Groupie

—

An Environment supporting Group-Oriented Architecture Development

Wolfgang Emmerich and Wilhelm Schäfer
Universität Dortmund
Informatik 10
Baroper Str. 301
44227 Dortmund
Germany

Abstract

The paper presents an architecture definition language and various mappings to different programming languages. In addition, a major new idea is that the language contains features which are exploited to define a concept to support cooperative distributed development of architectural descriptions. Besides the language and group-ware concepts the paper sketches functionality and implementation of the corresponding support environment called Groupie. The section describing the implementation of Groupie illustrates the suitability of an object database to build software engineering environments supporting multiple users.

1 Introduction

The architectural description has become an important, if not the most important document of a software systems life cycle. This description provides a possibility to structure a software system into well-defined parts called modules, subsystems etc. and to encapsulate the main design decisions within those parts. It serves as an important basis to guarantee maintainability (adaptability and portability), to reuse parts of a system in the same and across project(s), to systematically test and debug parts of (large) systems and to coordinate integration testing. The architecture description can even be used as a basis for project scheduling and work assignment.

Not surprisingly, a number of approaches based on data abstraction and in general the idea of object-oriented design have increased to rather widespread use as a paradigm for architectural descriptions. Those approaches include either languages which are a combination of a design and implementation language like Ada, Modula-2 or Eiffel or they are rather independent from any particular programming language. The latter ones are frequently called Module Interconnection Languages (MILs). Examples include languages like π [CFGGR91], HOOD [HOO89], Instress [Per89], CONIC [KMS89] and others. Our approach is not very different from these concerning just the language for defining modules and module dependencies except that we provide different types of modules which support structuring large complex software systems more than just using only modules of one type. We also provide precise definitions of mappings from the architectural definition into existing programming languages (like C, ML, Modula-2 and others) which are not provided at least in that detail by the others.

What is missing in most of the above mentioned approaches is support for concurrent development of architectures (and possibly implementations) by distributed cooperating teams of developers. Our experience, which is based for consulting on a number of industrial projects and software houses in architectural design, tells us that the architecture and implementation of almost any non-trivial size software project is developed and maintained in a distributed and concurrent fashion, but no appropriate support is available for this activity. Only Inscape [Per89], the environment supporting Instress and an environment called STILE [SW91] address the issue of support for multi-user, concurrent development. Both, however, propose a rather general strategy incorporated in their environment without giving details of the environment's functionality resulting out of the concerning strategy and, more importantly, no idea is given how this strategy can be implemented

In more detail, STILE proposes a strategy how to exploit knowledge about the semantics of software documents to improve the number of concurrent accesses to the same document. This is exactly what we do in our approach, but we combine this idea with the use of real languages and corresponding documents. This makes the approach really applicable especially for architecture development. In addition, aspects like negotiations through automatic composition of mails, automatic mailing, selective change propagations and the description of our implementation are unique to our approach. The Inscape approach sketches a concept of enforced and voluntary cooperation by building subsets and workspaces of modules. This is different from our approach because we have defined subsystems in a way that they additionally support structuring the architecture. In addition, subsystems are a precisely defined syntactic construct in our language. In fact our approach could probably be used to implement enforced and voluntary cooperation as proposed for Inscape. Finally, the defined mappings into a number of existing programming languages make our approach very widely applicable.

In summary, the main new features of our approach are:

1. The use of specific features of our architecture definition language to support not only structuring complex architectural specifications but also to define team structure and negotiation rules for managing software projects,
2. mapping to various programming languages,
3. the implementation of a support environment called *Groupie* which supports all the above sketched concepts and supports the areas programming in the small, programming in the large and project management (the latter one to some extend only so far) in a tightly integrated way which supports incremental, intertwined development of all areas concerned,
4. the support of team coordination based on the defined team structures through a built-in but adaptable message communication facility,
5. automatic Makefile composition by exploiting the knowledge about module dependencies in the architecture definition and
6. the use of an object database system as the underlying platform of our support environment.

As a side-effect, the construction of Groupie illustrates the adequacy of object database systems for software engineering applications in general as already argued in [EKS93].

It is worthwhile to mention that the approach presented in this paper have partly been commercialized. Ongoing improvement of the approach is based on a daily use of the system in industrial applications and frequent interaction with those industrial users (cf. Section 6).

The paper is structured as follows: The next section describes the architecture language. Section 3 defines how the language features are used to support multi-user architectural development and maintenance. Section 4 presents the functionality and user interface (derived from the concepts in the previous sections) of Groupie. Section 5 sketches the implementation of Groupie using the fully object-oriented database system GemStone [BMO$^+$89]. Finally, Section 6 concludes with an overview about the commercialisation of Groupie and sketches ongoing and further work.

2 Architecture Definition Language

2.1 Module Types and the Use-Relationship

The language is based on data abstraction and information hiding in the sense of object-oriented languages, i.e. an architectural description consists of modules each of which basically encapsulates a type and its operations. The interface of a module defines the exported type and the exported operations (export interface) as well as the imported types and operations of other modules (import interface), whereas the body contains the implementation of the export operations possibly using other internally defined (hidden) operations. The import/export relationship between modules is called use-relationship hereafter. Logically, a module represents one design decision. Main parts of the language have been described in [Lew88]. We will start with a brief introduction into the language and then concentrate on the extensions made by us.

The so far rather conventional notation distinguishes itself from other languages like Ada or Modula-2 by providing additional syntactical constructs to structure large and complex architectures. First of all modules have one of four possible types, which are

an abstract data type (ADT) that encapsulates one type and its operations, i.e. it basically corresponds to an abstract data type or class in the object-oriented terminology, i.e. it does yet not provide a formal specification language to define the semantics of operations by e.g. algebraic equations (c.f. Section 6).

an abstract data object (ADO) that encapsulates a type and its operations but it only exports operations and no type definition. It thus specifies that only one object of this type can be instantiated later during runtime because no other module can create an object of that particular type,

a function module (FM) that encapsulates a group of functions without any related type. Such a module itself has no internal memory and the invocation of any export operation has exactly the same effect in any case irrespective of the state of an (internal) object, and

a type collection (TC) that provides a construct to define system-wide basic types (like e.g. boolean, integer, or subtypes thereof) which are derived by renaming or combining predefined types and used to specify especially parameter types of exported operations in other modules. The reason not define them in separate ADT-modules is that they unnecessarily blow up the size of a system architecture since they are more geared towards a later implementation of the system.

Modules and their use-relationships have a textual and a graphical representation. As the example in Figure 1 illustrates, the textual representation provides more details than the graphical one. However, the graphical view provides the user with a traceable overview of the system.

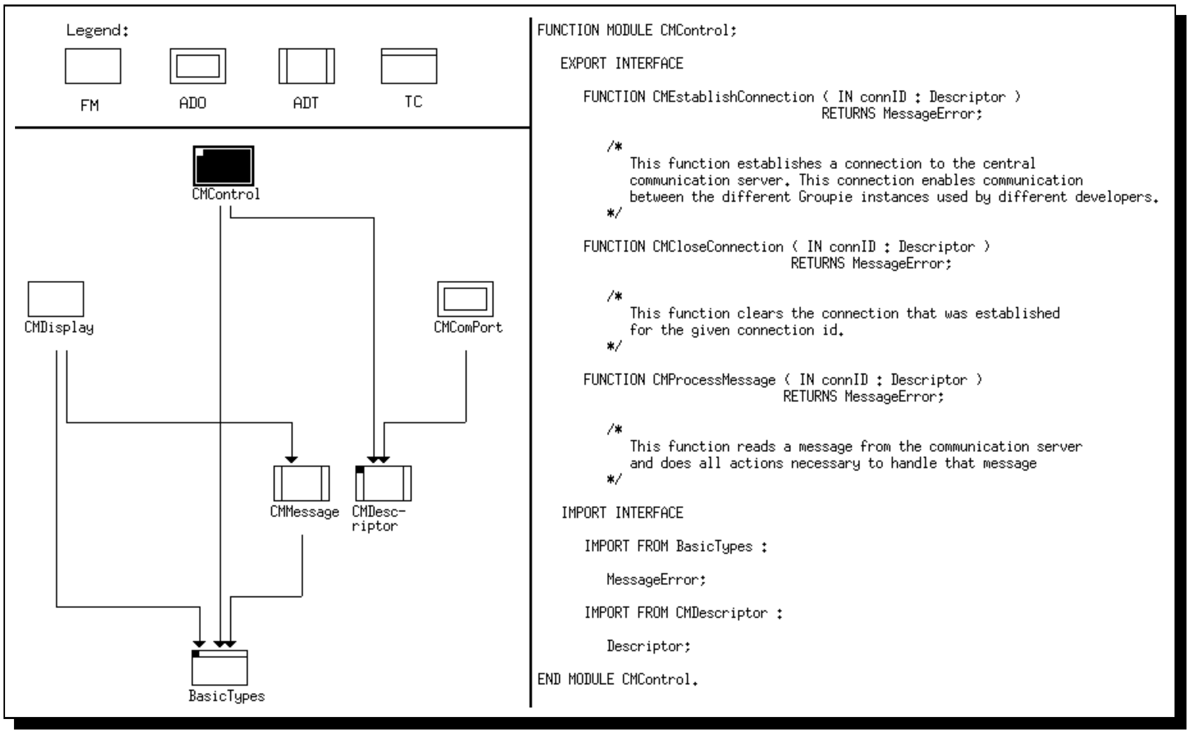

Figure 1: Module types and module relationships (graphical and textual notation)

The figure gives a small excerpt of the architectural description of Groupie itself.[1] At this point in the paper, it is only important to understand that the arrows in the graphical depiction represent use-relationships between modules and that the textual description in Figure 1 exemplifies the description of an interface definition of a module, namely `CMControl` in this case.

