Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

Supporting Reuse and Configuration: A Port Based SCM Model

Patrizia Asirelli, D. Aquilino, Paola Inverardi, P. Malara

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

Nota Interna B4-03
Gennaio1991

Supporting Reuse and Configuration: A port based SCM model†

D. Aquilino♪, P. Asirelli𝄞, P. Inverardi𝄞, P. Malara♪

𝄞Istituto di Elaborazione dell'Informazione CNR, Pisa

♪Dipartimento di Informatica, Università di Pisa

1. Introduction

In this paper we present a Software Configuration Model (SCM) that has been specifically developed to consistently support reuse of software components and configuration of systems. In designing the model we explicitely wanted to address the following issues:

- supporting modular software development and facilitating its modifiability ;
- supporting reusability of components and definition of reuse techniques;
- supporting definition of sets of equivalent (with respect to a certain property) components (variant/version control);
- supporting configuration of stable parts of a system;

The approach we have taken follows the experience we gained in supporting configuration management on a (logic-deductive) data base [Asirelli 87, Asirelli 88]. That is, we define a schema for a kernel configuration environment data base, (i.e. the objects, their relationships and the static constraints), and a set of operations which represent the dynamic component of the data model and among which configuration tools are defined.

The schema we propose is very general and the design focussed on the definition of a kernel data structure that maintain the information, relevant to configuration activities, produced during the various phases of the software life cycle; basic notions of our schema are those of module and variant group. A module is an elaboration unit characterized by an interface which describes the functionalities it needs in order to make available to the external world those ones it produces. Thus, an interface completely characterize a module by abstracting with respect to its internal topology. In this way, in the external context, a module is not different from an atomic structure.

---

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

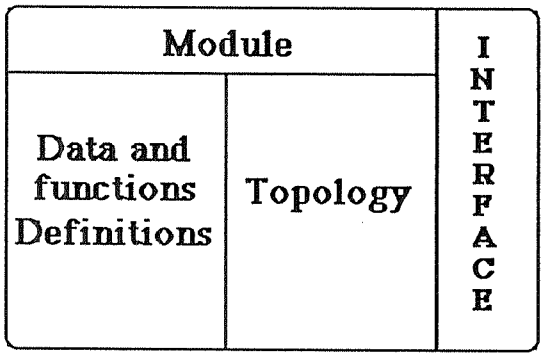


Figure 1

The concept of variant group serves multiple purposes since it plays a central role in our notions of reuse and versioning and in the way indirect connections among dependent module are expressed.

The model we present is a kernel in the sense that it is general, i.e. non concerned with a particular kind of existing programming environment or with a particular programming language. It is abstract enough to cope with the many different activities in software production, e.g. the design, implementation and code generation activities and it provides a general schema thus allowing the definition of concepts that can be further specialized to define a specific environment.

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

In the following, we describe our model: first the static component is presented following a relational style, i.e. the definition of all application object types, including their attributes, relationships and static constraints; while as the dynamic component of the data model is concerned the definition of a general schema of configuration tool, together with its properties, is given.

2. The Model

We describe the model by using a relational oriented style; the description, when appropriate, freely uses first order logic but it would be straighforward to turn it in a more conventional data definition language. The description is carried out into two steps. In the first step a rather semplified sketch of the data model is given to stress the basic concepts and only suggesting

the way components are connected. The second step enriches the model by adding the complete port mechanism, thus precisely defining the rules governing the connections among components.

2.1. A semplified schema

As we have already mentioned the objects in our model are of two kinds: i) *Variant_Group* objects; ii) *Module* objects.

Every object of type Variant_Group, from now on GV, is connected to objects of type Module by means of a relation *contain* .

