Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

Supporting software components production

P. Asirelli, D. Aquilino,
P. Inverardi

Progetto finalizzato Sistemi informatici e
Calcolo Parallelo, Sottoprogetto 6, Obiettivo AASS

Nota Interna B4-08
Aprile 1991

Supporting Software Components Production

D. Aquilino, P. Asirelli, P. Inverardi

Istituto di Elaborazione dell'Informazione CNR,
via S. Maria, n. 46, I-56100 Pisa

Abstract

In this paper we present an approach to the management of large-scale object-oriented applications based on a sharp distinction between the object-oriented language in which components are described and the way components are assembled into a specific application. In particular, we propose a database component schema which allows for a free development of components and permits to define at the database level all the policies connected to the grouping of components into a specific application.

The database schema we propose permits recording all the information about a component relevant to its use in several application contexts. At the schema level is then possible to express how components are connected in more complex structures and the definition of an application as a specific *view* of (a part of) the developed structure.

1. Introduction

Large scale development of object-oriented applications urgently demands for support during the production process. This is mainly due to the need of properly recording the relationship among developed components and the features of every component in order to make, at each stage of development, the creation, inspection and modification of the software manageable.

There are two approaches to provide the required support: the former is to build around the language a tool, for example a component database, which plays the role of repository of the produced information. An example in this direction is the Smalltalk browser which is however too limited to support large scale development. The latter consists in building in the language concepts which serve the purpose of controlling the evolution of the application. In this way the supporting facilities become actually part of the run-time support of the language, putting on the single component producer the weight of programming and controlling the interaction among the components. An example is Modular Smalltalk [Wirfs-Brock 88] in which programming in the large concepts are introduced at the language level.

The latter solution mimics the approach taken in modular conventional programming languages, e.g. Ada, Modula, etc.. We think that this is a step backward with respect to the original philosophy of the object-orientedness where the main motivation is *reusability* and not simply modularization. In fact, by reusability we mean that it should be possible to reuse the

Work partially funded by the Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Sottop.6, Ob. AASS.

same software components in several application contexts while modularity was mainly introduced to allow the change of a component in a precise application context.

In this paper, we propose a database component schema which allows for a free development of components and permits to define, at the database level, all the policies related to the grouping of components into a specific application.

In particular, the schema we propose allows the relationships among components to be recorded, to precisely describe the specific features (the interface) of a component, to attach to each component supplementary information by means of the attribute mechanism, to selectively inspect the state of the database in order to perform retrieving of components with given properties and consistency checking. Then it is possible to express, at the schema level, the requirement of an application in terms of a concept which allows the application to be defined as a specific *view* of (a part of) the developed schema.

The paper is organized as follows: section 2 presents a Simple Object-Oriented Language (SOOL) that we will use to describe our approach. Section 3, deals with the component database schema, section 4, shows how to interface SOOL with the database and how it is possible to use it in consistently support the software components production and the applications development. Section 5, concludes.

2. SOOL

In this section we present a simple object-oriented language (SOOL) which exhibits all the peculiar features of the object-oriented paradigm [Meyer 88].

SOOL is a language which provides basic constructs to build software components as abstract data type implementations, according to the object-orientedness. It is based on the definition of entity *class*; classes represent a sort of static definition of the *behavior* associated to program entities. The behavior described within a class definition refers to two different kinds of entities: instance-objects, which are the (dynamic) objects of the type described by the class, and class-object , the (static) object representing the class itself. The language supports multiple and repeated inheritance among classes, polymorfism and dynamic binding among program entities and objects; on the other side, no typing mechanism is defined in the language.

It is evident that SOOL can be seen as a subset of many other well known object-oriented languages. The motivations for another language are indeed in its kernel nature. In fact, we want to deal with an object-oriented laguage in which only constructs for the description of single entities are provided and there is no concern about developing applications. In particular, in SOOL, there are no constructs dealing with visibility rules, protection, hiding or views.

A SOOL program is a collection of static definitions of classes defined according to the following syntax.

```
< CLASS_DEF > ::=          class  < NAME >  [ < INHERITANCE > ]
                                               ( < STATUS > )
                                              { < BEHAVIOR > }
< INHERITANCE > ::=        < NAME >*
< STATUS > ::=             < ID >*
< BEHAVIOR > ::=           {[+] < MESSAGE > → < METHOD >}*

< MESSAGE > ::=            < SELECTOR > [ ( < ID >* ) ]
< METHOD > ::=             < COMMAND >*
< COMMAND > ::=            < LET_BINDING > { < COMMAND > } |
                           ↑ < EXPR > |
                           [ < SEND_EXPR > ] |
                           < PRIMITIVE_COMMAND >
< LET_BINDING > ::=        LET < ID > [= < EXPR >] {and < ID > [= < EXPR >]}*
< EXPR > ::=               < NEW_EXPR > |
                           < SEND_EXPR > |
                           <PRIMITIVE_EXPR >
< SEND_EXPR > ::=          < RECEIVER > . < SELECTOR > [(< EXPR >*)]
< NEW_EXPR > ::=           < PRODUCER > . NEW
< RECEIVER > ::=           < ID > | < PSEUDO >
< PRODUCER > ::=           < NAME > | < PSEUDO >
< PSEUDO > ::=             self | super
< SELECTOR > ::=           string | {string:}*
< NAME > ::=               string beginning with a capitol letter
< PRIMITIVE_COMMAND > ::= ...
<PRIMITIVE_EXPR > ::= ...
```

The syntax presented is sufficiently self explanatory. A class definition is identified by the classes it is based on, the inherited classes, its status layout and finally its behavior, i.e. a set of messages (selectors with parameters) attached to methods (procedures). The symbol '+' is used like in [Cox 86] and has the obvious purpose of distinguishing class methods from instance methods; the symbol '↑' is used to denote a value returned at the end of a method execution, while the LET_BINDING definition is a syntactig sugar to declare and possibly initialize local temporary variables.