2.2 Subsystems

An important observation is that architectures of non-trivial software systems tend to become rather complex if they are only described by modules (or classes respectively) and their relationships. We therefore have introduced an additional construct called subsystem in our language which supports structuring collections of modules into larger grains. A subsystem summarises groups of the above defined modules and provides a common interface to other subsystems or modules. The export interface of a subsystem is defined by the sum of all export interfaces of all modules which are identified by the system architect(s) as export modules. The import interface consists of all types and operations imported by any one module within

[1]Groupie's architecture has been fully defined by using Groupie as the architecture support environment.

the subsystem. In addition, subsystems can contain other subsystems such that an architecture is basically constructed as a hierarchy of subsystems. An external view of a subsystem is given by a set of export and import relationships without identifying the particular modules which export or import the types and operations. Thus, a subsystem transfers the idea of information-hiding from the programming level to the architecture level. The principle idea of subsystems is very similar to the notion of systems in [HP81]. The way how we effectively use the subsystem construct as the facility to support multi-user concurrent development is however unique to our approach. This is underlined by the very detailed definition of the visibility rules (as given below), i.e. the definition of subsystems from which a subsystem can potentially import.

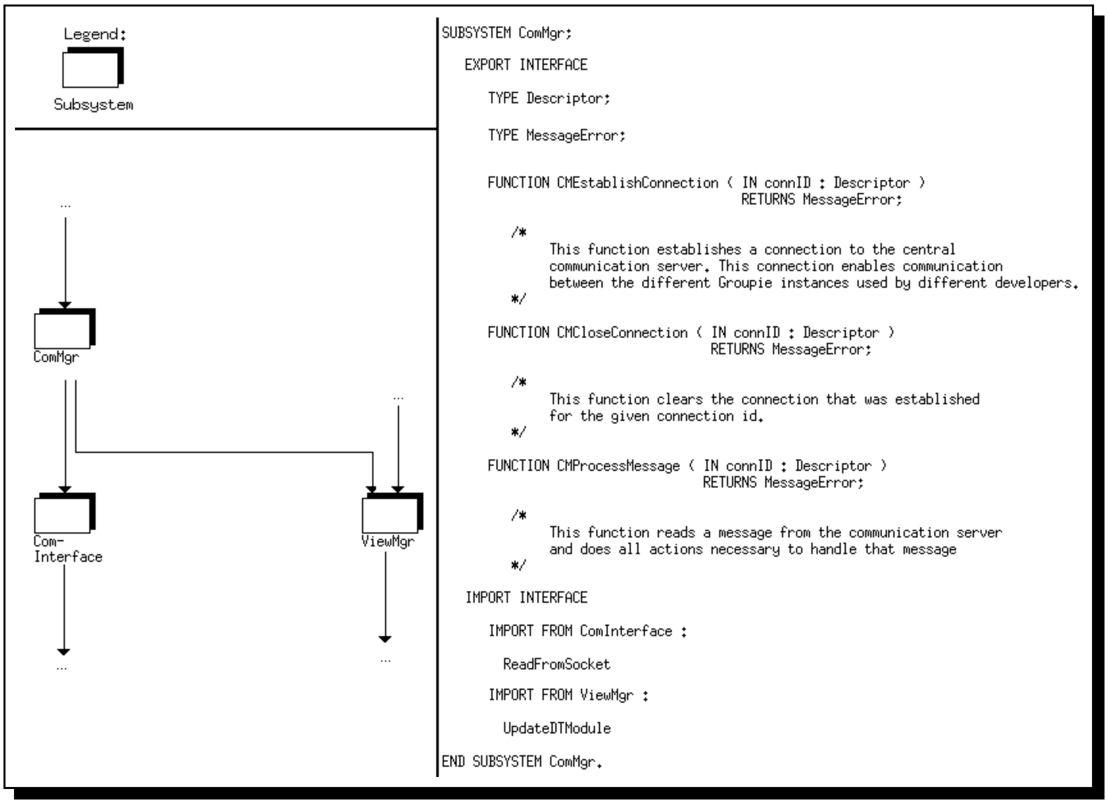

Figure 2: Example of a subsystem (graphical and textual notation)

As an example for the textual and graphical notation of a subsystem cf. Fig. 2, which illustrates the embedding of subsystem `ComMgr` in an architectural description by use-relationships and the detailed textual definition of its interface. The graphical representation of the internal structure of this subsystem and a part of its textual representation was given in Figure 1. Now we are able to explain the little black square in the upper left half of modules `CMControl`, `CMDescriptor` and `BasicTypes` in Figure 1. Such a black square indicates that the module's export interface contributes to the export interface of the enclosing subsystem. The export interface of subsystem `ComMgr` is hence defined by the sum of the export interfaces of modules `CMControl`, `CMDescriptor` and `BasicTypes`. What is not given in Figure 1 is the definition of the import relationships, i.e. which module's import interfaces contribute to the import interface of the subsystem. In summary, a subsystem has four different representations (containing various degrees of details). These are the graphical and textual representation of the interface as in Figure 2 and the graphical and textual representation of its internal structure which, according to the information-hiding principle, is not visible from the outside. (The supporting environment, Groupie, provides a zoom-in and zoom-out command to visualise interface and internal structure on demand if a person has the appropriate access rights (cf. Section 3 and 4)).

Following the above explanation of the context-free part of the language and its graphical and textual representation we now describe the static semantics of the language. We will not do this in full detail but rather concentrate on those parts which also effect the multi-user support as explained in the next section. It will be shown that there are quite a number of dependencies in the way static semantics are defined and how the language (and its supporting environment) supports multiple users.

The interesting constraints are:

1. A hierarchy of subsystems builds a hierarchy of nested name scopes (similar to a program language block structure). However, as a difference to nested blocks, any subsystem carries a name which is unambiguous in its scope. Thus names in different modules or subsystems are unambiguous by adding (conceptually) the name of the enclosing module or subsystem as an prefix.
2. Any imported type or operation must be defined in the export interface of another module or subsystem.
3. A subsystem B contained in another subsystem A can only import from modules or subsystems which are either (1) also contained in A, i.e. they are brothers of B concerning the contains-hierarchy of subsystems, or (2) they are already imported from A. (This restriction guarantees the mentioned information-hiding principle on the architecture level.)

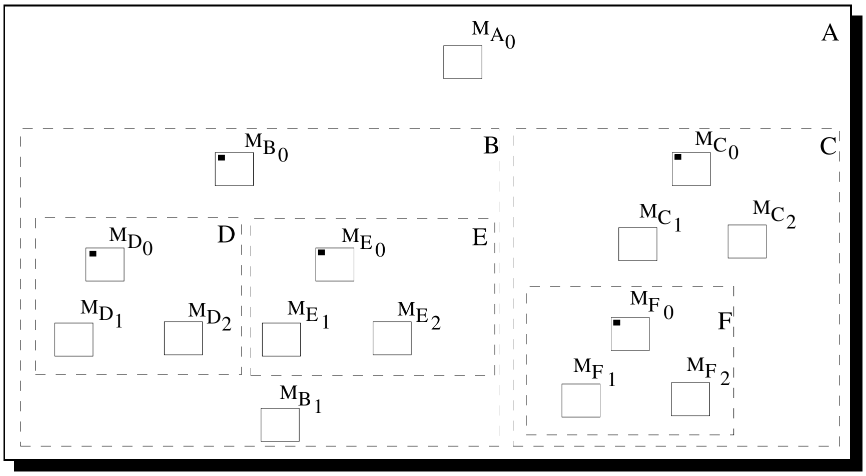

Figure 3: A Hierarchy of Subsystems

As an example for the last constraint consider Figure 3 which describes a hierarchy of six subsystems (and a few modules) and in particular indicates their internal structure. The full range of a subsystem is indicated by a dashed rectangle, i.e. subsystem A contains two subsystems B and C respectively as brothers in the subsystem hierarchy. Subsystem B in turn contains subsystems D and E respectively and modules M_{B_0}, M_{B_1}, whereas subsystem C contains a few modules (M_{C_0}, M_{C_1} and M_{C_2}) and subsystem F which in turn contains only modules anymore. For the sake of simplicity import relationships have not (yet) been defined in this example.

