Towards a Formally Based Component Description Language

J. Cramer†, W. Fey*, M. Goedicke‡, M. Große-Rhode* [1]

† STZ GmbH
Helenenbergweg 19
D 4600 Dortmund 50
Germany

* TU Berlin
FB 20 / FR 6-1
Franklinstraße 28-29
D 1000 Berlin 10
Germany

‡ University of Essen
FB 6/Informatik
Postfach 10 37 64
D 4300 Essen 1
Germany

Abstract The importance of a precise definition of what constitutes a **software component** and how to describe it have become critical issues in the considerations about enhancements of the software development process in general and reuse of software pieces in particular (see e.g. [Boo 87]). We consider these issues by first determining some requirements for component description languages. Based on that we discuss the Π-language as a candidate for a component description language. The Π-language was developed primarily for the specification of distributed modular systems, where the notions of data abstraction and concurrency play an equally important role. After describing the underlying concepts and the syntax of the Π-language we outline an attempt to define its formal semantics by means of algebraic module specifications.

1 The Need for Component Description Languages

In the short history of software engineering software development methods with associated languages have been developed in order to improve the software development process. Due to the increasing complexity of the problems attacked in current software development processes all these methods and languages are required to provide means for the support of large development teams with varying members. Furthermore, in order to improve the productivity of these teams and the quality of the developed software[2] they are also required to support the reuse of existing software components at any desirable level of abstraction.

For both issues a precise notion of the term software component and appropriate languages for their description are critical issues. We concentrate here on the languages for the description of software components called CDLs for short (Component Description Languages) rather than the methods. Concerning a comparable approach w.r.t. methods we refer to the concept of ViewPoints [FKG 90]. In order to judge to what degree a CDL supports team-work and reuse we determine some requirements a CDL should exhibit. Then we discuss to what degree several CDLs satisfy these requirements.

[1] Part of the work reported herein was funded by the CEC under the ESPRIT I Project PEACOCK (266), by the Ministry of Wissenschaft und Forschung of Nordrhein Westfalen and by the German Ministry for Research (BMFT) under the Eureka project ESF (Grant-No. ITS 8802)

[2] The term software not only means source or object code but all other types of documents produced during a software development process like e.g. requirements specification, design,..

1.1 Requirements for Component Description Languages

In order to support the work of large teams, a CDL must allow to divide the description of a complex software system into possibly independent components which can ultimately be conquered by individual team members. This system decomposition will not fall from heaven. A CDL therefore should enforce a clear component concept. This helps during system decomposition to identify the components. Furthermore, it assures that there is a clean semantics for the composition of the system out of these components.

Nevertheless, a decomposition will not be done as a single shot, but rather evolve during the development process. A CDL therefore must allow an incremental system decomposition. This requires particularly, that components can be described at any level of abstraction in the same way. This uniformity assures, that at any point in time of a development process a system is composed out of homogeneous but maybe different abstract components. It is worth noting, that in this context abstraction has a dual meaning. At first it refers to the complexity of the component. Secondly it refers to the level of precision of its description.

Such an incremental development approach requires, that the components can be developed independently to a large extent. A prerequisite for that is, that a component has a high internal cohesion and a low external coupling [Mye 75]. This can be achieved by a clear component concept in which components interact only via well defined interfaces. A CDL therefore should provide means for the precise definition of component interfaces. In order to avoid ambiguities and to enease reuse, *precise definition* not only refers to the syntax but also to the semantics of the interface.

The last requirement is that a CDL should provide means for a structured component description. In particular, it should allow to describe different properties of a component separately in a dedicated formalism. Furthermore, as some of these aspects might be irrelevant in one context while some new aspects become important, a CDL should provide a concept for the flexible omission or addition of the description of the various aspects of a component.

All these requirements are independent from a specific application domain and we don't claim that they are the only important ones. But they still allow to evaluate sample CDLs w.r.t. their suitability for large development teams and for reuse. In that evaluation we concentrate on languages for the design and implementation of software systems, as the Π-language is meant to support these steps in a software development process.

1.2 Evaluation of Component Description Languages

None of the requirements for a CDL is met by the imperative programming languages like e.g. Pascal or C as ultimately a system in these cases is just a single program. The only kind of component is that of a procedure so that a system is just a set of interrelated procedures. This kind of procedural abstraction is not sufficient for programming in the large. The same is analogously valid for languages allowing functional programming like LISP and the logic programming language PROLOG.