As regards the treatement of variables, collected in the *status* of a class definition, we follow the Modular Smalltalk semantics [Wirfs-Brock 88]: at each variable V is implicitely associated a method to assign a value to V and a method to read its content; this way the status and the set of methods are assimilated to the concept of behavior.

We are now going to show how classes can be defined in SOOL. The two sample classes that follow are excerpted from [Cox 86]. We will use them thorougthout the paper.

```
class Graph   [Set]
              ()
              {addNode : (point)
                → ↑ self.filter: (Nodo.str:(point));

               defNode : (point)
                → LET addedNode=self.addNode:(point)
                        {↑ addedNode.defined};

               assignLabel: (nodeName)
                → LET addedNode=self.addNode:(nodeName)
                        {↑ addedNode.assignLabel};
              }

class Node    [ByteArray]
              (definedNode labelledNode references)
              {addArc : (node)
                → if (self.references == NIL) then [self.references:(Set.NEW)] fi
                   [(self.references).add:(node)]
                   ↑ self;

               assignLabel
                → if (self.labelledNode == TRUE) then. ↑ NIL
                   else LET ref = self.references
                           {ref.eachElementPerform:('assignLabel')} fi
                   ↑ self;

               defined
                → [self.definedNode:(TRUE)]
                   ↑ self;
              }
```

3. The database schema

In this section we briefly describe the schema our software component database is based on. More detailed descriptions are in [Aquilino 90, Aquilino 91].

We follow a relational oriented style with some freedom in presenting attributes, in order to make the description more manageable. Moreover when appropriate, we freely use first order logic.

The main characteristics of the model our schema is based on, are that it provides the user with the possibility of designing static schema of systems in which i) cross references among components are not direct but take place by means of *ports* thus expressing only a consistent pattern of connection; ii) the reuse of a component is possible in any place of the system where an *equivalent* functionality is required; iii) sets of *equivalent* components can be defined and it is then possible to retrieve from such a set a specific component by using a semantic oriented selection criterium (query); iv) operations on the entities of the schema are definable that, for example, make out of a system specification a a stand-alone application, i.e. configuration tools.

The schema consists of the definition of the basic entities and of the definition of the relations among them.
There are two kinds of entities by means of which software systems are described: i) the *System_Component_View* entity; ii) the *component* entity.
Every entity of type System_Component_View , from now on SCV, is connected to entities of type component by means of a relation *contain.*
Components in a contain relation with the same SCV are *equivalent,* that is each component of the set can be used in the point of the system identified by the SCV. Two entities that are contained in the same SCV are in relation *variant_of,* where the meaning of this relation is compatible with the meaning of similar relations described in the literature [WSCM 88, WSCM 89].
Thus a SCV specifies a given *view* (observation) of the components belonging to it. Therefore the relation variant_of can be simply specified as follows:

Variant_of(C1,C2,SCV):-contain(SCV, C1), contain(SCV,C2).

where C1 and C2 are components and SCV is a System_Component_View .

Besides the relation contain other relations are defined among components. These are *structural* relations, that is they define the structure of a component in terms of its sub-components. We have identified two different relations *include* and *interact.* The former specifies the structural decomposition of a component in its internal sub-components, it is a binary relation among components and objects of type SCV; the latter describes the connections among SCV, it is a binary relation. Thus, we identify the notion of a system with the concept of SCV. Therefore, components can be internally structured, via the include relation, while only SCVs can interact.

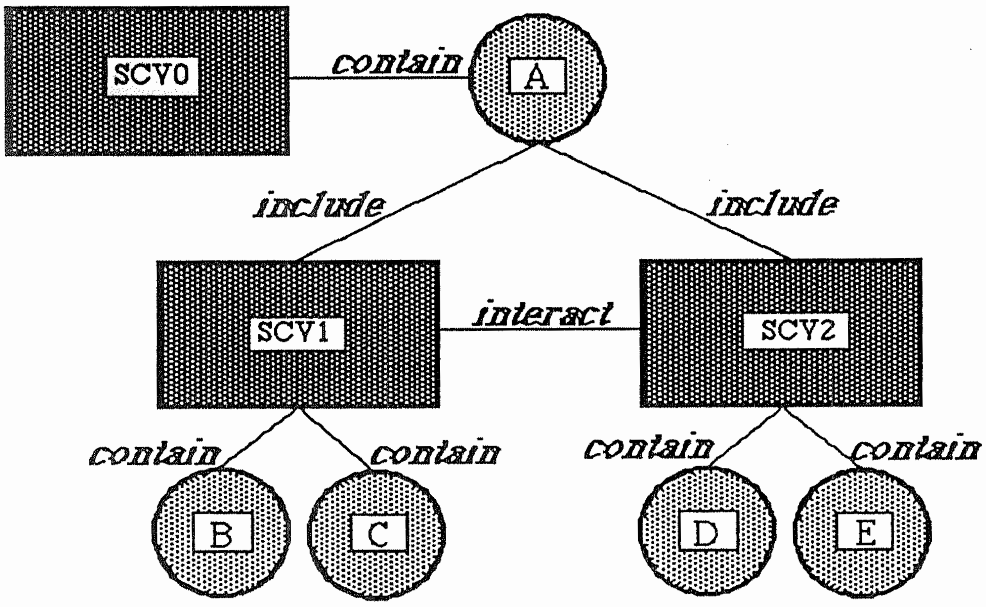


Figure 1

The situation depicted in Figure 1, describes the system identified with SCV0. Note that conventionally we will draw components as circles and SCVs as boxes. In particular, we describe the structure of the component A, the only component belonging to SCV0. It is internally decomposed into two interacting systems, represented by SCV1 and SCV2, respectively.
The functionalities represented by SCV1 can be provided by either of the atomic components B or C, the same applies to D or E with respect to SCV2. Note that structural relations make the internal topology of the component A explicit, while all the connections among components are expressed at the level of SCVs.
The data model described so far can be exemplified by means of the following Chen diagram:

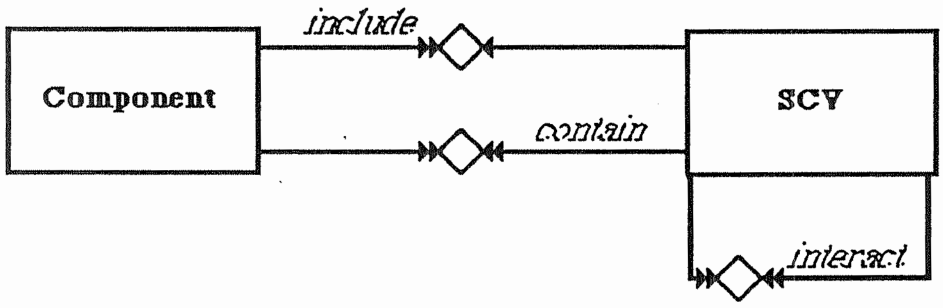


Figure 2

The diagram in Figure 2 shows that include is a N to 1 relation, while all the others are N to N relations. A SCV can appear as part of at most one component, while a component can be part of more SCVs, this again stresses the different role in the disegn of a software system the two concepts play. The former represents a specific point of the system, i.e. a specific sub-system, and thus it is unique. This means that there do not exist equivalent points of the system but in different points it is possible to use equivalent components.
So far we have presented the basic concepts of our schema, actually we want to accurately describe how interconnections among entities take place. In order to do that we have enriched components with *ports*. Ports are used to describe the interface of a component towards the external world, they are associated with their descriptions and are *attributes* of the components. A port mechanism has been introduced because of its indirectness which allows for a greater independence of a component from the context in which it is used, thus favoring the reuse of components. Connections among ports of different components are given by means of the SCV which describe the way a given component is used in that part of the system (i.e. correspoding to the given SCV). Note that a SCV may use only part of a component by connecting only a subset of its ports.

3.1 More about ports

We distinguish ports in three basic classes, depending on the kind of interaction each class is defined for:

• *Local*-ports, which model the interactions of components present at the same level of decomposition that is, among sub-components of a given component;
• *Inherited*-ports, which model the flow of information from a component to its sub-components (and viceversa);
• *Definition*-ports, which model the kind of interaction that has to be solved internally among sub-components.
Moreover, a class of objects exist, *Definitions*, which are similar to ports and can be connected with definition-port to model the interaction between a component and its sub-components.

Let us now see how to deal with the information associated to ports. Basically ports are attributes of components, and are introduced by means of the following relations:
• **R1**: PORT(PortId, In/Out, Description);
Ports are defined by means of the relation PORT, where PortId is the port identifier; In/Out specify the input or output attribute of the port; Description define the semantics associated to the port. For example, we could consider as a description the type of the functionality associated to the port.
• **R2**: DESCRIPTOR(DescriptorId, Description);
Descriptors are the counterpart of ports in a SCV. They consists of an identifier and a description as above.
• **R3**: Component(Componentname, list(PortId), list(DefinitionId), list(<Attribute, Value>));
A component is identified by a name, the set of its ports (its interface), the set of its definitions and a list of pairs, attributes-values, which characterize the component, e.g. its code, its history, etc.
• **R4**: DEFINITION(DefinitionId, In/Out, Description).
Definitions contain everything a component expresses as its own property that is, all functionalities and data that it provides locally (definitions of type Out) or those delegated to its sub-components (definitions of type In).
In the following we will depict components as in Figure 3, below:

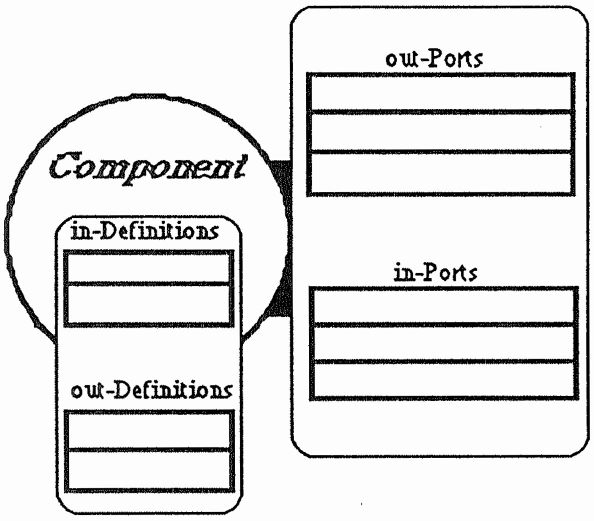


Figure 3 A Component

• R5:SCV(SCVname,list(DescriptorId), list(DescriptorId));
An SCV is characterized by a name and two lists of port descriptors which define the view that the rest of the system has of a component contained in the SCV. The former list describes the interaction with other system components, i.e. all the SCVs in the relation interact with the SCV being defined, the latter describes the structural decomposition of the part of the system identified by that SCV i.e. all the Components in relation contain and include with it. We will depict a SCV as follows:

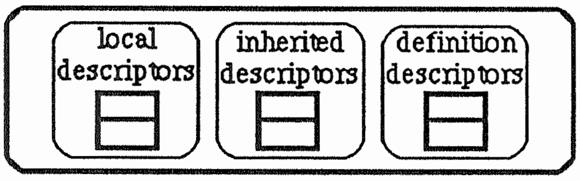


Figure 4

• R6:contain(SCVname, Compname, list(<DescritptorId,PortId>), list(<DescritptorId, PortId>));
The contain relation describes all the components that are part of the SCV. In doing this, it is explicitely declared, by means of the two connection lists, how to connect the component ports to the SCV descriptors. The former list describes which ports of the Component have to be connected to local descriptors in order to model the interactions with other components. The latter one, describes which ports have to be connected with inherited and definition descriptors of the SCV, in order to model the structural decomposition of the part of the system represented by the SCV.
•R7:include(Compname,SCVname,list(<PortId,DescriptorId>),list(<PortId,DescriptorId>));
The include relation defines the sub-systems of a component Compname; the former list specifies how to connect the defintions of a component to the definition descriptor of an SCV, while the latter declares which ports of the component have to be connected to which descriptor of the SCV (inherited ports).
• R8:interact(SCV1name,SCV2name, list(<DescriptorId,DescriptorId>));
The interact relation specifies the way local ports descriptors of two interacting sub-systems SCV1 and SCV2 are connected.
In order to specify the semantics of our relations a number of *constraints* is given, which express properties of a single relation and properties about the dependencies among different relations. In the following, we only give one of these constraints, the complete description is in [Aquilino 90].
First of all, a number of constraints exist on the structural properties of a relation, e.g. about the existence of a unique name for each relation, the cardinality of a relation, etc. More interesting for us is the constraint which specify the correct behaviour of the relation interact:
$\forall$ s1,s2 : interact(s1,s2,L) $\rightarrow \exists!$ C $\wedge$ include(C,s1,L1,L2) $\wedge$ include(C,s2,L3,L4).
where s1 and s2 are SCVs and C is a component.

This constraint states that interactions among SCVs are possible only if they are sub-systems of the same component.

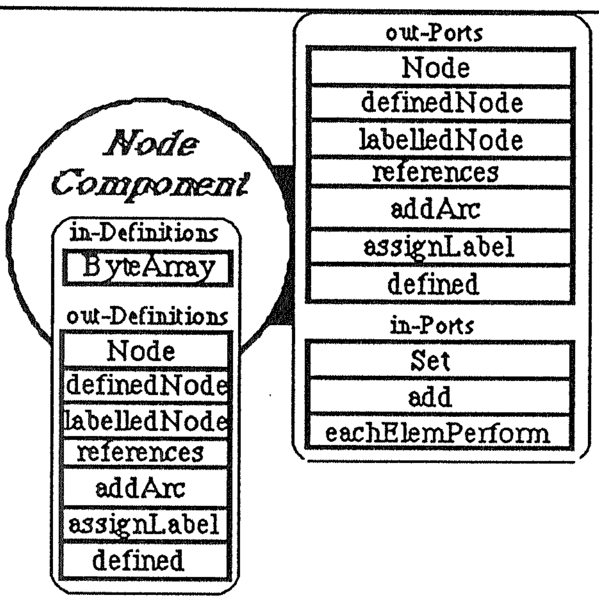
4. From SOOL to an application via the DB schema

The defined DB schema permits the description of software systems as particular *observations* of structured components. To that respect the schema identifies the *component* entity as the system modular entity, while the *SCV* is considered as the effective observation representation, which corresponds to a system. Furthermore, the schema gives a mechanisms, the port one, that, by means of relations, permits the definition of bindings among components in a way that is indirect and indipendent with respect to the information that the components represent.
From this point of view, the proposed data schema is suitable to represent systems and software components at a more general level with respect to the one generally offered by programming languages. Therefore, the information description represented by components may be independent from any precise and unique formalism which characterizes their behaviour and functionalities.
In this section we address the problem of integrating the SOOL language with the previously described data schema. The final goal is to have a database in which three degrees of freedom are provided:
i) SOOL manageable software components can be separately developed and recorded;
ii) it is possible to add structure on the components by specifying the way they can be connected, moreover version capabilities are provided ;
iii) it is possible to define application schemas, by defining observations on the components previously built, and then *configure* these schemas, by selecting appropriate versions of the components, in order to obtain stand-alone applications.
At each step all possible knowledge recorded during the previous ones, together with the classic DB-features, i.e. query mechanisms, integrity constraints, etc., are usable.

Developing SOOL manageable software components

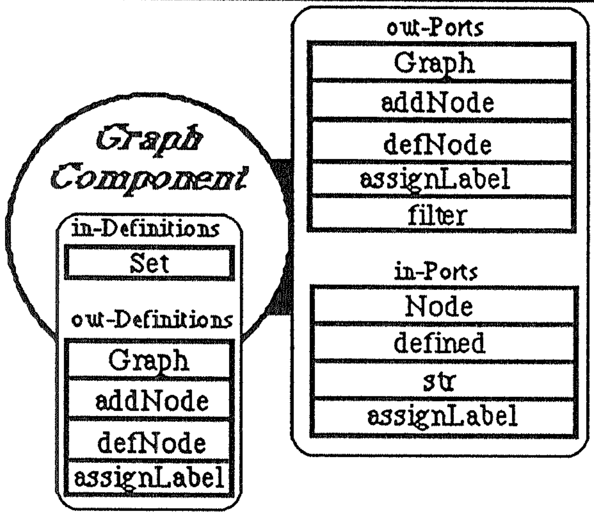
Interfacing SOOL with the data base schema means mapping the relevant subset of syntactic SOOL constructs into schema entities and relations.
The SOOL language has a single modular unit, i.e. the *class definition*, in which the functionalities it offers and the functionalities it depends on are statically defined. The counterpart of a class definition in our schema will be the *component* entity, in which its functionalities and its dependencies are recorded by means of the portlist attribute and the definitionlist attribute.
The mapping of a class definition in a component entity consists in building a set of relations in the database schema. Thus it can be thought as a mapping from the syntactic category *class_definition* to a set of relations of the schema which characterize a *component.*