Under the constraint defined above module M_{B_0}, for instance, is only allowed to import from subsystems D and E and from module M_{B_1}, but never directly from subsystem C or its subsystems or its contained modules respectively. If we now introduce a use-relationship such that B imports from C, then this expands the scope of possible imports for all modules and subsystems contained in B. Then M_{B_0} could import M_{C_0} as it is the export of subsystem C. This exemplifies the second part of Rule 3 above.

Typical examples of subsystems are a set of application specific windows types (where in turn each window type may be represented by a module), a specific I/O device driver, a database system or the application-specific access interface to a database system. Note, that our architecture language does not restrict to write the bodies of modules in different programming languages.

2.3 Mapping Architecture Definitions to Programming Languages

So far, we have defined three mappings of the architecture definition language to programming languages which are representatives for partly different underlying paradigms. They include

1. a mapping to C which had to bridge a gap to a rather low-level programming level with no modularisation concepts,
2. a mapping to standard ML [MTH90], which had to bridge a gap to a functional programming language,
3. and a mapping to Modula-2 (which is that straight-forward, that we do not address it any further in this paper).

Figure 4 exemplifies this mapping. It displays the result of mapping module `CMControl` that was used in Figure 1 already to ML and C.

```
use "BasicTypes.sml"
(* Import:
   MessageError
 *)

use "CMDescriptor.sml"
(* Import:
   Descriptor
 *)

signature CMCONTROL =

sig
   val CMEstablishConnection : cmdescriptor.Descriptor -> basictypes.MessageError
   (*
       This function establishes a connection to the central
       communication server. This connection enables communication
       between the different Groupie instances used by different developers
   *)

   val CMCloseConnection : cmdescriptor.Descriptor -> basictypes.MessageError
   (*
       This function clears the connection that was established
       for the given connection id.
   *)

   val CMProcessMessage : cmdescriptor.Descriptor -> basictypes.MessageError
   (*
       This function reads a message from the communication server
       and does all actions necessary to handle that message
   *)

end; (* signature *)
```

```
#ifndef __CMControl
#define __CMControl

/*-import-*/
#include "BasicTypes.h"
/* FROM BasicTypes IMPORT */
/*   MessageError */

#include "Descriptor.h"
/* FROM CMDescriptor IMPORT */
/*   Descriptor */

/*--*/

/*-export-*/
extern MessageError CMEstablishConnection (Descriptor connID);
   /*
    * This function establishes a connection to the central
    * communication server. This connection enables communication
    * between the different Groupie instances used by different
    * developers
    */

extern MessageError CMCloseConnection (Descriptor connID);
   /*
    * This function clears the connection that was established
    * for the given connection id.
    */

extern MessageError CMProcessMessage (Descriptor connID);
   /*
    * This function reads a message from the communication server
    * and does all actions necessary to handle that message
    */
#endif
```

Figure 4: Function Module mapping to ML and C

The most important problem to be addressed by the C-mapping is the absence of modularisation concepts. We address this problem using the C preprocessor. We therefore map each module interface specification to a function header document (an example for such a header document is displayed on the right-hand-side of Figure 4). Each of these header documents contains a `typedef` declaration for each exported type, and function prototypes which are declared using the `extern` linker directive (confer to the definition of the three functions in Figure 4). Then any import statement of the module interface definition is mapped to a preprocessor directive which textually includes the header document of the respective imported modules (The `include` statements of `ComInterface.h` and `BasicTypes.h` are examples for this). Thus, the preprocessor compiles a header document of a module into a file that textually contains all headers of all (transitively) imported modules. To assure that each header is only included once, a header document is guarded by a preprocessor switch which only declares the types and operations, if the document has not been included before (the `ifndef` directive takes care of this). In order to still provide the developer with information about imported types and operations, a comment is automatically inserted after each include directive that enumerates all imported types and operations.

C source code frames which are derived from body specifications contain at the very beginning a preprocessor directive which includes the header document of the respective module. Hence all types and operations that have been imported in the module interface are known during compilation of the C source code. Then a function definition is included for each exported or hidden operation. In case of hidden operations, the functions are declared with the `static` linker directive. Therefore these operations can only be used from within the source code document, but not from others. As an example for a C source code document, confer Figure 10 on page 18.

Opposed to C, standard ML does not suffer from the lack of modularisation concepts. For the ML mapping, we exploit ML `signatures` as implementations of module interfaces (the left-hand-side of Figure 4 displays such a signature). Their names are derived from the respective module name by translating them into upper-case letters. `Structures` are the implementation of body specifications. Their names are given in lower-case letters. Complete `signatures` are derived from interface specifications whereas the `structure` is generated as a code frame which needs to be completed manually. The `signature` derived from a module is attached to the respective `structure`. Therefore the `structure` definition loads its corresponding `signature`. Consistency checks between design and implementation can be done by the ML interpreter while checking for a signature match against the signatures which have been derived from the current interface specifications. Import relationships between structures are implemented by the `use` interpreter directive, which loads the respective structure into the interpreter (in the example the structures stored in the `CMDescriptor.sml` and `BasicTypes.sml` are loaded). Usage of an imported name is prefixed by the imported structure's name.
The problem we had to address with this mapping is the paradigm shift between an architecture definition language for imperative languages and a functional programming language. The only parameter passing mechanism in ML is call-by-value. This implies that functions only have a return value, but no in-out parameters. We addressed this in providing a language variant and also a Groupie variant that only offers IN parameters, hence we had to adjust our architecture language to the use of ML. Moreover, ML is much more powerful with respect to function parameters of functions, generic data types and inheritance. This clearly shows that architecture languages, which are completely independent from any programming language, are inappropriate for practical use, i.e. there is always a mutual influence on which language constructs are defined.

The two mappings discussed are summarised in Table 1.

Architecture Language	C	Standard ML
module interface	header document	`signature` definition
exported type	`typedef` in header document	`type`
exported operation	`extern function` declaration in header document	`val` in signature
in parameter in export	call by value parameter	type name
in-out parameter	call by value parameter of pointer type	—
import list	`include` of respective header document	`use` directive for interpreter
import	comment	comment
module body	source document	`structure` definition
type construction	—	TC modules: `type` ADT modules: `abstype`
operation	`function` in source document	`fun` in structure
hidden operation	`static function` in source document	`local fun` in structure
owner	comment	comment
comment	comment	comment

Table 1: Mapping between architecture definition language, and programming languages

3 Process Model

Work assignment and responsibilities are based on the notion of a subsystem. Any subsystem is assigned 1 to n developers who form a group, which is in charge of developing that subsystem.

To develop a subsystem means to design the architecture for that subsystem and to implement the bodies of all modules. (How Groupie supports the programming task will be explained in Section 4.4). Including the programming task into a group's work assignment (and not to identify different groups for architecture design and programming as often proposed) is based on the fact that our architecture language is still quite close to a programming language and thus a separation between architect groups and programmer groups would be rather artificial. In addition, the benefit of this decision is underlined by a lot of practical experiences with Groupie as a major part of Groupie has been commercialized and is in daily industrial use (cf. Section 6). Although as said, the abstraction level provided by our language is still somewhat close to programming, we will now illustrate that this is a very suitable level for supporting cooperative work in a software project. (Of course, this is especially due to the introduction of subsystems.)

Any group consists of its group members and a group leader who is called the owner of the respective subsystem. The leader controls the design of the interface of the subsystem assigned to that group and he or she is the negotiator with other group leaders which are owners of other subsystems on the same level of the subsystem hierarchy within the same enclosing subsystem. Thus, group leaders themselves in turn form a new group which is responsible for the next upper level in the subsystem hierarchy. In summary, the nesting of a group structure is analogous to the nesting of the subsystem hierarchy. Consequently, the overall responsibility for a system architecture would be with the owner of the top-level subsystem hierarchy (who could be called *chief architect*).