Modules in a contain relation with the same GV are *equivalent*, that is each module of the class can be used in the point of the system identified by the GV. Two objects that are part of the same GV are in relation *variant_of*, where the meaning of this relation is compatible with the meaning of similar relation described in the literature [WSCM 88, WSCM 89]. In particular, our relation is similar to the one defined in [Tichy 88], which is a ternary relation where the third parameter represents the abstract property with respect to which the two objects are equivalent. In our model the third parameter is implicitely expressed by the GV the two objects belong to, thus it is the GV that specifies a given property or better, a given *view* of the objects belonging to it. Therefore the relation variant_of can be simply specified as follows:

Variant_of(M1,M2,GV):-contain(GV,M1), contain(GV,M2).

where M1 and M2 are modules and GV is a Variant_Group.

Besides the relation contain other relations are defined among objects. These are *structural* relations, that is they define the structure of an object in terms of its components. We have identified two different relations *include* and *interact*.. The former specifies the structural decomposition of a module in its internal components, it is a binary relation among objects of type module and objects of type GV; the latter describes the connections among objects, it is a binary relation among objects of type GV.

Thus we identify the notion of components of a system with the concept of GV. Therefore, a module can be internally structured, via the include relation, while only GVs can interact.

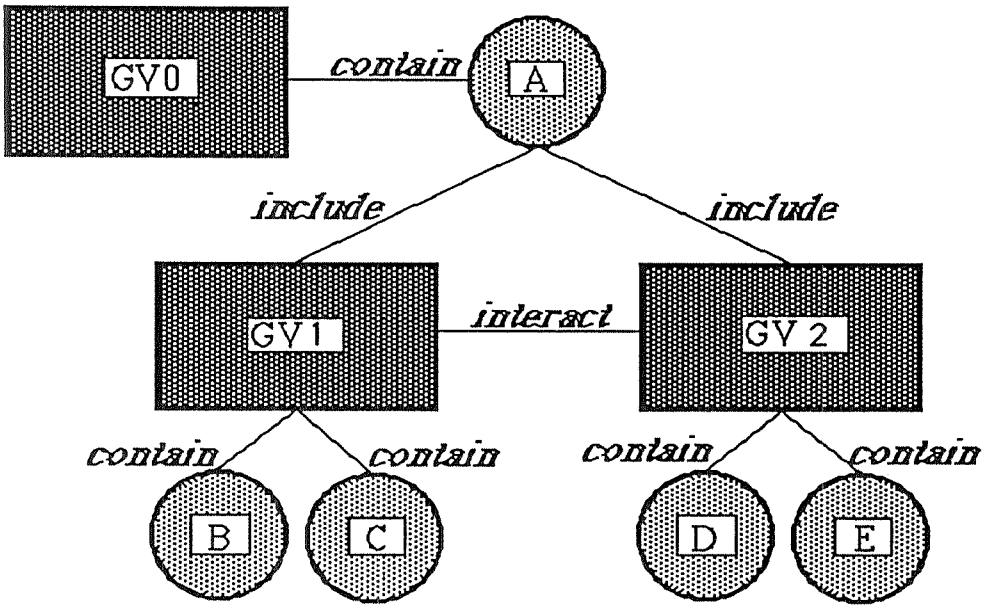


Figure 2

The situation depicted in Figure 2, describes the system identified with GV0. Note that conventionally we will draw modules as circles and GVs as boxes. In particular, we describe the structure of the module A, the only module belonging to GV0. It is internally decomposed into two interacting systems, represented by GV1 and GV2, respectively.

The functionalities represented by GV1 can be provided by either of the atomic modules B or C, the same applies to D or E with respect to GV2. Note that structural relations make the internal topology of the module A explicit, while all the connections among modules are expressed at the level of GVs.

