent. It is a generic task model that can be instantiated in a particular application. We define a reuse operator with the following parts:

- an informal description of the justification of the use of the operator and its role;
- interactive parameters and the conditions imposed on their values. These parameters make it possible to take into account the developer's knowledge concerning the problem to be specified;
- a description of the action of the operator on the workplan;
- a metaprogram which describes exactly how to build the new specification by reuse;
- the operators that are applicable in the next development steps.

Planning. The left part of figure 2 represents a workplan which is an organized set of tasks already reduced (by application of operators) or to be reduced (planning role). A task represents work to be done in order to evolve some parts of the specification. The first subtask allows choosing the operations that can be generalized according to the chosen import IMP (see OPN, section 3). The second subtask allows associating properties with the parameter PF. The third subtask is needed to prove that IMP is a model of PF (see (c_4) , section 3). The last subtask is used to further generalize the parameterized type TC if necessary.

Automation. In Proplane, a specification is handled using a metalanguage called METAPROG. A metaprogram (a text in METAPROG) is a set of variable definitions. Each variable represents a part of the specification to be built. The specification associated to the workplan of figure 2 is given by mixing the syntax of Act One and the syntax of METAPROG. The goal is to facilitate the reader's comprehension by showing the structure of the new type TC and the specification parts to be built. In appendix A, we present the metaprogram in which the generalization operation has been implemented. Our metaprogram is based on a set of importation and manipulation functions of specifications (words in bold type). With the notion of METAPROG consisting of naming each part of a specification we can reuse a specification with the desired granularity.

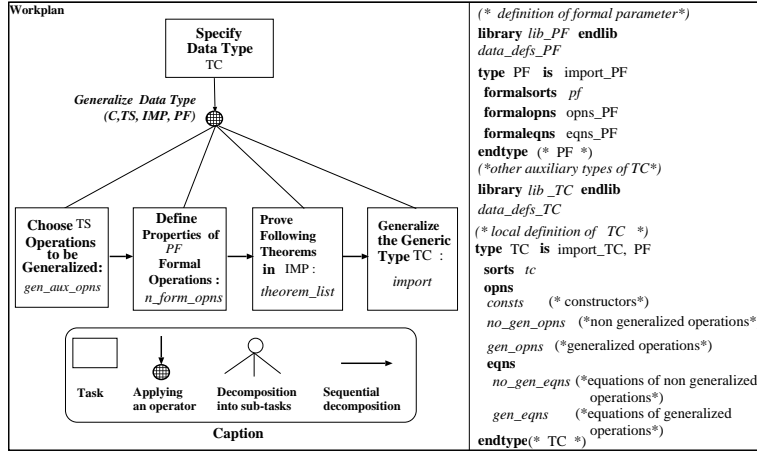
The principal activity of the developer is not writing the specification but focusing his/her attention on the reasoning to be followed in order to build the new specification. Then, we have to be able to optimize the automation of the majority of the tasks (syntactic or semantic) that can be difficult for the developers.

Characteristics of our tool. We define the formal specification of a software component as a pair $SPEC = \langle workplan, specification \rangle$. It is a matter of the external view. However, the subcomponent SPEC is recorded in the library as follows: $SPEC = \langle workplan, metaprogram \rangle$. At the time of the visualization of SPEC by the developer, the specification associated to the metaprogram is then generated automatically. With our tool, the reused specifications are not duplicated in SPEC but just referenced by their names. This allows saving considerable amount of memory and also creating links between components. These links are important for the reuse. Indeed they allow the components to benefit from the correction of errors in the reused components. The modification (evolution and adaptation) of a component is done in a new component because after modification the component will not still be usable by the other components. With these links, we can retrieve all the evolution and adaptation of a given component.

To assist the development and reuse of formal specification component, our reuse tool supports both textual and graphical notations and should allow:

Interactive Parameters

- C “Name of the component containing TS:”
such that $TS \in \text{types}(C)$ (c₁)
- TS “Name of the type to be generalized: ”
such that $\text{length}(\text{aux_sorts}(\text{constructors}(C,TS))) \neq 0$ (c₂)
- IMP “Choose the auxiliary type to be abstracted into a parameter
in the following list:” $\text{aux_sorts}(\text{constructors}(C,TS))$
such that $IMP \in \text{aux_sorts}(\text{constructors}(C,TS))$ (c₃)
- PF “Name of the formal parameter :”



Definition of the Metavariables: see metaprogram in appendix A.

