MODULAR ALGEBRAIC SPECIFICATIONS
Horst Hansen, Michael Löwe

Abstract

Module, import, export and detail hiding are well-known notions in software engineering. If algebraic specifications together with their operational semantics of term rewriting should be useful as a programming language, similar concepts must be developed to cope with very large specifications. Especially the concept of detail hiding is important, since many problems and their efficient solutions require hidden sorts or hidden functions.

In this paper we discuss a module concept for algebraic specifications. A module consists of an algebraic specification, the body specification, together with import and export specifications. The semantics is given by a free functor from the class of import algebras to the class of body algebras followed by a subalgebra restriction according to the export signature. A module is correct, if its semantics does not change the reexport part of any import algebra. The reexport specification is the common part of import and export specifications. This requirement is much weaker than the requirements for module correctness we find in other approaches, e.g. in [EW 86], [BEP 87]. Since the restriction construction in our approach differs from the classical construction we are able to allow incompleteness and inconsistency of the body specification with respect to the import and reexport specifications.

This makes module specifications more suitable for software engineering purposes. For instance there is no need for explicit error handling in the body specification if the export restricts data to non-error data only. The example of a specification of a process scheduler demonstrates the usefulness of this concept. On the other hand, theoretic results concerning correctness of the semantics can be carried over to the new module concept. In this paper, we show that some basic constructions on modules preserve correctness and that the composition of correct modules yields a correct module.

Introduction

Term rewriting systems like OBJ [FGJM 85], ASSPEGIQUE [BCV 85] or RAP [Hus 85] provide operational semantics for algebraic specifications (for basic notions see [EM 85]). Thus, algebraic specifications written to be evaluated by these systems can be called algebraic programs. Since most of the problem-oriented specifications cannot be performed efficiently by term rewriting, algebraic programming means to construct an algebraic specification that is correct w.r.t. the problem-oriented one and can be performed by a special system efficiently enough to satisfy the users requirements.

Usually these algebraic programs resp. parts of them contain non-problem-oriented data types, functions, interpretation strategy information or system directives that should not be made reference to by the system user or by a programmer who uses the specification as a part of a bigger program; these details should be hidden from the user. Imperative and functional programming languages, like MODULA 2 [Wir 85] resp. ML [HMM 86], offer module concepts with explicit import and export for detail hiding.

Although the need of structuring facilities for large specifications has been recognized in the algebraic specification community (compare [SW 83], [EW 86], [Gogu 86]) the concept of detail hiding is not very well investigated up to now. Many approaches like [Gogu 86] provide only syntactical features which can be used to mark certain parts of the specification as hidden parts. Viewed parameterized canons as defined in [Rei 85] can be seen as modules with import (called 'parameter') and export (called 'view'). These modules do not have functorial semantics constructing export algebras from import algebras as the notions of [Rei 85] are based on behavioural semantics. In contrast [BEP 87] or [EW 86] define modules which reflect detail hiding on the semantical, i.e. functorial level.

These modules consist of body, import and export specifications. The semantics is given by a free functor from the class of import algebras to the class of body algebras followed by a restriction according to the export specification. The restriction models detail hiding. There are many module operations defined in this approach.

1 Institut für Software and Theoretical Computer Science, Department of Computer Science, Technical University of Berlin, Falklinstraße 28/29, D-1000 Berlin 10
like composition, union, actualization [BEP 87], partial composition and recursion [Par 87], product and iteration [Par 88] that make this concept easy to use in the process of software design. There is a concept of refinement and realization [EFHL 87] that constitutes a framework to express development steps like implementation or optimization. And the theory of this approach to module specifications is well-elaborated, too. There are a lot of compatibility results and results about compositionality of the semantics of combined modules (a comprehensive survey will be given in [EM 88]).

But the whole theory is based on very strong and restrictive requirements every single module must satisfy: The free functor from the import to the body must be conservative, which implies consistency and completeness of the body with respect to the import. This requirement seems too restrictive for software engineering purposes. The need for detail hiding very often arises from the fact, that certain hidden operations are not complete or consistent on all data. Hiding via export is then used as a tool to get rid of these (error-)cases. [BEP 87] requires explicit "error handling" (which is very hard to achieve in many cases, compare [Gogo 87] or [GM 87]) even if the export restricts to non-error data only.

Therefore we feel encouraged to propose a different module concept with restriction semantics, the theory of which is able to handle modules that hide inconsistency and incompleteness. Modules are algebraic specifications together with import and export specifications. The common part of the import and export specifications is called reexport. The semantics is given by a free functor from the import to the body followed by a restriction construction yielding an export subalgebra of the body. The restriction used here is different from the one in [BEP 87]. The theory will be designed for modules, the semantics of which preserves the reexport part. These modules are called correct. This requirement seems to be a natural extension of the concept of persistency for parametrized algebraic specifications to the level of modules. Note that this assumption does not imply completeness or consistency of the module's free functor. Thus, theoretical results are much harder to achieve. Especially the extension lemma which plays a central role in the theory of parametric specifications [EM 85] and module specifications [BEP 87] cannot be applied in our approach. We therefore prove an analogon for non-persistent functors. Based on that, we are able to show that some basic constructions on modules, i.e. export restriction, import extension, and module extension, preserve correctness. With the help of these results, we show that the composition of two correct modules (import export match) yields a correct module.

Contents

1. Algebraic Modules: Syntax, Semantics, and Correctness
2. Example Module: A Process Scheduler
3. An Extension Lemma for Non-Persistent Functors
4. Constructions on Algebraic Modules
5. Example: Composition of Modules
6. Further Research
7. References

1. Algebraic Modules: Syntax, Semantics and Correctness

In this section we introduce algebraic modules, their syntax and semantics as well as a notion of correctness of algebraic modules. These modules are intended as an abstract description of some aspects of software or program modules as known from modern high-level programming languages as e.g. MODULA 2 or ML. Algebraic modules consist of an import specification, an export specification and a body specification which are connected by an injective specification morphisms from the import to the body and another specification morphism from the export to the body. The import specification defines the class of allowable import algebras for the module while the export specification tells a user of the module what properties of the module he can rely on. The body specification describes a semantical construction relating the class of import algebras to the class of export algebras.

While in most modern programming languages the requirements on possible import algebras are just syntactical requirements given by some signature, in our approach, as well as in the one by [EW 86], these requirements can be enhanced by semantical requirements on the operations. The same is true for the export part of a module. This differs from many other approaches where properties of the export can only be stated informally by natural language comments.