Some descendants of these languages overcome this drawback. Modula-2, Modula-3 [CDGJKN 89] and OBERON [Wir 88] as the descendants of Pascal provide the notion of modules as a component for programming in the large. Furthermore, they allow to precisely define the *syntax* of a module interface. Ada adds to these languages the possibility to describe generic modules and to specify exceptions. But they all still lack w.r.t. the other requirements.

The same holds for the descendants of C like C++ [Str 86] and Objective-C [Cox 86]. But a disadvantage of these and some other so called object oriented languages like Simula, and Smalltalk is that they rely on the inheritance- and use-relation between their objects without the possibility to define the interfaces precisely. W.r.t. the object-oriented programming language Eiffel [Mey 88] provides more support. Eiffel also offers the inheritance relation and a use-relation but allows to control the use-relation by an explicit (syntactic) interface definition. Furthermore, due to controlled export it is possible to built up configurations of classes, which can't be accessed by other ones via the use-relationship. But the problem with that is, that the inheritance relation is not controlled by that interface definition. A further aspect worth noting is, that the Eiffel environment provides tools which resolve the inheritance relation and generate a complete interface description. This complete interface description also covers some semantical information by means of assertions. A similar situation can be found for the object oriented extension of CommonLisp the so-called CLOS [Moo 89]. Here multiple inheritance etc. is supported without control via explicit interfaces. A different situation applies to modern functional programming languages which in contrast to Lisp have a static type system. ML [MHT 90], for example, has a well developed module concept although it remains on the level of syntactic interface declarations.

An analogous development has been taken place w.r.t. descendants of Prolog (c.f. [Die 89]) like M-Prolog or PROTOS-L [Bei 88], which particularly add a kind of module concept and the possibility to explicitly define their interfaces. PROTOS-L furthermore provides a type system. P4 [Hei 89] adds to that the investigation of a inheritance relation and of concurrency. Other relations for the composition of logic programs are discussed in [Roh 90].

The precise definition of such relations between components and their composition are the main contribution of the algebraic specification languages OBJ-2 [FGMO 87], ACT-TWO [Fey 88], COLD [FJ 89] and ASL [Wir 86]. There main drawback is, that they are restricted to the specification of the static properties of a component. A further disadvantage of some of them is, that they don't provide means for a semantical complete interface definition.

At the same time it is clear that each language cited has its own specialities to express certain aspects of software components quite well and naturally and falls short in other aspects. On one hand a description in a declarative style using functional or logic languages is desirable to express abstract, high level properties. On the other hand the same languages are less suited to express concurrent distributed systems with sideeffects on a detailed level. Thus it is necessary to offer a combined approach to describe software components on various levels of abstraction without loosing structural and semantic information when moving from one level to another of these levels, e.g. by adding more detail to a description.

In the following we now introduce the Π-language as such an approach. First, important concepts of the Π-language are explained using fragments of an example specification. Then an outline for a formal semantics in an algebraic setting is given.

2. The Π-Language as Component Description Language

We now discuss a candidate for a Component Description Language, the Π-language. It was developed within PEACOCK [GDS 89], [Goe 90]. The aim of PEACOCK was to develop a specification language which provides only one kind of building block for software systems and corresponding specification concepts. This is quite near to the notion of component although other desirable features to describe components are under development e.g. the notion of quantitative performance or non standard transaction concepts for distributed systems.

First a survey of basic assumptions for the Π-language is given. Then the conceptual model of the software structure is described. The chapter concludes with a survey of the language concepts of Π using an example.

2.1 Basic Assumptions

The Π language is aimed at the specification of distributed modular systems, where the notions of data abstraction and concurrency play an equally important role.The main concept of Π is an object oriented structuring of software, where objects serve as the unit to encapsulate data by operations. A software system comprises a hierarchy of such objects. Thus we follow arguments of [Sim 84] and [Res 89] and apply the general design principle divide & conquer. In describing properties of objects the developer wants the possibility to define the 'type' or 'class' of objects in order to use instances of it, objects, to configure an actual system. A 'class' of Π-objects is called a CEM (Concurrently Executable Module and its associated objects). Thus the properties of objects are specified in a CEM specification for which the Π language is the appropriate formalism.