Further Development Operators

- First subtask: *Give List of < Operation Names to be Generalized, Names of Formal Operations >.*
- Second subtask: *Define Properties, Define Properties with Existing Ones or Terminal.*
- Third subtask: *Prove Theorems or Terminal.*
- Fourth subtask: *Generalize Data Type, Restrict Data Type or Terminal.*

Fig. 2. Definition of the generalization operator

- recording the trace of the developer’s reasoning including the construction approaches, the decisions and their justifications. Furthermore, in a reuse process, this trace contains the names of the specifications that have been reused and also the relationships between the new specification and the reused ones. This information does not exist in the final formal text and represents however an aid to the documentation, comprehension and the further reuse of the specification;
- expressing non-monotone developments that are useful to maintain specifications. The maintenance is done by processing intermediary versions that can be tested and then challenged. This allows verifying the specification during its construction in order to produce directly a final specification that is correct;

- defining libraries of development operators. These operators allow us to define and assist different reuse methods;
- replaying the construction process. This type of reuse is called transformation-based reuse in opposition to component-based reuse [3]. With our tool, the development process can be reused because all the steps and decisions are recorded. The justification of choices are good aid for determining the steps that can be replayed and the steps that can be challenged. Replaying a step consists of instantiating the current operator with new values of its interactive parameters. One of the interests of this approach is to be able to replay in one step a whole development process. This approach is useful when we want to reuse and follow a certain construction approach.

5 Conclusion and future work

Our goal is to integrate formal methods into component-based software development process. To achieve this goal, we have to define methods that promote the reuse of existing components. These methods should be (i) based on formal methods (to produce correct components), (ii) automatic (because the developers are not necessary expert in formal methods) and (iii) interactive (to take into account the knowledge of the developers).

We have defined a method that assists the partial generalization of algebraic specifications. There are a number of other generalization problems such as:

- proving the correctness of the generalization operation;
- generalizing highly modular specifications and preserving modularity in the generic specifications. This generalization is possible only if the formal parameters are defined in a hierarchical way;
- benefiting from the validation of the reused components. In our case, given a theorem and its proof in the original specification, show how to reproduce the same proof on the generic specification [9];
- refining and implementing a generic algebraic specifications in order to produce the generic object oriented code.

The model Proplane is built with four different environments: GNU-Emacs to describe the workplan and the development operators, Centaur [1] to manipulate the specification and the language being used, TCL/ TK for the graphical representation of the workplan and TCP/IP for the communication between the platforms. The current implementation suffers from portability and from efficiency due to the slow interaction between the platforms. Also the platforms evolve without preserving compatibility between them. Our aim is to implement our own assistance tool for the development and reuse of formal software components. This tool can integrate JAVA for the graphical part and a generic environment for manipulating any specification language. This tool must have a distributed architecture to facilitate the integration of heterogeneous components such as theorem provers, syntactic analyzers, etc.

References

1. I. Attali. Description Action SmartTools, Document de travail. Technical report, INRIA Sophia Antipolis, <http://www-sop.inria.fr/oasis/SmartTools>, 1999.
2. D. Bert and P. Jacquet. Design and Reuse of Software Components based on Algebraic Specification Techniques. *ERCIM Workshop on Methods and Tools for Software Reuse, Crete, Octobre, 1992*.
3. L. Dusink and J. van Katwijk. Reuse Dimensions. In *Proc. of ICSE'95, 17th International Conference on Software Engineering, Seattle, WA USA*, pages 137–148, 1995.
4. H. Ehrig and B. Mahr. Fundamentals of Algebraic Specification. *ETACS Monographs on Theoretical Computer Science, Springer Verlag*, 6, 1985.
5. M. C. Gaudel and T. Moineau. A theory of software reusability. In H. Ganzinger, editor, *ESOP'88, Lectures Notes in Computer Science*, vol 300. Springer Verlag, 1988.
6. R. Hennicker and C. Schmitz. Object-oriented implementation of abstract data type specifications. In *M. Wirsing and M. Nivat, editors, AMAST'96, LNCS-1101, Springer Verlag*, 1996.
7. N. Lévy and J. Souquières. Modeling Specification Construction by Successive Approximations. In M. Johnson, editor, *6th International AMAST conference*, pages 351–364. Springer Verlag Lectures Notes in Computer Science, vol 1349, 1997.
8. N. A. Maiden. Analogy as a Paradigm for Specification Reuse. *Software Engineering journal*, January 1991.
9. A. Martins. La généralisation : un outil pour la réutilisation. *Thèse de troisième cycle, INPG, Grenoble*, decembre 1995.
10. B. Meyer. Library design. In *Tutorial Notes, Tools Europe'94, Versailles, France*, 1994.
11. F. Orejas. Algebraic Implementation of Abstract Data Type: A survey of Concepts and New Compositionality Results. In *Mathematical Structures in Computer Science*, pages 85–96. Cambridge University Press, 1996.
12. F. Parisi-Presicce and A. Pierantonio. *Reusing Object Oriented Design: An Algebraic Approach*. In E. Bertino and S. Urban, editors, *Object-Oriented Methodologies and Systems*, Lecture Notes in Compute Science, vol 858. Springer Verlag., 1994.
13. J. Quemada, A. Azcorra, and S. Pavon. *LotoSphere : Software Development with LOTOS*. In Tommaso Bolognesi, Jeroen van Lagemaat and Christopher A. Vissers, editors, 1995.
14. S. Sadaoui. Assistance in the generalization of algebraic specifications. In *Proceeding of the International Conference on Applied Informatics, IASTED, February 18-21, Innsbruck, Austria, 2002*. To appear.
15. S. Sadaoui and J. Souquieres. Some Approaches of Reuse in Proplane Model. In *Workshop AFADL-97, Toulouse, France*, pages 85–96, 1997.
16. M. Siff and T. Reps. Program Generalization for Software Reuse: From C to C++ . *SIGSOFT 96:Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, CA, October 16-18*, pages 135–146, 1996.
17. A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *Proc. of the 3th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 20(4):6–17, October 1995.