As an example consider the scenario given in Fig. 5 which is taken from the previous figure in the last section but annotated by the names of groups and owners. Group G_0 consists of the group leader O_0 (who is thus the overall responsible) and its members O_1 and O_2. Members O_1

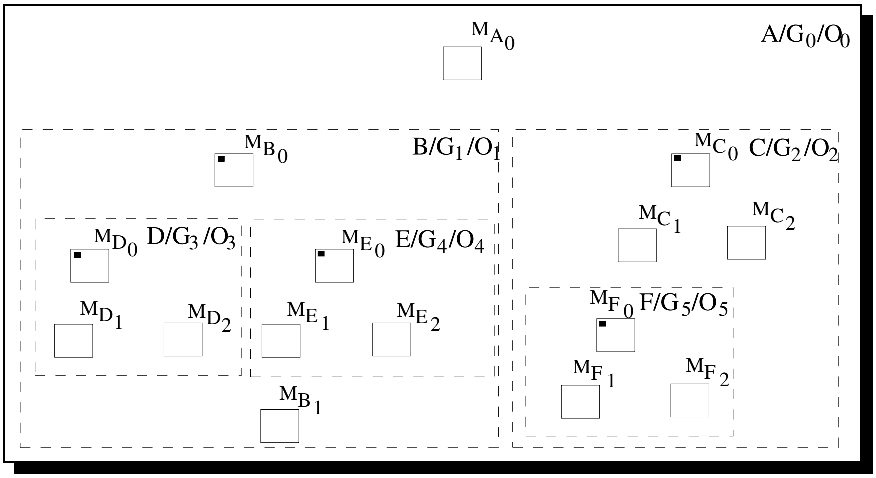

Figure 5: Sample Task Assignment

and O_2 in turn are group leaders (and members) of groups G_1 and G_2 and owners of subsystems B and C. As B is refined into subsystems D and E with groups G_3 and G_4 being assigned to them, group leaders O_3 and O_4 of those groups G_3 and G_4 form group G_1 together with their group leader O_1 and so on. The dashed rectangles visualise the areas of responsibilities, i.e. the ownership in the architecture. The annotation in the upper right corner of each rectangle gives the name of the subsystem, group, and group leader in the form subsystem/group/group leader.

The advantage of such a nested group structure is that necessary communication to achieve a consistent architecture is minimised and that even for large projects (we have examples of a few hundred modules and subsystems) the structure of the development team and the corresponding software is still easy to manage. Minimisation of communication is also the reason for not defining ownerships only for single modules. This strategy forces architects to build systems out of subsystems and not to spoil the architecture by basically not using subsystems. The more responsibility has to be shared (in usually large projects), the more subsystems have to be defined, whereas small single developer (sub-)projects can come along without using subsystems.

This advantage is however only fully achievable, if we put an additional constraint on the definition of export interfaces of subsystems. We do not allow subsystems to be exported by the enclosing subsystem. This would possibly result in a use-relationship between a fairly high-level subsystem and a fairly low-level subsystem, if the low-level subsystem is exported a number of times along the subsystem hierarchy. As a consequence communication structures, i.e. possible negotiations of group leaders could significantly increase. Logically, subsystems can be considered as abstract machines in a layered architecture. Therefore, export of a subsystem interface upwards through various levels of the hierarchy does not make sense logically either.

The read and write access rights for the different groups can now be automatically derived from the group structure by adhering to the information-hiding principle. A group leader has write access right to every module of the assigned subsystem. The whole group has read access right to every component interface from which a type or operation can be imported according to the static semantics constraints as given in Section 2. Note that this possibly means to see all export interfaces of all subsystems which are on a direct path to the root starting with the subsystem somebody is assigned to. Group leaders, in addition, get read access right to every subsystem included in the one which they are assigned to, on an arbitrary level of the subsystem hierarchy, because as negotiators in discussions within their groups they can thus better represent the whole subsystem and better decide on rearrangements of the architecture.

In summary, the strict policy on write access rights ensures a highly consistent architecture development without the possibility of a group to introduce bad side effects into the design of another one. Read access rights are granted more freely to ensure as much overview about (parts of) the architecture as possible without diluting this overview by too many details.

So far, our approach structures and minimises communication in building a consistent architecture without considering how to detect and resolve inconsistencies in the architecture if they happen to be included. Despite the introduced group and negotiation structure there are still a couple of cases where inconsistencies may arise. We intentionally allow those inconsistencies to arise in order not to hamper concurrent development of different subsystem.

In the first case, a developer can import a type or operation, although that type or operation has not (yet) been defined. If we would not allow this inconsistency, we would always enforce a bottom-up design as exports must always be defined, before they can be imported. This is a pragmatically not acceptable approach. The resolution of such an inconsistency is however nicely enabled through the well-defined group structure. If such a non-existing import is requested, a note (requesting such an import) must be sent to the owner of the subsystem where the import is requested from or even more, all owners of all subsystems from which an import is allowed could be noticed about the newly requested export.

In the second case, a consistent import could become inconsistent if the imported type or operation is changed in the corresponding export interface. The policy to recover from this inconsistency is that a change of an exported type or operation has to result in a note to every owner of a subsystem which imports that type or operation. The concerned group leaders can then get together and negotiate that change. In case they agree on it, the change can be performed system-wide or in case some owners do not agree the change must be performed selectively only to those subsystems whose owners have agreed to the change. Of course, if no one agrees, no change happens at all.

4 Environment Functionality

4.1 Overview

Groupie provides two syntax-directed tools to develop and maintain architectural descriptions. These are a tool to develop a graphical layout of a software system (based on the notations introduced in Section 2) and a tool which enables development of module interface definitions. Figure 6 depicts the user interfaces of these two tools.

As usual in syntax-directed editing, any syntactic construct can be selected using a pointing device. The tools deduce commands applicable to the selected construct and offer them in a pop-up menu. The developer can select a command, and Groupie responds after having executed the command with updating the display. (Of course, textual input of commands and text is also possible.)

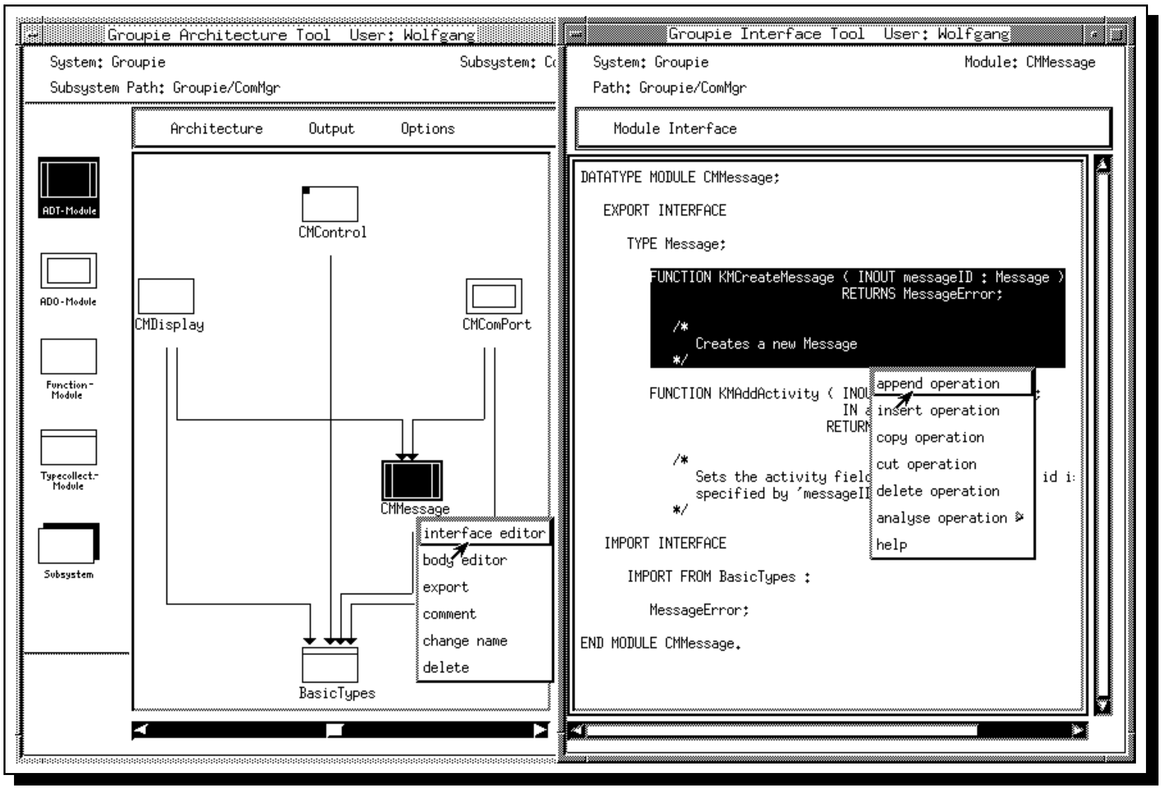

Figure 6: User Interface of Groupie's Architecture and Interface Tools

To work on a subsystem which is represented as an icon in the enclosing subsystem, a developer can use a zoom-in command which is offered in the menu when a subsystem is selected (confer also Figure 7). A zoom-out command is offered in order to change the display from the current subsystem to the enclosing one.

