Export Components Unify the Notion of Module and Subsystem in EMIL

Jürgen Börstler
Umeå University, Sweden

1 Introduction

The architectural design phase plays a central role in the development of large software systems. The resulting software architecture is one of the most important documents produced during the software life cycle. It lays the technical basis for all further development steps and the managerial basis for process and project management. The software architecture is an ideal guide to system understanding and the identification of reusable assets. The importance of software architecture was noticed more than twenty-five years ago ([BuRa 69]). The next important step towards a study of software architecture was the work of Parnas ([Parn 72]) on the decomposition of systems into modules. Starting with INTERCOL ([Tichy 79]) more and more languages where developed to describe the basic components of a software system and their interrelationships, i.e. the software architecture. At the same time people began to study actual software systems to detect commonalities across similar applications or within restricted domains (e.g. in the domain-specific software architecture programme [DSSA 90]). With the appearance of object-oriented approaches the discussions focused on frameworks (e.g. [KrPo 88]). In recent years there is a real revival in the study of software architecture. This work can be splitted into two categories: (1) the study of software architectural design, i.e. methods, languages, tools, and their underlying foundations to support the description of software architectures ([PeWo 92]), and (2) the study of concrete architectures to detect commonalities and define architectural patterns ([John 94]).

This work falls into the first category and focuses on some language aspect of the study of software architecture and how they are realized in the language EMIL. Of particular interest in the design of EMIL was the development of a notion for subsystems. For the development of large software systems subsystems are the first and most important components to identify. The importance of subsystems is well-known and pointed out everywhere in the literature. Surprisingly, there exists only very limited support for subsystems in current design languages.

In chapter 2 we give an overview over the language EMIL. Chapter 3 shortly reviews some existing approaches to support subsystems. The last chapter summarizes status and future work in the EMIL project.
2 An Overview over EMIL

The quality of a software architecture heavily depends on the application of well-known design principles like (data) abstraction, information hiding, encapsulation, structuring, low coupling, and strong cohesion. To support these principles a design language should cover concepts like (different kinds of) modules, subsystems, genericity, inheritance, usability, and locality.

**EMIL (Extended Module Interconnection Language)** is a language to describe modular software architectures according the principles described above. It is in the spirit of traditional module interconnection languages (MILs, [PDNe 86]) and developed as a pure design language. EMIL does not provide any control structures and is therefore independent from actual programming languages. Nevertheless can EMIL architectures be implemented in common imperative programming languages.

Compared to other design languages EMIL has a slightly different notion of module, which builds on the new concept of export components. These export components are the basic elements for the encapsulation of design decisions. An export component contains a cohesive set of resources. Different kinds of export components are provided to support different kinds of abstraction. Currently EMIL supports export components to encapsulate data types, data objects, functions, and simple components. The simple components exist for pragmatic reasons and can only contain constants and open types (i.e. non-opaque types).

EMIL modules make available export components on their (export) interface to be used by other modules. This new level of indirection allows for a natural and uniform notion and treatment of modules and subsystems.

```plaintext
datatype_component ExampleComponent
(* To encapsulate bibliographic references *)
datatype Reference;
methods
Constructors (* Special identifier to group and classify methods *)
create( in author, title: String; in date: Date): Reference;
Selectors
author( in r: Reference): String;
title( in r: Reference): String;
date( in r: Reference): Date;
...
end
```  

**Figure 1: A simple export component encapsulating a datatype.**

Modules and subsystems both provide (lists of) export components in their export interfaces. In the case of modules, there apply some restrictions concerning the composition
of this list. The real difference between modules and subsystems lies in their im-
plementation.

Module implementations are atomic in the sense, that they are realized by single
units, which are not further decomposed into modules and subsystems. Subsystems
have complex realizations, which are described by separate (sub-)architectures (see fig-
ure 2). In the current textual version of EMIL this subarchitecture is given by a list of
modules and subsystems, following the keyword is_subsystem. For each resource\(^1\) listed in
one of the export components of a module, there has to be a corresponding resource ex-
ported by one of the modules or subsystems listed in the realization. To give the software
architect control over this plug-in mechanism, the language supports explicit selection
and renaming of resources (see figure 2). To support the bottom-up development of hier-
archical architectures and the reuse of existing modules and subsystems, tools can be
supported to generate higher level interfaces from existing assemblies of modules and
subsystems.

```emil
module SimpleReferencesDataBase
  interface_servers
    from AnotherModuleOrSubsystem import String, Date;
  exports
    (* The services (in terms of export components) provided by the module *)
    datatype_component ExampleComponentForReferences
      (* see figure 1 *)
    end
  dataobject_component ReferenceRepository
    methods
      (* The typical operations to manipulate a collection of elements *)
    end
  realization_servers
    (* Further (probably qualified) modules, which may be used in the realization part. They do *)
    (* not contribute to the export interface of the current module (as do the ones in the realization) *)
  realization
    (* It is first in this private part, where the differences between *)
    (* modules and subsystems become apparent *)
  is_subsystem
    (* Partly given bindings for the export component 'ExampleComponentForReferences: *)
    SomeModuleInTheSubArchitecture
      where OneOfItsExportComponents ⇒ ExampleComponentForReferences
      renames aRessourceInTheRealization to Reference, ...
    end;
    ...
    (* More realization modules may follow here *)
  end
```