Let us now show how the mapping works on the two sample classes given in section 2. The complete mapping definition (and related conventions) is given in the Appendix.



component (*Node*, (p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12), (d1,d2,d3,d4,d5), nil)
port (<*Node*, p1>, OUT, <*Node*, CLASS>)
port (<*Node*, p2>, OUT, <*addArc*, METHOD (1)>)
port (<*Node*, p3>, OUT, <*defined*, METHOD (1)>)
port (<*Node*, p4>, OUT, <*assignLabel*, METHOD (0)>)
port (<*Node*, p5>, OUT, <*definedNode*, METHOD (0)>)
port (<*Node*, p6>, OUT, <*definedNode*:, METHOD (1)>)
port (<*Node*, p7>, OUT, <*labelledNode*, METHOD (0)>)
port (<*Node*, p8>, OUT, <*labelledNode*:, METHOD (1)>)
port (<*Node*, p9>, IN, <*references*, METHOD (0)>)
port (<*Node*, p10>, IN, <*references*:, METHOD (1)>)
port (<*Node*, p11>, IN, <*str*, METHOD (1)>)
port (<*Node*, p12>, IN, <*assignLabel*, METHOD (1)>)
definition (<*Node*, d1>, OUT, <*Node*, CLASS>)
definition (<*Node*, d2>, OUT, <*addArc*, METHOD (1)>)
definition (<*Node*, d3>, OUT, <*defined*, METHOD (0)>)
definition (<*Node*, d4>, OUT, <*assignLabel*, METHOD (0)>)
definition (<*Node*, d5>, IN, <*Set*, CLASS>)

Figure 5 Translation of the class Node in the sub-schema



component (*Graph*, (p1,p2,p3,p4,p5,p6,p7,p8,p9),(d1,d2,d3,d4,d5), nil)
port (<*Graph*, p1>, OUT, <*Graph*, CLASS>)
port (<*Graph*, p2>, OUT, <*addNode*, METHOD (1)>)
port (<*Graph*, p3>, OUT, <*defNode*, METHOD (1)>)
port (<*Graph*, p4>, OUT, <*assignLabel*, METHOD (0)>)
port (<*Graph*, p5>, OUT, <*filter*, METHOD (1)>)
port (<*Graph*, p6>, IN, <*Node*, CLASS>)
port (<*Graph*, p7>, IN, <*defined*, METHOD (1)>)
port (<*Graph*, p8>, IN, <*str*, METHOD (1)>)
port (<*Graph*, p9>, IN, <*assignLabel*, METHOD (1)>)
definition (<*Graph*, d1>, OUT, <*Graph*, CLASS>)
definition (<*Graph*, d2>, OUT, <*addNode*, METHOD (1)>)
definition (<*Graph*, d3>, OUT, <*defNode*, METHOD (1)>)
definition (<*Graph*, d4>, OUT, <*assignLabel*, METHOD (0)>)
definition (<*Graph*, d5>, IN, <*Set*, CLASS>)

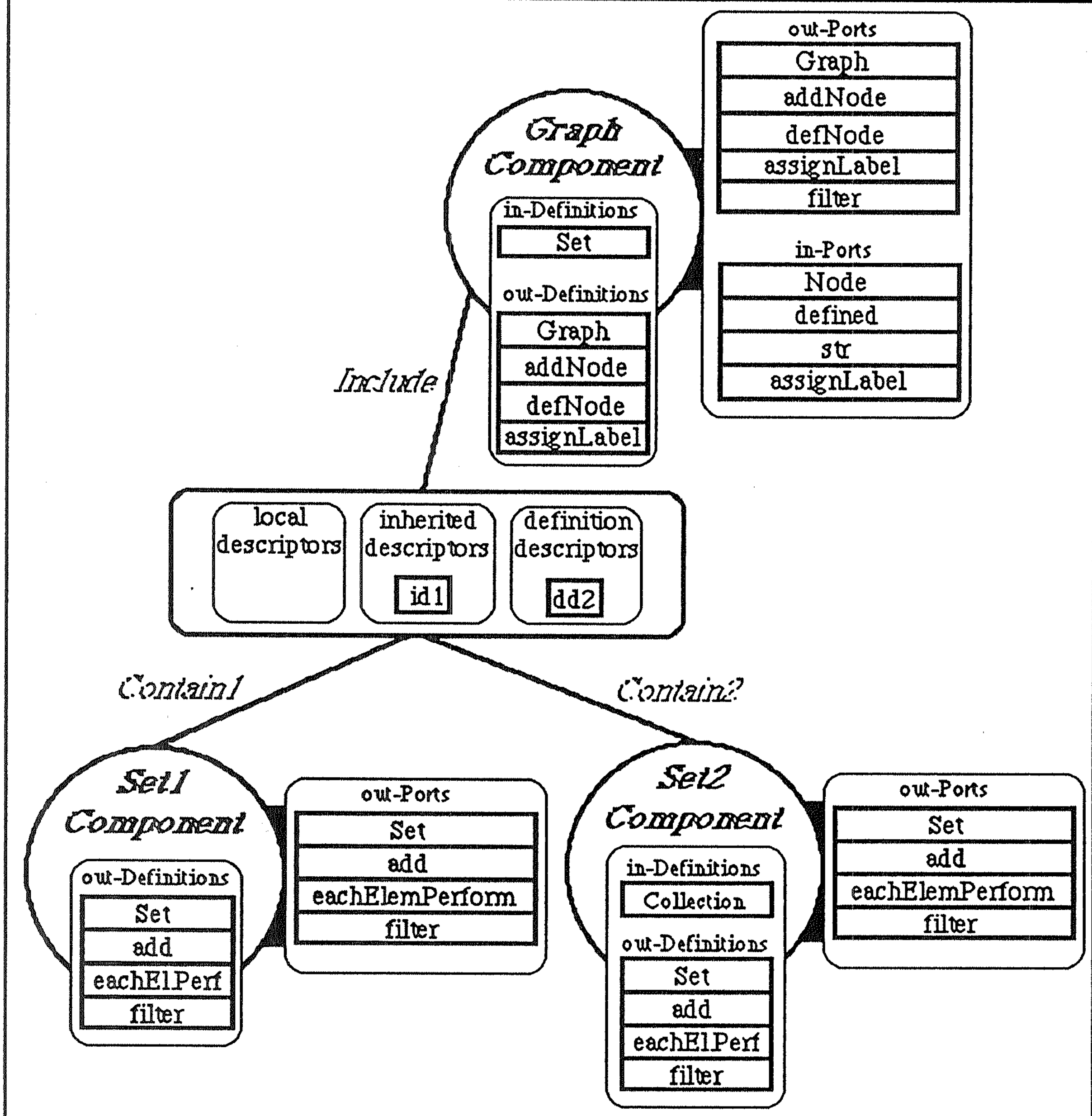
Figure 6 Translation of the class Graph in the sub-schema