Groupie automatically maintains consistency between the graphical and the textual architecture depiction. If, for instance, a use-relationship is created by the graphical tool between the module `CMDisplay` and `CMMessage`, not only an arrow will be displayed in the graphical depiction between the two respective modules, but also a new import list will be inserted in the textual interface representation of `CMDisplay` .

Groupie commands support users in achieving correctness regarding static semantics. As an example, consider that we expand an import list. Then all exports of the respective module, which have not been imported so far, are displayed in a selector box and the developer can select those he or she wants to import. Similarly, all types declared within a module are displayed in a selector box, as soon as a developer wants to expand a parameter type or a result type of an exported operation.

For the developers' convenience, they may also enter imports or types which have not yet been declared. In this case, Groupie informs the user about an error. If the user confirms that his or her input was intended, the command will be executed and the resulting error is visualised by putting the erroneous parts into brackets.

Analysis commands may later on be used to see whether a module contains errors. There are two kinds of analysis commands: commands to identify static semantic errors and commands to visualise the dependencies between different syntactic constructs. Using the first kind of commands, a developer can identify erroneous constructs and navigate to their location. Using the second kind of command, a developer can identify where operations or types exported from a module are actually used. Finding obsolete operation or type definitions is also supported by this kind of analysis command.

4.2 Access Rights

We have introduced the notion of group-oriented access rights in Section 3. These are enforced by Groupie in the following way. If a new architecture is created by a developer, the developer is the leader and the only member of the group which by default works on the root subsystem of that architecture. He or she therefore becomes the owner of the new system.

If a new subsystem is created it is contained in an already existing subsystem. The owner of the existing subsystem is initially the owner of the new subsystem as well. He or she can change the ownership of the subsystem by a tool command applicable to a subsystem which is accessed by the owner. Using this command, a group leader can include a new member in his or her group and transfer the ownership for a subsystem to the new member. When doing so, a new subgroup of the group leader's group is created and the newly introduced member becomes the group leader of this subgroup.

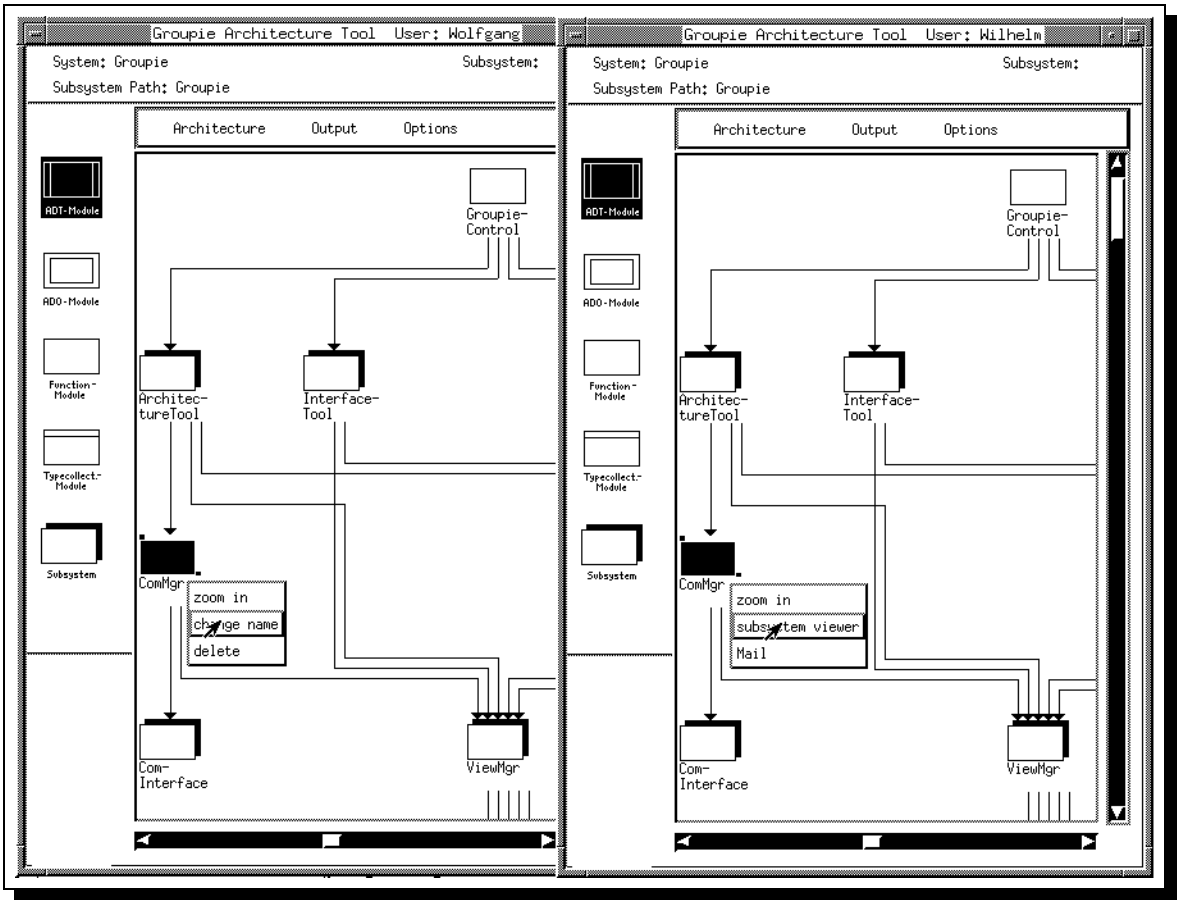

Figure 7: Enforcement of Access Rights in the Architecture Tool

When starting Groupie, developers have to identify themselves with their Groupie account. Groupie authenticates whether a person is a Groupie developer by requesting a password during startup. The access rights defined are enforced in the architecture tool by only offering those commands that respect the rights defined. If, for instance, the textual interface tool is invoked by a developer who has only read access granted, the tool works like a browser, i.e. no command can be invoked which would change the interface.

Figure 7 depicts the way access rights are enforced for the architecture tool. The figure depicts two instances of the architecture tool which display the same subsystem. One instance is used by developer Wilhelm and the other is used by developer Wolfgang. Wilhelm is the group leader of the subsystem displayed. As the displayed menus suggest, Wilhelm can not change Wolfgang's subsystem (`ComMgr`). Except for `ComMgr`, Wolfgang can only read the other components displayed, but not change them.

4.3 Concurrent Development

Groupie supports concurrent development, which allows multiple developers to work on an architecture at the same time. Two problems must be resolved to support concurrent development adequately. The first problem is that due to the access rights defined, negotiations on changes between developers may be required. The second problem is that despite the access rights defined concurrency control conflicts between concurrent Groupie sessions can occur.

Change Negotiations may become necessary, whenever a developer imports types or operations from some other developer's subsystem or module. Sometimes, he or she may then not be able to stick to the static semantics defined for use relationships. Three alternatives require negotiations:

1. a developer wants to import from another developer's subsystem which does not yet export the respective type or operation,
2. a developer wants to import from another developer's subsystem but can not due to a missing import of the enclosing subsystem, or
3. an export is changed that is used already in some other developer's subsystem as an import.

In the first case, the developer who imports a non-existing export must negotiate with the imported subsystem's owner to define the export. In the second case, the group leader is the developer to negotiate with on defining an import in the enclosing subsystem. In the last case, the developer who wants to change or delete an already used export must negotiate with the developers using the export on whether or not the change or deletion can be performed.

Groupie supports these kinds of negotiations based on its knowledge about use relationships between subsystems and the definition of ownership. Groupie incorporates a mail tool which is used for passing messages between developers. To anticipate the first two of the above mentioned cases, i.e. an export or subsystem import is not yet defined, Groupie offers a command to post a request to an owner of a subsystem. When selected, this command provides a text editor to write the request and then sends the request to the developer who is currently in charge of the respective subsystem. Please note, that Groupie enables the definition of an import although it is not exported and hence the developer can continue in specifying the respective module while the other developer is handling the change request.
Figure 8 depicts how this functionality is offered at Groupie's user interface. It depicts a

situation in which Wolfgang who is in charge of designing `ComMgr` requests a function from the owner of subsystem `ViewMgr`. Therefore, Wolfgang selects the subsystem `ViewMgr`. In the pop-up menu that displays the available command, a Mail command appears. After being selected, this command pops up a new window in which the request can be written. After the window is closed, the request is automatically sent to the developer responsible for `ViewMgr`.