Another important requirement for Π was proper support for incremental development.This means that development is done in small increments or steps that address different aspects of the software product. Such a working schema is in our and many other people´s opinion superior to what might be termed the traditional or "big-bang" approach.

The Π-language employs the concept of *views* to structure the specifications of object properties. Views are partial specifications each using its own representation scheme, to express the desired properties. This concept is based on the principle of *separation of concerns* and hence forms a good basis for incremental development. This means that by looking at a single view at a time we ignore the other views for a moment.

How can incremental development of component based software systems be supported? In the Π-language we separate the incremental development of a system architecture from the incremental development of each CEM specification. This approach is similar to that used in the CONIC toolkit [KMS 89] where configurations of logical (processing) nodes are specified independently of the nodes´ implementation. Thus the development using Π is characterized by a frequent switch back and forth between the development of single components – CEM specifications – and configuration of components – CEM and related object interconnections.

2.2 Conceptual model of the software structure

Objects are the building blocks of the software in its running form[3]. This means that objects provide a capsule for a piece of the entire system state and some operations, which are the only means to inspect and / or manipulate that piece of state. This concept is depicted in figure 1.

[3] In some authors´ terminology our approach is *object based* since Π does not support inheritance as Smalltalk and Eiffel do. Instead we are in favour of explicitly stating all properties used from other components including their semantics as far as it is possible.

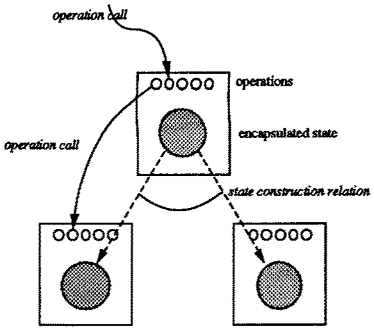


Fig. 1 The structure of objects

Objects provide operations which can be used by other objects. Since the object´s operations can manipulate the object´s state (shaded circle enclosed by rectangle in the figure above) a superior object uses the state of the underlying objects as well. Thus we say that a complex object is constructed from lower level ones in terms of their state. However, the only way objects communicate is via operation call. In the case an operation call is made to an object it controls when the requested operation is executed since it must preserve its internal consistency. This is also the way concurrency comes in naturally. Since the various operations an object offers can be invoked any time this concept includes parallel invocations of different or the same operation(s). The object, of course, can allow the execution of more than one execution request at a time if this situation is not endangering its consistency.

An entire system is represented by an object configuration. This has to be a hierarchy which means that the directed graph derived from the objects´ usage relation has to be acyclic. This also allows for multiple root-objects and shared objects as well. These are purposely built into the concept since they enable to structure distributed and especially autonomous loosely coupled cooperating systems. In such systems often peer level communication is desired. This is accomplished by sharing a communication-channel object between two or more top level objects of an object configuration.

In the following we will use the example of the patient monitoring system (see also [Kra 90]).