Figure 2: An example subsystem.

\(^1\) A resource is a type, a method, or a constant.
The unification of the notion for module and subsystem is of great help in the development of large software systems. First, one can abstract from the complex realization of a subsystem. Second, one need not even care about a component being a module or a subsystem before it has to be realized. The additional level of indirection in the definition of module realizations supports the independent and distributed development of subsystems, which is very important ([CFGG 91], [FeSch 93]).

EMIL supports non-strict multiple inheritance. Since we distinguish between different modes of parameters for operations, we can support co- and contravariant redefinition in restricting their applicability to the appropriate modes.

The concept of genericity is defined independent from inheritance. This has two advantages compared to solutions connected to inheritance. First, it allows for greater freedom in the definition of formal generic parameters, i.e. they need not necessarily be types. Second, it does not restrict the actual generic parameters to concrete subtypes of the formal ones. The first point enables more generic components to be defined. The second point enables more generic components to be used, since it frees the generic component from name space dependencies and actual type systems. EMIL supports unconstrained as well as constrained genericity.

EMIL is statically and strongly typed. It supports genericity and multiple inheritance. This allows for great flexibility in the definition of the components of software architectures and their interconnections, but still facilitates extensive consistency control.

The ability to express, analyze, and control architectural constraints is seen as very important by several authors ([LN 85], [Wegn 87], [PeWo 92]), because this ability provides some means for the development of certifiable software components. Since architectural design should always have reuse in mind (development-for-reuse) this is especially important ([Bigg 84], [Börs 93b]).

3 A short Subsystems Review

Reviewing the literature uncovers surprisingly few material on subsystems. The literature work on subsystems can be grouped into three main groups:
1. Approaches without any special notion for subsystems (most of them).
2. Approaches viewing subsystems solely as subjects of version- and configuration control (e.g. [Tichy 79]).
3. All others. Only these are of further interest.

Almost all remaining approaches treat subsystems as collections of modules, where some of these modules are visible outside the subsystems. The interface of a subsystem is then the sum of the interfaces of all the visible modules (e.g. [Noha 87], [Madh 88],
and all the object-oriented approaches, which have adopted Smalltalks Class Categories, [GoRo 83]). Taking a closer look this simple solution has several deficiencies compared to EMIL. These deficiencies complicate top-down development, maintenance, and (re-)use of architectures, i.e. such subsystems are not perfectly in the spirit of the well-proven design principles like abstraction, information hiding, etc.

- The top-down development, which seems to be natural at higher levels of abstraction is not very well supported, because there has to be some lower level component first, which can contribute to the interface.
- Subsystem interfaces are always concrete in the sense that there is some lower level component visible in its interface.
- Replacing a module in the realization of a subsystem always affects the subsystems interface, if this module contributes to it. This is even the case, when this new module provides identical resources.
- The reuse of subsystems is not well supported, because “plugging-in” other lower level components may affect clients of the subsystem.
- Another reason why reuse is not well supported is the fact that those subsystems are to concrete. You have to carry around all of the interface modules to compose the subsystems interface. EMIL subsystems in contrast are more abstract, because subsystem interfaces can exist without realizations.

4 Status and Future Work

Several tools to support the development of software architectures using EMIL have been implemented in the context of the IPSEN project ([ELNSS 92]). There are a textual and a graphical editor for EMIL as well as several analysis tools to check the consistency and completeness of (sub-)architectures, e.g. for the bindings done in the realization part of subsystems. With its introduction of export components, the unification of the notions of module and subsystem, and its independent definition of genericity and inheritance, EMIL already now provides rich support for the definition of reusable software components.

Current and future work on and with EMIL (will) follow three directions:

- The application of EMIL to detect typical patterns of subsystems and their development process.
- Extension of the language itself (for example to provide a notion for concurrency).
- Development of further tools (with special respect to reuse aspects, [Börs 93a]).

References