Let us now analyze the Graph component. The class Graph inherited the class Set, we express this fact by asserting, via the in-definitions, that Set is an internal component of Graph. The out ports make all the functionalities made available from Graph, visible to the external world. That is, all the methods directly provided by Graph with its own code (out definitions) and all those potentially provided by the inherited classes. Note that the last, at the moment the translation takes place, are only partially known, e.g. *filter* above, and their completion depends on the particular subsystem representing *Set*s that will be connected to the graph component. In this way, at production time, no assumption is made on the way systems are built allowing for both a bottom-up and top-down development.
The same considerations apply on the in ports. In fact, at translation time we can only say which are the dependencies local to that component, but we do not already know if the inherited classes will also have unresolved local dependencies.
After inserting these sets of relations, that is the sub-schemas obtained from the translation of our software components, in the database, we are ready to deal with the problem of completing components.

Adding structure and versions capability to complete components definitions

The possibility of connecting components definitions is achieved in the DB schema by means of the SCV entity and of the include relation. Version capabilities are expressed by means of the contain relation.

In the following we exemplify the topic by showing a database context in which two components Set, with different characteristics, exist besides the Graph and Node components shown above. The goal is to show how to add structure on the Graph component in order to fill up its definitions. For semplicity we mainly make use of paintings.



SCV(SET_SYSTEM, NIL, (id1,dd2));
contain1(SET_SYSTEM, Set1_Component, (<id1,filter>), (<dd2, Set>));
contain2(SET_SYSTEM, Set2_Component, (<id1,filter>), (<dd2, Set>));
include(Graph_Component,SET_SYSTEM,(<p5,id1>),(<d5,dd2>));

Figure 7

In the Figure above, it is shown the component Graph connected with the Set_System SCV. Note that on one side the SCV plays the role of an abstraction of the Sets, more versions of Sets are actually part of the SCV, and on the other side it contains all the information needed to connect Sets with the external world. In this way SCVs represent the observation that the external world has of the components contained in it. Thus the SCV concept can be used to specify *views*, to *hide* information and to provide *versioning* facilities. Moreover, an SCV can also be used to query the database in order to retrieve all the existing components that fit the interface description it represent.

Building stand-alone applications

The last thing we have to show is how to obtain stand-alone applications. In our schema a stand-alone application is a component which has only out-Ports defining the functionalities it offers, and uniquely identifies its sub-components.

In order to do that we have to resolve all the dangling references which still exist in the schema, for example in the Graph component the in-ports dependencies are not connected yet; and operate a choice among versions of the same sub-components. The interact relation allows for expressing the interaction among components which provide complementary functionalities, for example the Graph component can interact with the Node component which provides out-functionalities in correspondence to the Graph component dependencies. The same applies to the Node component as regards its interaction with Set.

In the following picture we show a completed schema of an application based on the previously defined components can be obtained.