The data model described so far can be exemplified by means of the following Chen diagram:

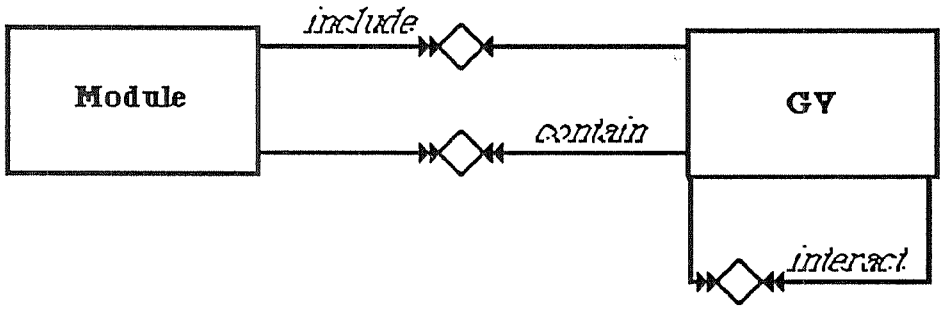


Figure 3

The diagram shows that include is a N to 1 relation, while all the others are N to N relations. A GV can appear as part of at most one module, while a module can be part of more GVs, 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 modules.

2.2 Port mechanism and components connection

In this section, we introduce a port mechanism to be used to describe the interconnections among objects within our model. A port mechanism has been introduced because of its indirectness which allows for a greater independence of a module from the context in which it is used, thus favoring the reuse of modules. Ports are used to describe the interface of a module towards the external world, they are associated with their descriptions and are *attributes* of the objects of type module. Connections among ports of different modules are given by means of the GVs which describe the way a given module is used in that part of the system (i.e. correspoding to the given GV). Note that a GV can use only part of a module by connecting only a subset of the ports of the module.

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 modules present at the same level of decomposition that is, among components of a given module;
- *Inherited*-ports, which model the flow of information from a module to its components (and viceversa);
- *Definition*-ports, which model the kind of interaction that has to be solved internally to the components of a module.

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 module and its components.

3. Relational model

In this section we summarize all the concepts introduced so far. We use a relational like style with some freedom in presenting attributes, in order to make the description more manageble. The following conceptual schema represent the initial model we have described in section 2; this will represent the starting point to be enriched with the port mechanism in order to describe the whole model.

Conceptual schema

GV (GVname);
Module (Modulename);
Interact (GVname,GVname);
Include (Modulename, GVname);
Contain (GVname, Modulename).

We now expand the above schema by adding some relations and as a consequence the above relations will be modified, too.

• R1: PORT(PortId, In/Out, Description);
Ports are defined by means of the relation PORT, with a port identifier, an IN/OUT attribute which declares the port to be input or output, and a description of 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 GV. They consists of an identifier and a description as above.

• R3: MODULE(Modulename, list(PortId), list(DefinitionId), list(<Attribute, Value>));
A module 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 module, e.g. its code, its history, etc.

• R4: DEFINITION(DefinitionId, In/Out, Description).
Definitions contain everything a module expresses as its own property, that is all the objects, functionalities and data that it produces locally (definitions of type Out) or whose production it delegates to its components (definitions of type In).

In the following we will depict modules as in Figure 4, below:

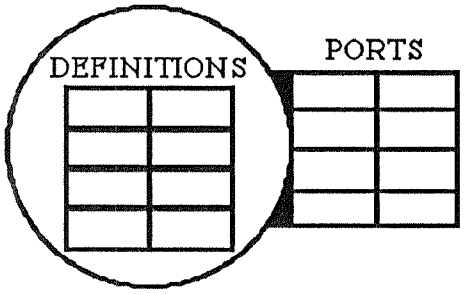


Figure 4

• R5:GV(GVname,list(DescriptorId), list(DescriptorId));
A GV is an object characterized by a name and two lists of port descriptors which define the view that the rest of the system has of a module contained in the GV. The former list describes the interaction with other system components, i.e. all the objects in the relation interact with the GV, the latter describes the structural decomposition of the part of the system identified by that GV, i.e. all the objects in relation contain and include with the GV. We will depict a GV as follows:

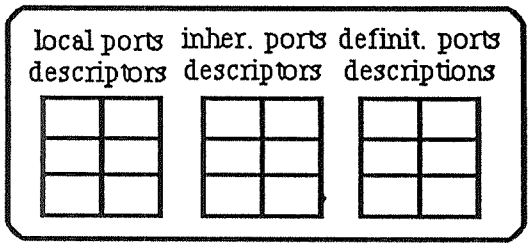


Figure 5

• R6:contain(GVname, Modulename, list(<DescritptorId,PortId>), list(<DescritptorId, PortId>));
The contain relation describes all the modules that are part of the GV. In doing this, it is explicitely declared, by means of the two connection lists, how to connect the module ports to the GV descriptors. The former list describes which ports of the module 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 GV, in order to model the structural decomposition of the part of the system represented by the GV. The picture below, illustrates an example of this situation: A GVm contains a module M. GVm uses M ports as follows: P1 as a definition port, P2 as an inherited port and P3 as a local port.

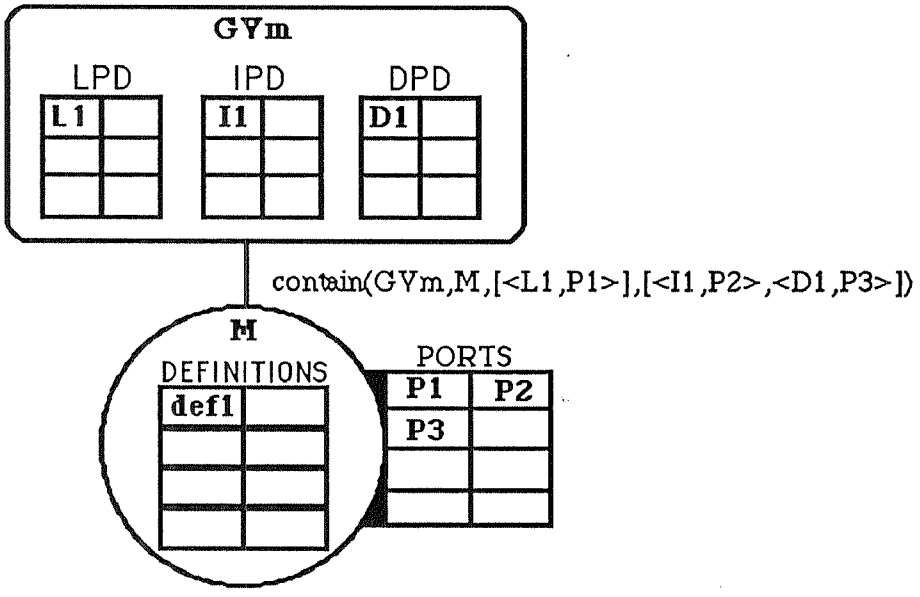


Figure 6

• R7: include(Modulename, GVname, list(<PortId,DescriptorId>), list(<PortId,DescriptorId>));
The include relation defines the components of a module Modulename; the former list specifies how to connect the defintions of a module to the definition descriptor of a GV, while the latter declares which ports of the module have to be connected to which descriptor of the GV (inherited ports). Again we exemplify this situation with a picture where a module M includes a component represented by GVm.