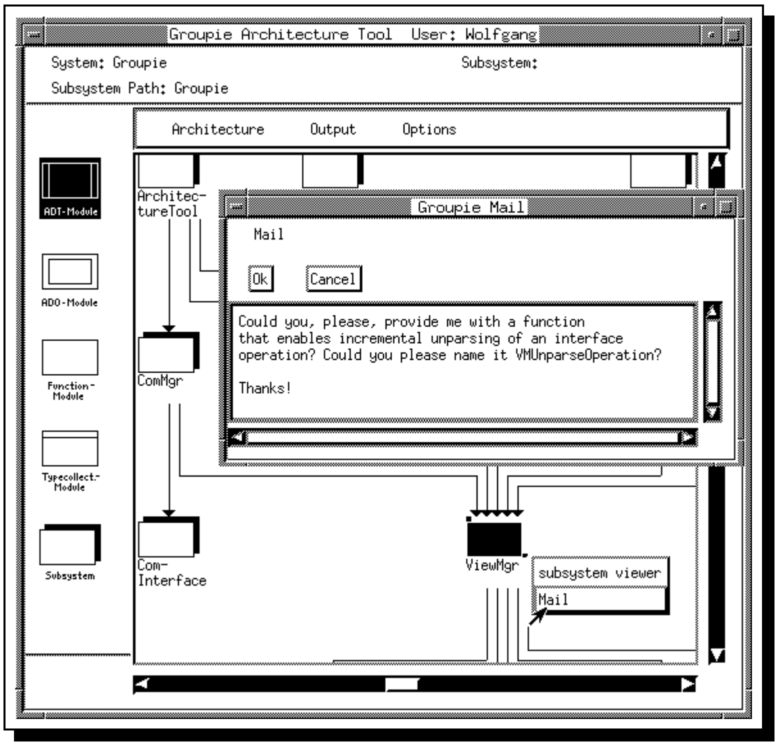

Figure 8: Requesting a Change in Groupie

To support negotiations between developers that are necessary due to a change or a deletion of an already used export, Groupie implements a two-phase message protocol. When a developer selects an export that is used and executes the tool command to change or delete the export, Groupie first of all warns the developer that the export is used already. If the developer confirms that the export is to be changed, Groupie requests a rationale for the change from the developer. It then sends in the first phase of the protocol a change request with the given rationale to all owners of subsystems in which the export is used. Each of these owners have to confirm that they have no objections against the requested change. In the second phase, Groupie awaits a confirmation of all these owners to perform the change. Only after having received all confirmations, Groupie performs the change automatically. In case of a change to an exported identifier, it then propagates this change to all places where the identifier is used and changes these places. In case of a deletion, it marks the using places to be inconsistent.
A developer who had requested a change can abort waiting for confirmations and then the change is done selectively only to subsystems of owners who confirmed the change. In the other subsystems, the respective imports are marked to be inconsistent.
If Groupie has performed a change propagation, it redisplays the effect not only on the workstation of the developer who actually made the change, but also on the workstations of all developers who work on subsystems in which the change propagation has resulted in changes.
Finally, an automatically composed mail is sent by Groupie to all owners who were involved in this change negotiations saying that change has been completed.

A Concurrency Control Conflict occurs if two different Groupie sessions (running under the same Groupie account) modify the same module or subsystem at the same time (write-write conflict). There is also a conflict if one Groupie session accesses a component which is at the same time modified by another session (read-write conflict). As an example of a read-write conflict, consider that one developer executes the tool command to import an operation. Then all exports of the respective module or subsystem have to be retrieved for presenting it in a selector box. The developer selects a particular operation. A conflict occurs, if the owner of the imported module or subsystem at the same time changes the export by deleting the operation the other developer wanted to import.
Groupie resolves these conflicts by regarding each tool command (like e.g. to import an operation or to change an operation's name) as an ACID transaction [Gra78]. The chance for a concurrency control conflict is fairly remote, because the accessed objects are of very fine granularity and the execution of a tool command only needs a few hundred milliseconds. Moreover, only a single command is involved in such a conflict and developers therefore can tolerate a small delay or even an abort of the command execution.
Hence, the above concurrency control conflict is resolved by Groupie in either of the following ways. In the first case, the delete command is executed earlier and then the developer who had issued the import command will get a message that the export is not available, or if the import command is executed before the delete command, then the developer who wants to delete the exported operation will have to negotiate about the deletion as described at the beginning of this subsection.

4.4 Programming Support

In Section 3, we have explained that subsystem owners not only define the subsystem's architecture, but also implement the contained module bodies. To support this implementation, Groupie provides two further tools, a body editor and a programming language tool.

The body editor supports a pseudo-code specification of the exported types and operations and a definition and pseudo-code specification of hidden operations. Therefore, the body editor by default displays all types and operations that have been declared in the respective export interface. It then allows the developer to specify the type constructions and to describe the algorithms of the operations' bodies in the pseudo-code notation. Figure 9 depicts the the body editor invoked on the body specification of Module `CMMessage` whose interface specification was sketched already in Figure 6.

The interface tool and the body editor can be used in an intertwined fashion. The effect of each change, which a developer performs in the interface definition of a module, is immediately propagated to the body editor. If a developer, for instance, defines a new exported operation, this operation will automatically be included in the body specification. We have decided not to support the reverse. This is because, we want to enforce designers to separate the concerns of interface design and implementation design. We therefore want to have them using the interface tool for interface design only and the body editor for implementation design only.

As an example for a programming language support tool, we describe the C programming tool. This C tool displays all operations that have either been defined as export in the interface definition of a module or as hidden operation in the body specification. It then allows the developer to select a place holder for a function body and to implement this body. The programming language tool is hybrid in the sense that it supports not only structure-oriented

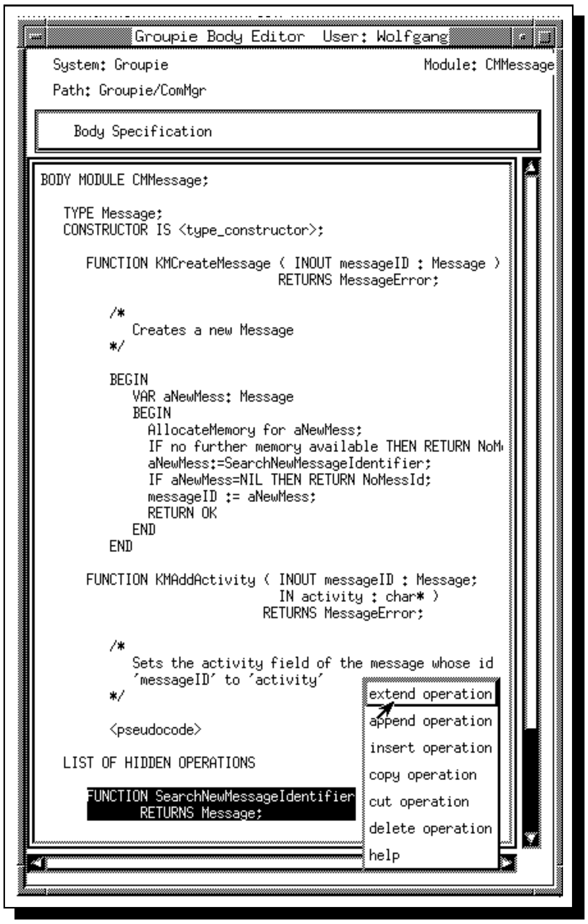

Figure 9: Body Editor

editing as do the other tools, but also enables free textual input with a text editor. It checks the static semantics of the C programming language as defined in the ANSI specification.

The integration of the tool with the other tools is done in the same fashion as for the body editor. As soon as changes are done, for instance, to an operation declaration in the interface definition or the body specification, this change is propagated to the programming language representation.

The functionality of the programming language tool that goes beyond that of an editor is concerned with static semantic analysis. The tool provides capabilities in order to find obsolete variable declarations or statically unreachable statements. Figure 10 depicts the user interface of the programming language tool. It displays the implementation of the function `KMCreateMessage` whose body specification is depicted in Figure 9.

Finally, the programming tool is capable of writing the implemented source code into the UNIX file system. It therefore maintains a directory hierarchy according to the subsystem hierarchy of the architecture definition. The written source code may then be translated with the system's C compiler in order to obtain the object code for the module.