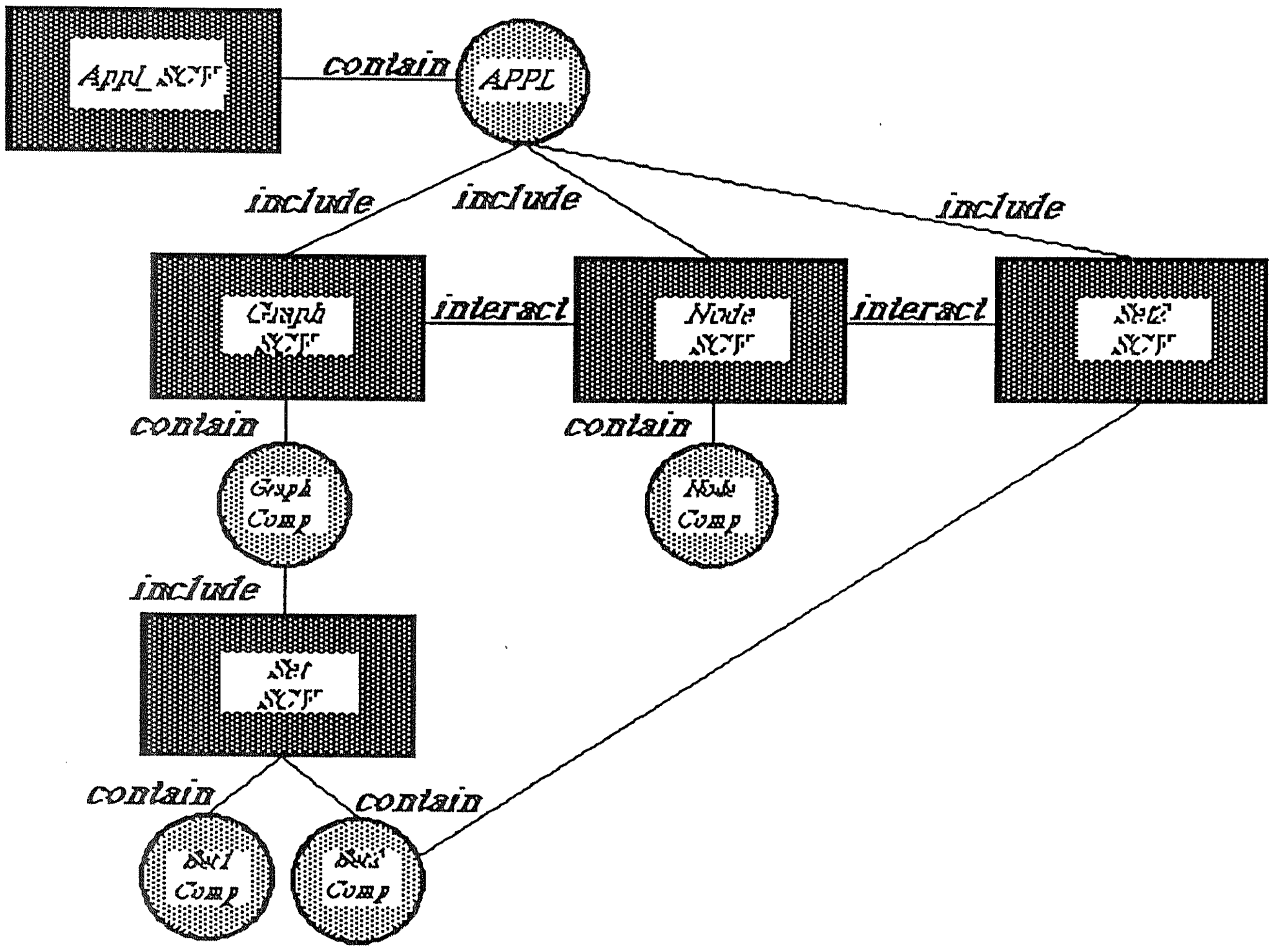


Figure 8

As it appears from Figure 8, the possibility of versioning implies the need for a final configuration step in order to retrieve a stand-alone application. That is, we have to operate a choice between the component Set1 and the component Set2, this choice may depend on several criteria, based on the attribute information associated to these components. We will not address this issue here and we refer to [Aquilino 91] for a detailed description of the configuration algorithms supported by our database schema.

5. Conclusions

In this paper we have presented a proposal for improving the development and reuse of object-oriented software components. The proposal is based on the convinction that the capability of supporting the development of software components separately from the development of a specific application, is more adequate for the objec-oriented paradigm than other approaches. Anyway, it is not just a matter of taste since the above assumption has precise implications in both the programming language structure and in the related supporting environment, and hence, on the characteristics of the produced software.

From the linguistic point of view, our approach implies that the language has to be as poorer as possible with respect to the *programming in the large* capabilities. On the other side, we require

a powerful supporting environment which permits both to record all the relevant information associated to each components and to express application requirements.
In the paper we have presented a database schema which allows for experimenting in the above perspective. We do not think that it represents an answer to *all* the open questions in terms of object-oriented supporting environments but we believe it is a step ahead in the right direction.
As concerns the experimentation of the proposal, we have provided a prototypical implementation of the schema on the deductive database management system EDBLOG [Asirelli 88] and we are trying to use it in supporting Smalltalk [Goldberg 83] application building.

References

[Aquilino 91] Aquilino, D., Malara, P., Asirelli, P., Inverardi, P., Supporting Reuse and Configuration: A Port Based SCM Model, to be presented at the *Third Int. Work. on Soft. Config. Management*, 12-14 June 1991, Trondheim, Norwey.
[Asirelli 88] Asirelli, P., Inverardi, P., Using logic databases in Software Development Environment, *Work on Prog. Lang. Impl. and Logic Prog.: Concepts and Techniques*, 16-18 May 1988, LNCS 348.
[Booch 83] Booch, G. Software Engeneering with Ada, Benjamin/Cummings Pub. Co., Menlo Park, California, 1983.
[Cox 86] Cox, B. J., Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley Pub. Co., 1986.
[Goldberg 83] Goldberg, A., Robson, D. SMALLTALK-80 The Language and its Implementation, Addison-Wesley Pub. Co., 1983.
[Meyer 88] Meyer, B. Object-oriented Software Construction , Prentice Hall International, Series in Computer Science, 1988.
[Wirfs-Brock 88] Wirfs-Brock, A., Wilkerson, B. An Overview of Modular Smalltalk, Proc. OOPSLA'88 Conference, S. Diego, CA., Sept. 25-30, 1988.
[WSCM 88] Proc. Int. Work. on Soft. Vers. and Config. Control, Grassau 1988, J.F.H. Winkler Ed., G.B. Teubner-Verlag.
[WSCM 89] Second Int. Work. on Soft. Config. Management, Princeton 1989, ACM Press.

APPENDIX

MAPPING SOOL CLASS DEFINITION INTO THE DB SCHEMA

In this appendix we precisely define how to carry on the mapping between the SOOL language syntactic constructs and the db schema. We will use a functional style.