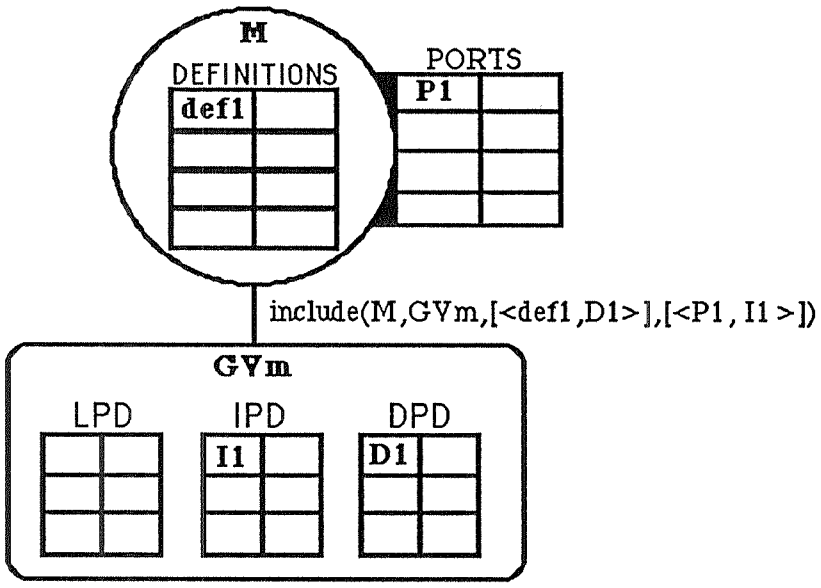


Figure 7

M includes GVm so that GVm can refer to the definition of M, def1, and it inherits port P1, that is: the functionality associated to the port P1 has now to be provided by the susbsystem GVm by using the descriptor I1.

• R8:interact(GV1name,GV2name, list(<DescriptorId,DescriptorId>));
The interact relation specifies the way local ports descriptors of two interacting subsystems GV1 and GV2 are connected.

Semantics of relations
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 some of these constraints, the complete description appears in [Aquilino 90].
First of all, a number of constraints exist on the structural properties of a relation, e.g. existence of unique name for each relation, cardinality of a relation, etc. More interesting for us are the constraints which specify the correct behaviour of a relation, for example for the relation interact we have the following two constraints:

i)$\forall$ g1,g2 : interact(g1,g2,L) $\rightarrow \exists!$ M $\wedge$ include(M,g1,L1,L2) $\wedge$ include(M,g2,L3,L4).
where g1 and g2 are GVs and M is a module.
This constraint states that interactions among GVs are possible only if they are components of the same module.
ii) $\forall$ g1,g2 : interact(g1,g2,L) $\rightarrow$ g1 $\neq$ g2.
The second constraints is an obvious one, it says that an interaction make sense only between different GVs.

The last class of constraints specifies, depending on the relation, which "type" each object involved in the relation has to have. For example, as regards the interact relation we have:
$\forall$ g1,g2,M1,M2:
interact(g1,g2,L) $\wedge$ contain (g1,M1,L11,L12) $\wedge$ contain(g2,M2,L21,L22) $\rightarrow$
($\forall$ <D1,D2> $\in$ L : $\exists$ <D1,Port1> $\in$ L11 $\wedge$ $\exists$ <D2,Port2> $\in$ L21 $\wedge$
(Port1 is_of_type IN $\wedge$ Port2 is_of_type OUT) $\vee$ (Port1 is_of_type OUT $\wedge$ Port2 is_of_type IN)

4 Operation on the data model: A configuration tool template

In this section we exemplify how to operate on our data model in order to create or acquire new knowledge out of the knowledge which is already in the model. In this framework it is possible to define operations which, on the basis of the information already present in the objects of a given relation, produce new information, e.g. the extraction of the information characterizing an interface of a given module (i.e. its ports) could be performed by a function operating on the attribute "code". In the following, we focus on a particular class of operations: a set of *configuration* tools. We will present a general schema of a configuration tool, that can be properly instantiated to obtain a specific configuration strategy.

An abstract configuration operator
Our notion of a configuration tool is that of an operation which collects modules together, according to a given criterium, to produce a new object whose semantics results from the semantics of the collected modules. This definition is intentionally vague, since we will precisely define the property of a configuration tool in the following.
Basically, a configuration activity, on a system, is defined in terms of two subtasks:
• **selection** of a specific module among all the modules contained in a GV. This selection can be performed by means of an *assertion* on the values of the module attributes. It can simply be assimilated to a boolean function depending on the verification of the assertion;.
• **composition** of the selected modules obeying the relations which define the structure of the system under configuration. Composition takes as input the set of modules previously selected and returns an atomic module which is the final output of the configuration operation.