Appendix A: Metaprogram of the generalization

NAMES	TYPES	DEFINITIONS
lib_PF data_defs_PF imports_PF opns_PF n_form_opns n_aux_opns aux_opns eqns_PF	NAME_LIST DATA_DEF_LIST NAME_LIST OPN_LIST NAME_LIST NAME_LIST OPN_LIST EQN_LIST	(*definition of PF*) pred_types (imports_PF) definitions (C,IMP,no_pred_types(imports_PF)) iter_high (aux_sorts(opns_PF)) iter_rename (rename(aux_opns,imp,pf), n_aux_opns,n_form_opns) access_second (int_list) unif_second (gen_aux_opns,int_list) imp_opns (C,IMP,n_aux_opns) empty_list (*defined when the second subtask is reduced*)
lib_TC data_defs_TC old_types imports_TC	NAME_LIST DATA_DEF_LIST NAME_LIST NAME_LIST	(*auxiliary types of TC*) pred_types (imports_TC) definitions (C,old_types) no_pred_types (imports_TC) iter_high (aux_sorts(opns_TC))
consts no_gen_opns nogen_opns1 nogen_opnsi1 n_nogen_opns1 nogen_opns2 nogen_opnsi2 n_nogen_opns2 int_list	OPN_LIST OPN_LIST OPN_LIST OPN_LIST NAME_LIST OPN_LIST OPN_LIST NAME_LIST NAME_LIST	(*profiles of the non generalized operations*) rename (rename(constructors (C,TS), ts,tc),imp,pf) (*constructors of TC*) concat (nogen_opns1,nogen_opns2) rename (nogen_opnsi1,ts,tc) imp_opns (C,TS, n_nogen_opns1) napp_profil (imp,no_constructors(C,TS)) rename (nogen_opnsi2,ts, tc) imp_opns (C,TS, n_nogen_opns2) if int_list = empty_list then empty_list else delete (access_first (gen_aux_opns), access_first (int_list)) empty_list (*defined when the first subtask is reduced*)
n_gen_opns gen_opnsi gen_opns n_depends	NAME_LIST OPN_LIST OPN_LIST NAME_LIST	(*profiles of the generalized operations*) concat (access_first (int_list), n_depends) imp_opns (C,TS, n_gen_opns) rename (rename(gen_opnsi, ts, tc),(imp,pf)) depend1 (C,TS, access_first (int_list))
no_gen_eqns nogen_eqns1 nogen_eqns2	EQN_LIST EQN_LIST EQN_LIST	(*equations of the non generalized operations*) concat (nogen_eqns1,nogen_eqns2) imp_eqns (C,TS, n_nogen_opns1) imp_eqns (C,TS,n_nogen_opns2)
resp gen_eqns gen_eqns1	BOOLEAN EQN_LIST EQN_LIST	(*equations of the generalized operations*) false (*defined when the third subtask is reduced*) if resp then gen_eqns1 else empty_list iter_rename (imp_eqns (C,TS, gen_opns), n_aux_opns, n_form_opns)
opns_to_gen gen_aux_opns theorem_list import	NAME_LIST NAME_PAIR_LIST EQN_LIST NAME_LIST	(*information displayed on the substaks*) app_profil (imp,no_constructors(C,TS)) iter_auxopns (C,TS, opns_to_gen) iter_rename (eqns_PF,n_formal_opns, n_aux_opns)) iter_high (aux_sorts(import_TC))