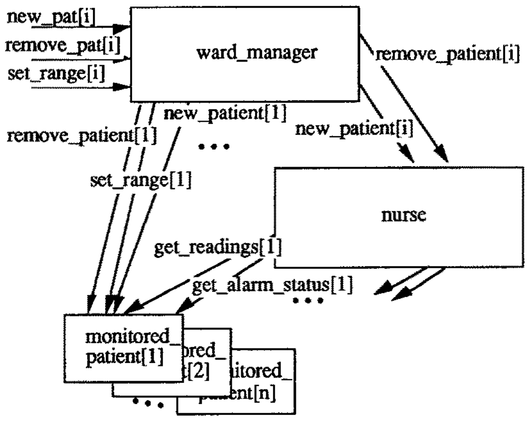


Fig. 2 Object structure of a sample patient monitoring system

Parts of this example will be used in the sequel of the paper to highlight features of the Π-language. As with the other object oriented approaches the structuring of a system with Π objects yields a natural decomposition where the artificial objects resemble the real world entities. In our examples these are the patients, the nurse and the ward manager. The relations between these objects reflect the various real world relations e.g. responsibilities and tasks of persons and equipment. The figure 2 above shows the entire object configuration of a sample patient monitoring system. Thus the system consists of 3 classes of objects. The ward_manager object creates and removes patients from the system while the nurse object scans the various patient objects for life critical conditions. Thus central to the system are the monitored_patient - objects. They have the task to check a real patient´s values, like blood pressure, in order to check whether they fall within a given patient specific safety range. If a value falls outside the safety range a bed alarm is set. Consequently monitored_patient objects offer operations like set_range for changing the safety range of values, get_readings for returning the current patient´s values and get_alarm_status for returning whether the alarm is on or not. The nurse & ward_manager -objects share each monitored_patient object. The monitored_patient objects are not primitive, but consist of a configuration of 4 objects (not including the real human being see figure 3). The central component is the monitor whose main task is to constantly read the scanner´s value, check it with the range stored in the range-component and set the bed_alarm accordingly. The scanner object is shared between the monitor and the real patient. This has to be interpreted that the sensors of the scanner are set by the patient´s physical conditions.

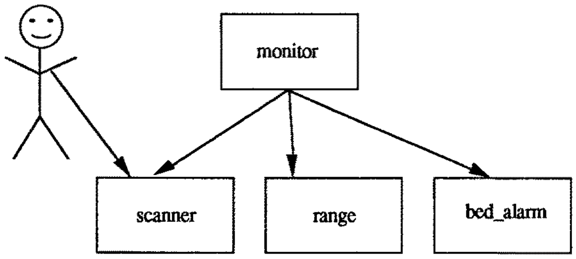


Fig. 3 Object configuration of a monitored patient

2.3 Survey of the Π-language concepts

We first introduce in this chapter the notion of CEM specification as a means to describe components. Thus a CEM specification serves as a description of all its instances -the objects. There are three views available to describe such components in isolation from other ones. These views will be introduced below. Then a description follows of how such component descriptions can be configured to form new component descriptions and eventually entire system specifications. This is reflected in the Π-language´s syntax which starts with the rule:

<component_specification> ::= <cem_specification> | <configuration_specification> (G1)

Thus a component is described by either a CEM specification or a configuration of such component specifications (see chapter 2.3.2 below).

2.3.1 CEM Views

In order to achieve the desired information hiding within the hierarchical structure of components comprising the system each CEM-specification is divided into 4 sections: export , import, common parameter and body. The task of the export and import sections is to describe the interfaces to other CEM specifications. The import is a so called formal one which means that only requirements to potential other CEM-specifications are stated. The common parameters section contains the specification of those properties which are imported and exported unchanged thus allows to see some (specifier chosen) parts of the import at the export interface. The body describes the realization of the exported properties of a CEM in terms of the imported ones. This structure of a specification is orthogonal to the structure of views i.e. in general each view has these 4 sections. Thus a CEM specification is given by

```
<cem_specification> ::=  "CEM" <cem_name>
                         ["general description" <comment>]
                         [<type_view_specification>]                (G2)
                         [<imperative_view_specification>]
                         [<concurrency_view_specification>]
                         "end CEM" <cem_name>
```

Type View

In the type view a description of the static (i.e. execution independent) properties of a CEM is given. It can also be seen as to define the 'functionality' of a CEM. This is done by specifying an abstract data type using algebraic techniques. Thus we define on a very abstract level the execution **effects** of all operations of a CEM, i.e. the effect a CEM´s operation can have in principle.

In order to achieve an abstract data type (ADT)-oriented structure of the desired software system it is important for our concept that each CEM may introduce at most one new sort with associated operations. Thus with the exception of the import and common parameter section each of the aforementioned sections in the syntax rule defines one ADT algebraically by giving sorts, operations and equations. The import and common parameter resp. are defined as a number of ADT specifications each defining properties of a CEM specification to be imported. These can be either satisfied by a single CEM specification or a configuration of these later on (see chapter 2.3.2). The body describes the CEM´s underlying ADT. The export makes only a part of this abstract data type accessible outside by exporting those operations whose execution does not unveil the internal construction of the ADT. Thus the abstract data type of the export is a sub-data type of the one defined in the body.