Summarizing, from an abstract point of view a configuration tool is a function that, given a system, represented by a GV, and an assertion, that can be represented by a formula in the query language (of the host data base), gives back a module M with the following properties:
i) M is atomic;
ii) M verifies the configuration assertion; i.e. the values of its attributes satisfying the assertion (i.e. the query);
iii) M is independent from the modules it has been composed from.

The definitions of the two operators corresponding to the selection and the composition tasks have to mantain consistent the above requirements. In the following, we sketch the recursive algorithm describing the abstract configuration operator.

```
Configure(GV, Assertion: in; M: out);
begin
<select a module M0 such that contain(GV, M0) ∧ satisfies(M0, Assertion)>;
<if M0 is_atomic then return M0>;
/* note that a module is atomic if it does not have include relations defined on it*/
<if M0 ¬ is_atomic then for all GVi s.t. include(M0,GVi, L1,L2) do
                                        Configure(GVi, Assertion; Mi);
                                        cons [Mi, R] od

<M:= COMPOSE(M0, R); return M>;
end.
```

As we have already said at the beginning of this section, we want to establish a precise semantics relation between the module output of a configuration and the (structured) system, input to the configuration process. Now we can precisely state which relation we want: If GV is the input system then the resulting module M has to be in the contain relation with GV, i.e. contain (GV, M, L1, L2) must hold, where the two lists L1 and L2 can be derived from an existent relation. In particular, considering the relation contain(GV,M2,L1',L2') where M2 is the first selected module in the configuration process (the root module), then L1 and L2 are equal to L1' and L2' modulo renaming of port identifier.

4. Conclusions

In this paper we have presented a SCM model specifically designed to support software reuse and a variety of software configuration techniques.

It is worth to stress that, despite its simple structure as regards the number of relations and the kind of objects involved in, the model is very powerful. Reuse is easily obtained thanks to the port mechanism and the variant group notion, and configuration management becomes a rigorous task that can be performed incrementally and consistently with the rest of the environment. Given an input system, he result of a configuration tool is, in fact, simply an atomic module which exhibits the same properties of the root-module of the input system.

The model is going to be implemented on the Edblog system, our deductive data base management system, and it will also be equipped with a graphical interface to facilitate the user interaction. The principal application area in which we plan to use it, is the supporting environments for object-oriented languages one. We believe, in fact, that our model is well

suited to be integrated in an object-oriented environment, in which the amount of objects to be mantained is critical. In particular, the notion of variant group is going to be very important since it naturally models the notion of class as well as important is the port mechanism that allows for a straightforward modelling of objects intercommunications.

References

[Aquilino 90] Aquilino, D., Malara, P., Ambienti di configurazione e sviluppo di software riusabile: un modello di rappresentazione e sue proprietà, *Tesi di Laurea*, Dipartimento di Informatica, Dicembre 1990.

[Asirelli 87] Asirelli, P., Inverardi, P., A Logic database to support configuration management in Ada, *Proc. of the Ada-Europe Int. Conf.*, Stockholm, 26-28 May 1987, The Ada Companion Series, Camb. Univ. Press.

[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.

[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.

[Tichy 88] Tichy, W., Tools for Configuration Management, *Proc. Int. Work. on Soft. Vers. and Config. Control*, Grassau 1988, J.F.H. Winkler Ed., G.B. Teubner-Verlag.

Appendix A

The Plane-tracker example

In the following we show how to specify within our model the structure of part of an Ada system which deals with the real-time track planning of 512 planes [Booch 83]. The Ada program is composed of a package (the PLANE_TRACKER) which uses a task TRACKER; the part we model consists of the body of the package PLANE_TRACKER, of the subunit REMOVE_PLANE and of the body of the task TRACKER. In the Ada code the outline style is used to mark definitions while the style underlined marks port, program entities that will act in the model both as ports and definitions will be underlined and outlined.