Derivation of executable files from a set of source modules is most often controlled by `make` [Fel79] in a UNIX environment. The input for `make` is a Makefile which defines the dependencies between the different source modules and the build rules. The tedious task of

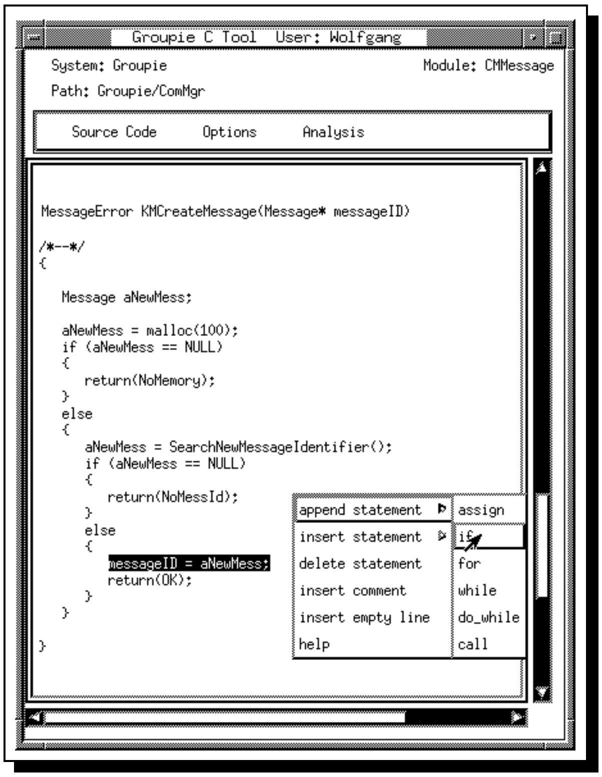

Figure 10: Groupie Programming Tool

manually writing and especially maintaining Makefiles becomes obsolete when using Groupie, because Makefiles are generated by Groupie. This is enabled by Groupie's knowledge about the use-relationships defined in the architecture which enable to derive the source module dependencies and due to the fact that Groupie knows the source module locations in the file system.

The other mappings to programming languages discussed in Section 2 have so far only been implemented as compilers which translate an architecture into initial programming language code frames. They could, however, as well be implemented as interactive tools tightly integrated into Groupie.

5 Implementation Issues

We now explain the main features of the Groupie implementation. We especially elaborate on how to use the object database system GemStone [BMO+89]. The suitability of object database systems as a basis for environment construction is discussed in [EKS93]. Exploiting GemStone's transaction mechanism and its client/server architecture, was an excellent basis for implementing the discussed negotiation and concurrency control strategies.

5.1 An object-oriented Schema for Abstract Syntax Graphs

Groupie's internal data structure is an abstract syntax graph.[2] Figure 11 depicts a partial display of such an abstract syntax graph. It sketches the part of the graph that represents subsystem `ComMgr` which we have displayed previously in Figure 6.

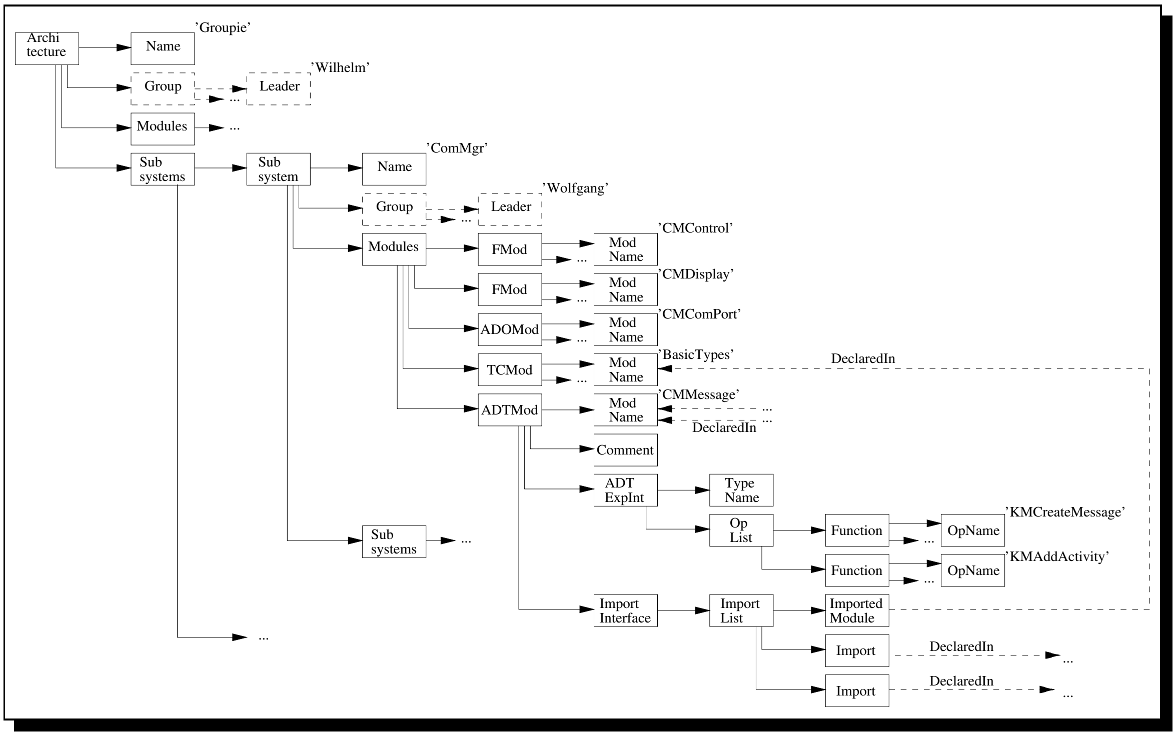

Figure 11: Excerpt of Abstract Syntax Graph used by Groupie

We want to store these abstract syntax graphs in a database without changing their representation because

1. it does not fit into main memory,
2. developers must not loose significant effort in case of hardware or software failures,
3. the efficiency required at the user interface will not be achieved if complex transformations between external and internal representation are necessary and
4. the same graph must be accessible by different users concurrently.

To define a GemStone schema for Groupie's abstract syntax graphs, the graph structure and the applicable operations have to be defined in OPAL, GemStone's data definition and manipulation language. OPAL evolved from Smalltalk [Gol85]. Therefore, common properties of nodes such as out-going edges or attributes are defined in OPAL classes. Nodes are implemented as complex objects whose instance variables implement edges or attributes. Integrity constraints on the abstract syntax graph are enforced by encapsulation, i.e. Groupie's tools are not allowed to directly modify instance variables, but must use the methods defined for that purpose. As an example, consider terminal classes such as `ModName` or `OpName`, which offer a `scan` method. This method provides the only facility to set the attribute for storing lexems

[2] A rationale why this is the appropriate data structure for environments like Groupie may be found in [ELN+92] and [ESW93].

of identifiers. It only stores a string passed as argument, if the string is a lexically correct identifier.

Objects have unique identifiers which are called object oriented pointers (OOPs) in GemStone. GemStone's programming interface which is used by the tools then provides means to send messages to OOPs. To execute a tool command such as expansion of an identifier placeholder, the tool sends the message `scan` to the OOP associated with the currently selected placeholder. All further computation of the tool command is then controlled by the database schema and executed within the database server process. The performance gain is that for executing tool commands no objects need to be transferred via expensive network communication from the database server to tools. After the operation is completed, its effect is displayed and all modified OOPs are associated with their respective displayed representation.

5.2 Using Optimistic Transactions for Command Execution

GemStone offers an optimistic transaction mechanism which performs concurrency control of concurrent sessions. After having started a transaction with ACID properties in GemStone, a session can work on a logical copy of the database. During commit of the transaction, GemStone checks whether the transaction has performed accesses to objects which are in read-write or write-write conflict to other successfully completed transactions. In this case the transaction can not be completed, but has to be aborted. A transaction abort updates the logical copy of the database taking into account all database updates made by successfully completed transactions. Then the transaction can be restarted. If the commit succeeds, the logical copy of the database is also updated.

GemStone's transaction mechanism is exploited in Groupie for executing tool commands. As soon as the user selects a command from a menu (or has finished its textual input), Groupie starts a transaction. The command is executed by sending a message to the currently selected object. After the message execution is finished, Groupie tries to commit the transaction. If it fails, a conflict occurred with a tool command executed concurrently by some other developer. Groupie then aborts the transaction, and automatically restarts another transaction again in order to resent the same message, whose parameters are still available. All this is done transparently to the developer and in many cases he or she will even not recognise the conflict at all.

Note, that using this transaction mechanism is only possible because we are storing the abstract syntax graph in the database as discussed in the previous subsection. Groupie's tools are then enabled to access those and only those objects necessary for executing a command. If for instance, an operation is imported, a new `Import` object is created and the set of `OpName` objects contained in the imported module object is searched for a match against the name of the import. If such an `OpName` object is found, this object is modified as well in order to establish a context-sensitive edge `DeclaredIn`. This transaction mechanism cannot be used if the database representation of the abstract syntax graph takes the form of some coarse-grained objects (such as files) from which the abstract syntax graph is loaded and into which it is stored. Then a lot of objects will be accessed unnecessarily. The result are many unnecessary concurrency control conflicts.