The function *Trans* takes a class definition as input and gives a sub-schema, i.e. a set of relations, as result. *Trans* uses a number of auxiliary functions, that are listed below:
Trans : <class_def> ⟶ sub_schema, *E* : < NAME > ⟶ sub_schema ⟶ sub_schema;
V: SYN ⟶ sub_schema where SYN = SOOL_SYNTAX / < CLASS_DEF > that is it applies on all the synctactic categories except on class definitions;
UNION: sub_schema_list ⟶ sub_schema.
Let N1, ..., Nk be variables ranging on elements of the syntactic category < NAME >; In1, ..., Ini ranging on < INHERITANCE >; S1, ..., Sh ranging on < STATUS >; B1, ..., Bz ranging on < BEHAVIOR >; C1, ..., Ch ranging on < COMMAND >; Id1, ..., Idm ranging on < ID>; MS1, ..., MSn ranging on < MESSAGE >; MT1, ..., MTj ranging on < METHOD > and E1, ..., Er ranging on < EXPR >; S_S1, ..., S_Sl ranging on sub_schemas, etc.

Trans (class N, I, S, B) = *E* (N) (*UNION* (*V*(I), *V*(S) , *V*(B)))

V(I) = *V*(In1 ... Ink) = *UNION*: (*V*(In1), ..., *V*(Ink))
V(S) = *V*(S1 ... Sh) = *UNION*: (*V*(S1), ..., *V*(Sh))
V(B) = *V*(MS1⟶ MT1 ... MSn⟶ MTn) = *UNION*: (*V*(MS1⟶ MT1), ..., *V*(MSn⟶ MTn)
V(MS⟶ MT) = *UNION*: (*V*(MS), *V*(MT))
V(C) = *V*(C1 ... Ch) = *UNION*: (*V*(C1), ..., *V*(Ch))
V(L_B {C}) = *UNION*: (*V*(L_B),*V*(C))
V(L_B) = *V*(**LET** I1 = E1 **and** ... **and** In = En) =*UNION*: (*V*(E1), ..., *V*(En))
V(↑ E) = *V*(E)
V([E]) = *V*(E)
V(N_E) = *V*(P. **NEW**)
V(S_E) = *V*(R. SEL(E1, ... En) =*UNION*: (*V*(R. SEL),*V*(E1), ..., *V*(En))

V(In) = $R_{inheritance}$(In)
V(S) = R_{status}(S)
V(MS) = $R_{message}$(MS)
V(P. NEW) = R_{new}(P)
V(R.SEL) = R_{send}(R, SEL)
E(N) S_S = $R_{class\_definition}$(N, S_S)

R_{name} are relation schemas to be properly instantiated. The following conventions are used:

A port identifier is represented as a couple <component-name, IndexP>, where the component name is the name of the component to which the port belongs to; while IndexP is an index which uniquely identifies a port in the space of the port indexes of a given component. Analogous conventions holds for the definition identifiers: DefinitionID = <component-name, IndexD>. The two pre-defined functions getPindex() and getDindex() serve the purpose of generating unique indexes.
The description field of a port relation is defined as a pair <Name, type>, where Name refers to the identifier of the functionality represented by that port and type refers to the two possibilities CLASS or METHOD(Arity), that exist in SOOL.

$$R_{\text{inheritance}}(In) = \begin{cases} \text{port } (<c\_name, \text{getPindex}()>, \text{OUT}, <In, \text{CLASS}>) \\ \text{definition } (<c\_name, \text{getDindex}()>, \text{IN}, <In, \text{CLASS}>) \end{cases}$$

$$R_{\text{status}}(S) = \begin{cases} \text{port } (<c\_name, \text{getPindex}()>, \text{OUT}, <S, \text{METHOD } (0)>) \\ \text{port } (<c\_name, \text{getPindex}()>, \text{OUT}, <S, \text{METHOD } (1)>) \\ \text{definition } (<c\_name, \text{getDindex}()>, \text{OUT}, <S, \text{METHOD } (0)>) \\ \text{definition } (<c\_name, \text{getDindex}()>, \text{OUT}, <S, \text{METHOD } (1)>) \end{cases}$$

$$R_{\text{message}}(MS) = \begin{cases} \text{port } (<c\_name, \text{getPindex}()>, \text{OUT}, <MS(select.), \text{METHOD } (parameterlist)>) \\ \text{definition } (<c\_name, \text{getDindex}()>, \text{OUT}, MS(selec.), \text{METHOD } (parameterlist)) \end{cases}$$

$$R_{\text{new}}(P) = \begin{cases} \text{if } P \neq \text{self or super} \\ \text{port } (<c\_name, \text{getPindex}()>, \text{IN}, <P, \text{CLASS}>) \end{cases}$$

$$R_{\text{send}}(R.SEL) = \begin{cases} \text{if } R \neq \text{self or super} \\ \text{port } (<c\_name, \text{getPindex}()>, \text{IN}, <SEL, \text{METHOD}(parameterlist)>) \end{cases}$$

$$R_{\text{send}}(R.SEL) = \begin{cases} \text{if } R = \text{super} \\ \text{definition } (<c\_name, \text{getDindex}()>, \text{IN}, <SEL, \text{METHOD}(parameterlist)>) \end{cases}$$

$$R_{\text{send}}(R.SEL) = \begin{cases} \text{if } R = \text{self} \\ \text{port } (<c\_name, \text{getPindex}()>, \text{OUT}, <SEL, \text{METHOD}(parameterlist)>) \end{cases}$$

$$R_{\text{class\_definition}}(N, S\_S) = \{\text{component } (N, Portlist, Deflist, nil)\} \cup S\_S\ (N)$$

where *Portlist* and *Deflist* are port and definition indexes, respectively, which are defined in the sub-schema S_S . S_S (N) indicates the operation which instantiate all the relations in the sub-schema with the class name N.
Note that the *UNION* operation is a union among sets that is no duplicate elements exist.