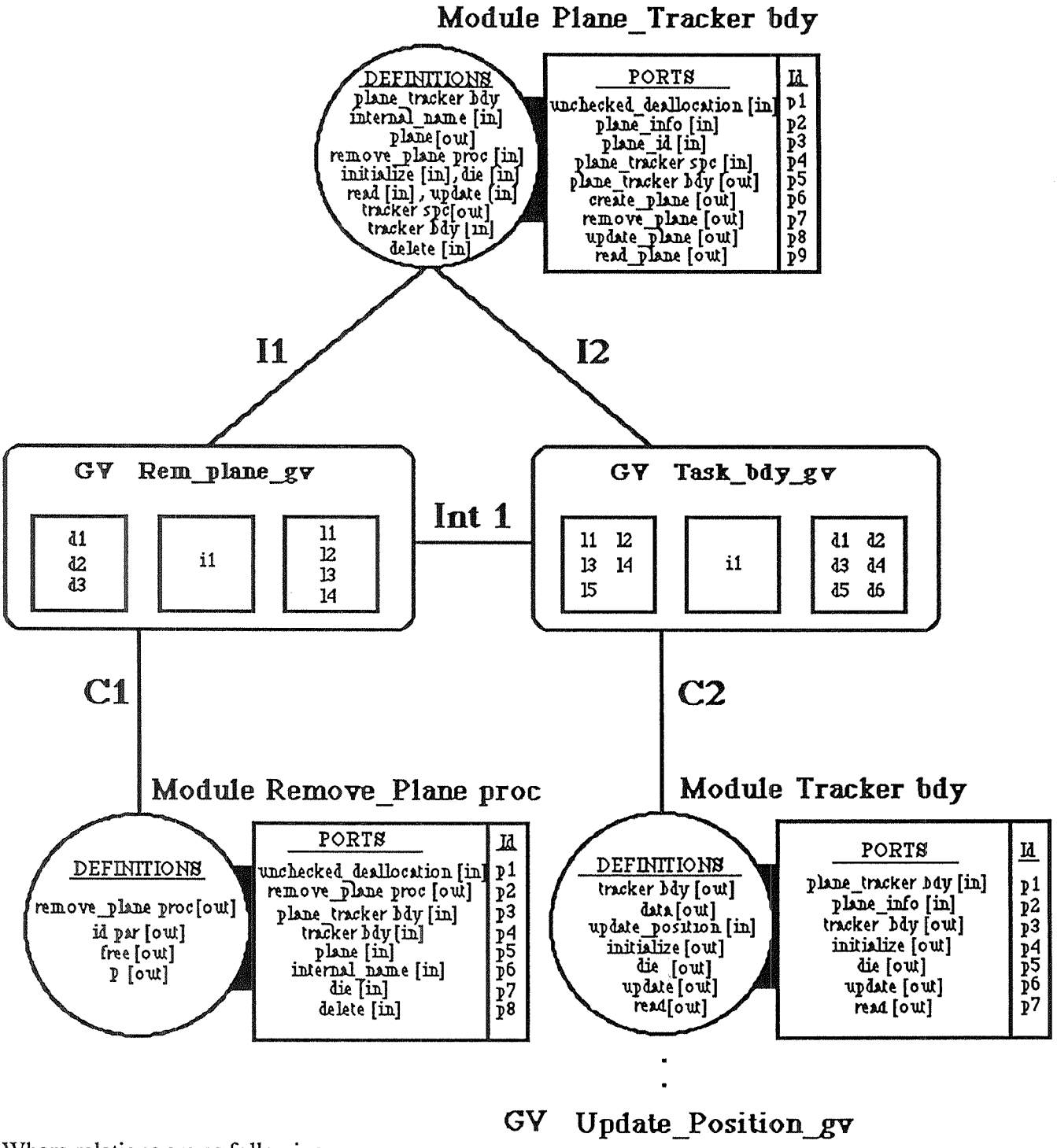
```
with UNCHECKED_DEALLOCATION;
package body PLANE_TRACKER is
   MAX_PLANES: constant := 512;
   task type TRACKER is
      entry INITIALIZE(I:PLANE_INFO);
      entry DIE;
      entry UPDATE(I:PLANE_INFO);
      entry READ(I: out PLANE_INFO);
   end TRACKER;
   type PLANE is access TRACKER;
   package ACTIVE_PLANES is
      procedure ADD(P:PLANE; ID: out PLANE_ID);
      procedure DELETE(ID:PLANE_ID);
      function INTERNAL_NAME(ID:PLANE_ID) return PLANE;
   end ACTIVE_PLANES;
   use ACTIVE_PLANES;
   task body TRACKER is separate;
   package body ACTIVE_PLANES is separate;
   procedure CREATE_PLANE(I:PLANE_INFO; ID: out PLANE_ID)
             is separate;
   procedure REMOVE_PLANE(ID: in out PLANE_ID) is separate;
   procedure UPDATE_PLANE(ID:PLANE_ID; I:PLANE_INFO) is
   begin
      INTERNAL_NAME(ID).UPDATE(I);
   end UPDATE_PLANE;
   function READ_PLANE(ID:PLANE_ID) return PLANE_INFO is
      I: PLANE_INFO;
      begin
         INTERNAL_NAME(ID).READ(I);
         return I;
   end READ_PLANE;
end PLANE_TRACKER;
```

```
separate(PLANE_TRACKER)
procedure REMOVE_PLANE(ID: in out PLANE_ID) is
   procedure FREE is
      new UNCHECKED_DEALLOCATION(TRACKER,PLANE);
   P: PLANE;
begin
   P := INTERNAL_NAME(ID);
   P.DIE;
   DELETE(ID);
   FREE(P);
end REMOVE_PLANE;
```

```
separate(PLANE_TRACKER)