In our example the CEM MONITOR introduces the new sort Monitor. In the export the associated operations create, get_readings, set_range, get_range and get_alarm_status are stated. Important properties are given by some equations stating, for example, the relationship between the alarmstate and the patient´s value readings.

```
CEM MONITOR

type view specification

export
sort Monitor
   operation     create : Range -> Monitor
   operation     get_readings: Monitor -> Value
   operation     set_range : Range Monitor -> Monitor
   variables     rng1,rng2: Range
    equations
  set_range(rng1,create(rng2)) = create(rng1)
operation     get_range : Monitor -> Range
variables     m: Monitor; rng: Range
equations
  get_range(set_range(rng,m)) = rng
  get_range(create(rng)) = rng
operation     get_alarm_status : Monitor -> Boolean
variables m: Monitor;
equations
  get_alarm_status(m) =
       within_range(get_readings(m),get_range(m))
```

Fig. 4 Export section of the type view specification of CEM MONITOR

In the common parameter section the properties of e.g. Range are stated. This specifies that a sort and 3 operations are required (and exported as well) where the operation within_range provides the service to check a value against a safety range

```
import
sort Scanner
   operation     make_scanner : -> Scanner
   operation     read : Scanner -> Value
.
.
common parameters
sort Boolean
. [specification of the usual Booleans comes here]...
sort Range
   operation     within_range : Value Range -> Boolean
    variables     r: Range; v: Value
    equations
    within_range(v,r) = and(leq(min(r),v),leq(v,max(r)))
    operation     min : Range -> Value
    operation     max : Range -> Value
    variables     r: Range;
    equations
    leq(min(r),max(r)) = true
 sort Value
    operation     leq : Value Value -> Boolean
```

Fig. 5 Fragment of the import and common parameter type view specification of CEM MONITOR

In the import section properties of Scanner, Bed_alarm etc are stated. In the body section the various exported operations are defined along with some internal not exported auxiliary operations. For example the operation check_range expresses the crucial consistency condition of the CEM´s objects that the bed_alarm component reflects the relation of the current patient´s value reading to the safety range properly.

```
body
construction of sort Monitor is internal

   operation make_monitor: Range Scanner Bed_alarm ->
                                                 Monitor
   operation     create :Range -> Monitor
   variables     rng: Range
   equations
   create(rng) = make_monitor(rng, make_scanner,
                                   make_bed_alarm)

   operation     get_readings : Monitor -> Value
   variables     v:Value; rng: Range; sc: Scanner;
                 bed_al: Bed_alarm
   equations
   get_readings(make_monitor(rng,sc,bed_al)) = read(sc)

   operation     set_range: Range Monitor -> Monitor
   variables     rng1,rng2:Range; sc: Scanner;
                 bed_al: Bed_alarm
   equations
   set_range(rng1,make_monitor(rng2,sc,bed_al)) =
        check_range(make_monitor(rng1,sc,bed_al))
   operation     get_range: Monitor -> Range
   variables     rng:Range; sc: Scanner;
                 bed_al: Bed_alarm
   equations
   get_range(make_monitor(rng,sc,bed_al)) = rng
   operation     check_range: Monitor -> Monitor
   variables     rng: Range; sc: Scanner; bed_al: Bed_alarm
   equations
   check_range(make_monitor(rng,sc,bed_al)) =
        make_monitor(rng,sc,
        set_alarm(within_range(read(sc),rng),bed_al))
   operation     get_alarm_status : Monitor -> Boolean
   variables     rng: Range; sc:Scanner; bed_al: Bed_alarm;
   equations
   get_alarm_status(make_monitor(rng,sc,bed_al)) =
                                   get_alarm(bed_al)
```

Fig. 6 Body section of the type view specification of CEM MONITOR

This shall suffice as an example for the type view specification.

Imperative View

The imperative view contains specification information which relates to how a request to execute an operation – if admitted – is actually carried through including the desired side effects of executions. In principle the operations which are specified in the type view in a functional style are defined imperatively here. Therefore an operation is called a **procedure** in the imperative view. The execution aspect is specified by introducing thread(s) of control. Thus in the body for each operation it is defined which thread(s) of control to follow when the operation is executed. Also the possibility exists to fork one thread of control into more than one thread of control in parallel. Thus the imperative view describes effects on 'storage' and algorithmic concurrency.