5.3 A Communication Server for Message Passing between Developers

Developers expect messages they sent to other developers that work in parallel to be delivered immediately. Only then can Groupie's negotiation mechanism work as a communication facility to support developer dialogues. Moreover, a document change (e.g. due to a change propagation) must immediately be displayed in every Groupie session that has opened the document, even if it is running on a different workstation. Otherwise, developers could get confused when their displays show document states different from those in the database (They may for instance not understand why they can not perform a particular import, although their display shows the export). To address this, only storing messages in a database and polling for them is inappropriate, because it is too inefficient. Instead an active component is required that handles inter-tool communication. The Groupie architecture therefore contains a communication server which is used for sending messages between parallel Groupie sessions.

Groupie needs to have four kinds of messages sent:

1. developer initiated change requests,
2. developer initiated negotiations with all users of a type or operation,
3. developer initiated confirmation to or objection against a requested change and
4. automatically generated messages about changed exports to all users of the exports, which also updates the users display if the respective document is opened.

The addressees of these messages have to be determined according to the current state of the abstract syntax graph. Messages may have to be stored persistently, if developers are not logged in. For developers who are working in parallel, their messages may have to be routed to other workstations in the network.

We have implemented these requirements by running the message server as a database session. When doing so, we manage to

- find the addressee of a message and
- store messages persistently if the addressees are not working.

As soon as a tool requests a message to be delivered to a developer, the server looks up the responsible for the subsystem or module in the abstract syntax graph. Therefore the object identifier of the respective subsystem or module is passed as argument of the message. The server then looks up in some internal tables whether the developer is currently working. If so, it looks up the display address of the developer's display and communicates the message to the Groupie session on that display. This, of course, requires that all Groupie sessions perform a server login which informs the server about the developer's account and the display address. If the developer is not working, the message server will store the message within the abstract syntax graph and display it after the next login.

For the implementation of this communication server, we have decided not to use one of the standard communication mechanisms such as the HP Broadcast Message Server [Cag90], DEC Fuse or Sun Tool-Talk since they do not support persistent storage of messages. In addition, the HP Broadcast Message Server and DEC Fuse cannot pass messages to remote displays.

6 Current and Further Work

The language as proposed here is currently improved by introducing the concepts of inheritance and genericity. A first proposal has already been made by [B93̈].

A multi-user version of Groupie called OPUS which does not yet include the message server functionality and does not yet support the explicit definition of access rights (but enables concurrent development of subsystems) has been commercialized by two local software houses STZ and ProDV. Those companies which together have a staff of about 100 people apply the C-version of OPUS in their mainly technical oriented customer-specific software development and in building and maintaining their software products which are a geographic information system and a production planning and control system, both consisting of a few 100.000 lines of C-code. In addition, OPUS is used in graduate and undergraduate software engineering courses at the University of Dortmund serving a few hundred students each year and the University of Leiden (NL). Several other universities have expressed interest in getting a licence. Finally, OPUS is part of the Kernel/2R software factory [ADH+92] developed as a result of the EUREKA project ESF (EUREKA Software Factory).

Current improvements of OPUS are the ongoing work of introducing the already available Groupie features concerning access right specification and message passing.

Current extensions of Groupie include the development of a version/configuration management system and the introduction of a possibility to make the Groupie process model more flexible. Currently, as explained above, the group model, i.e. the correspondence between group (owners) and subsystems and the negotiation paths between group owners are predefined and basically hard-coded into Groupie. By making this process definition more explicit by extending Groupie with a process definition language and a corresponding interpreter, the so far rigorously fixed group model can easily be changed. For example, new roles, new responsibilities and other negotiation protocols could be defined. As a first step in this direction Groupie has been integrated into our own process-centred environment called Merlin which uses a rule-based specification of software processes [JPSW94].

This is also a step in the direction of generalising the group-ware concepts in Groupie towards applying them to concurrent development of arbitrary documents which are highly dependent of each other. In particular, the negotiation and conflict resolution concepts and the corresponding implementation using an object database are independent from the specific languages used.

Two more long-term development efforts concern firstly the introduction of a formal language to specify the semantics of exported operations. The approach we intend to take is similar to the specification proposed for Inscape, i.e. the definition of logically different types of pre- and postconditions for the execution of each operation. Secondly, such a specification will serve as a basis to construct libraries of subsystems which can then be reused in different architecture descriptions as described in the Inscape paper. In fact, it is already possible right now to store subsystems separately from a particular architecture description and to include them as a *reused* subsystem into another architecture. As the specification of those subsystems is however yet only on the syntactical level of signatures of operations (as explained in this paper), the retrieval mechanisms are not very sophisticated and will be improved on the basis of formal pre-postcondition specifications.

Acknowledgements

Most of the concepts of the architecture definition presented are based on a language that has been developed in the IPSEN project. We therefore thank the former colleagues of the second author, namely Gregor Engels, Claus Lewerentz, Manfred Nagl and Andy Schürr. The concepts have been further developed in the Merlin project. We therefore thank our colleagues Rainer Fehling, Gerald Junkermann, Petr Kroha, Olaf Neumann, Burkhard Peuschel, Sabine Sachweh, Doris Schmedding, Jim Welsh and Stefan Wolf for the intensive and stimulating discussions on the relationships between software development environments and database systems. Moreover, we are indebted to our current and former students, namely Andreas Baumkötter, Jörg Bierbass, Heike Jeibmann, Thomas Knepper, Martin Kampmann, Thomas Leppek, Klaus-Dieter Marquardt and Hans-Dieter Schneider who contributed to the development of Groupie. The very valuable comments of Kelvin J. Ross helped to improve the presentation of the paper a lot.

References

[ADH+92] R. Adomeit, W. Deiters, B. Holtkamp, F. Schülke, and H. Weber. $K/2_R$: A Kernel for the ESF Software Factory Support Environment. In P. A. Ng, C. V. Ramamoorthy, L. C. Seifert, and R. T. Yeh, editors, *Proc. of the* 2^{nd} *Int. Conf. on Systems Integration, Morristown, N.J.*, pages 325–336, 1992.

[B$\ddot{9}$3] J. Börstler. *Programmieren-im-Großen: Sprachen, Werkzeuge, Wiederverwendung.* PhD thesis, Aachen University of Technology, 1993.

[BMO+89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 283–308. Addison Wesley, 1989.

[Cag90] M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

[CFGGR91] J. Cramer, W. Fey, M. Goedicke, and M. Große-Rhode. Towards a Formally Based Component Description Language as a Foundation for Reuse. *Structured Programming*, 12(2):91–110, 1991.

[EKS93] W. Emmerich, P. Kroha, and W. Schäfer. Object-oriented Database Management Systems for Construction of CASE Environments. In V. Mařik, J. Lažanksý, and R. R. Wagner, editors, *Database and Expert Systems Applications — Proc. of the* 4^{th} *Int. Conf. DEXA '93, Prague, Czech Republic*, volume 720 of *Lecture Notes in Computer Science*, pages 631–642. Springer, 1993.

[ELN+92] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building Integrated Software Development Environments — Part 1: Tool Specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, 1992.

[ESW93] W. Emmerich, W. Schäfer, and J. Welsh. Databases for Software Engineering Environments — The Goal has not yet been attained. In I. Sommerville and M. Paul, editors, *Software Engineering ESEC '93 — Proc. of the* 4^{th} *European*

Software Engineering Conference, Garmisch-Partenkirchen, Germany, volume 717 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 1993.

[Fel79] S.I. Feldman. Make - A Program for Maintaining Computer Programs. *Software: Practice and Experience*, pages 255–256, 1979.

[Gol85] A. Goldberg. *Smalltalk 80: The Language and its Implementation.* Addison Wesley, 1985.

[Gra78] J. N. Gray. Notes on Database Operating Systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating systems – An advanced course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3.F., pages 393–481. Springer, 1978.

[HOO89] HOOD Woorking Group. *HOOD Reference Manual.* European Space Agency, 1989.

[HP81] A. N. Habermann and D. E. Perry. System Composition and Version Control for Ada. In H. Hünke, editor, *Proc. of the Symposium on Software Engineering Environments, Lahnstein, FRG*, pages 331–344. North-Holland, 1981.

[JPSW94] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting Cooperation in Software Development through a Knowlege-based Environment. In A. C. W. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Advances in Software Process Technology*, pages 103–129. Wiley, 1994.

[KMS89] J. Kramer, J. Magee, and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):, 1989.

[Lew88] C. Lewerentz. Extended Programming in the Large in a Software Development Environment. *ACM SIGSOFT Software Engineering Notes*, 13(5):173–182, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

[MTH90] R. Millner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Per89] D. E. Perry. The Inscape Environment. In *Proc. of the* 11^{th} *Int. Conf. on Software Engineering, Pittsburgh, PA*, pages 2–12. IEEE Computer Society Press, 1989.

[SW91] M. P. Stovsky and B. W. Weide. Access Control Strategies for Coordinating Teams of Software Engineers. *International Journal for Software Engineering and Knowledge Engineering*, 1(1):57–73, 1991.