task body TRACKER is
   DATA: PLANE_INFO;
   procedure UPDATE_POSITION is separate;
begin
   accept INITIALIZE(I:PLANE_INFO) do
      DATA := I;
   end INITIALIZE;
   loop
      select
         accept DIE; exit;
      or when DIE`COUNT = 0 =>
         accept UPDATE(I:PLANE_INFO) do
            DATA := I;
         end UPDATE;
      or when DIE`COUNT = 0 and UPDATE`COUNT = 0 =>
         accept READ(I: out PLANE_INFO) do
            I := DATA;
         end READ;
      or delay 1.0;
         UPDATE_POSITION;
      end select;
   end loop;
end TRACKER;

use CALENDAR;
separate(PLANE_TRACHER.TRACKER)
procedure UPDATE_POSITION is ....etc.
```

The representation, in our model, of the above system is shown in the following picture:



Where relations are as following:

I1= Include(PLANE_TRACKER bdy,Rem_plane_gv,[<remove_plane proc,d1>, <plane_tracker bdy,d2>,<plane,d3>], [<p1,i1>]).

I2= Include(PLANE_TRACKER bdy,Tracker_bdy_gv, [<plane_trackerbdy,d1>, <tracker bdy,d2>,<initialize,d3>,<die,d4>,<update,d5>,<read,d6>], [<p2,i1>]).

C1= Contain(Rem_plane_gv,REMOVE_PLANE proc,[<l1,p4>,<l2,p6>,<l3,p7>,<l4,p8>], [<d1,p2>,<d2,p3>,<d3,p5>,<i1,p1>]).

C2= Contain(Tracker_bdy_gv,TRACKERbdy,[<l1,p3>,<l2,p4>,<l3,p5>,<l4,p6>,<l5,p7>], [<d1,p1>,<d2,p3>,<d3,p4>,<d4,p5>,<d5,p6>,<d6,p7>,<i1,p2>]).

Int1= Interact(Rem_plane_gv,Tracker_bdy_gv,[<l1,l1>,<l3,l3>]).