The detailed specification of the operation in the imperative view is contained in the body section. In the remaining sections it is necessary to give the procedure headings of the operations which are exported and imported respectively. The imperative view specification for the export section and for each ADT listed in the common parameter and import section is given by a list of procedure headings. Thus e.g.

```
imperative view specification
export type Monitor
procedure create (in rng: Range ) returns Monitor
procedure get_readings(in m : Monitor ) returns Value
procedure set_range (in m : Range inout m: Monitor )
procedure get_range (in m : Monitor ) returns Range
procedure get_alarm_status (in m : Monitor ) returns Boolean
```

Fig. 7 Export section of the imperative view specification of CEM MONITOR

In_parameters may only be examined within the procedure´s body while an inout_parameter (as in procedure set_range) may be altered as well.A procedure which has no inout_parameter is returning a new object of the given type.

In the body, however, the operations are fully specified in an imperative style. The primitives of the sub-language available for describing the imperative properties of operations are assignment and procedure invocation. The usual control-flow combinators like sequence, selection and repetition are available including the parallel combinator cobegin..coend to express the potential parallel execution of more than one statement sequence. Here we give the imperative version of check_range, where an auxiliary operation select_bed_alarm is used to access the bed_alarm-component of the MONITOR-object passed as parameter. Such access operations are unnecessary in the type view since the argument matching of the equational logic used there expresses the same kind of access.

```
body
...
procedure check_range(inout m:Monitor)
declare  rng:Range;
         v: Value;
begin
   rng:= get_range(m);
   v:= get_readings(m);
   set_alarm(within(v,rng),select_bed_alarm(m));
end
procedure checking(inout m:Monitor)
begin
   while true() do
         check_range(m);
   end
end
....
```

Fig. 8 Fragment of the body section of the imperative view specification of CEM MONITOR

In order to express the constant checking of the patient´s value – a typical control-flow oriented problem not expressible in the equational logic of the type view – a new procedure for this purpose is introduced: checking as given in the above fragment of the imperative view body.

Concurrency View

The concurrency view defines the necessary ordering of operation executions to maintain the consistency of the data object by specifying the restriction of the potential full concurrency. Thus in this view the question is addressed when a request to execute an operation can be granted. We use the notation of path expressions over operation names for this purpose. [CH 74] introduced path expressions and [See 87] provided the work on modular path expressions which is also the basis for our work. In the various sections (export, common parameters, import, and body) the restrictions defined by a path expression in each of the sections play slightly different roles. The path expression in the body defines the necessary execution orderings to maintain the integrity of the object. The path expression in the export gives the information which degree of concurrency can be delivered by each object of the CEM. This can be the same as or less than the concurrency possible in the body. From the algorithms in the body the requirements to the imported objects are derived. The desired degree of concurrency to be delivered by imported objects is defined as a path expression in the import section.

The MONITOR's export and partial import section of the concurrency view is shown below and defines that the modifying operation set_range have to run exclusively while operations which merely inspect a MONITOR object (get_readings,get_range,get_alarm_status) can be executed any time unless a modifying operation is running. In the import the path expression specifies the requirements that any reasonable scanner has to be created first and then can be inspected by the read operation any time.

```
concurrency view specification
export
path expression
   create;[{      (*get_readings*) +
                  (*get_range *) +
                  (*get_alarm_status*)  }
                  | (*set_range *)]
import
type Scanner
path expression
   make_scanner;{read}
```

Fig. 9 Fragment of the concurrency view specification of CEM MONITOR

2.3.2 System Views

A system is a configuration of objects. Therefore it is necessary to discuss the important issue of connections between CEM specifications to describe system properties. Thus configurations of CEM specifications and related object configurations are considered now. For distribution and other system aspects see [PEA 88]. Syntactically a configuration specification is the 2nd alternative of syntax rule G1:

```
<configuration_specification> ::= "configuration" <configuration_name>
                                  ["general description" <comment>]
                                  <type_configuration>                    (G3)
                                  [<object_configuration>]
                                  "end configuration" <configuration_name>
```

Three aspects are defined in the configuration specification:

1) how objects are connected in principle according to their type,
2) which objects exist in the configuration,
3) which of the objects are shared.

The first aspect is defined by actualizing the formal import along the type view (nonterminal <type_configuration>). The latter two aspects are expressible by the sublanguage given by the nonterminal <object_configuration>.

For an actualization of the formal import the condition for a match between the import of an importing and the export of the imported CEM is that

- after possible renaming the signatures match (i.e. each required sort and operation is available in the export),

- the mapping defined above is a specification morphism between the two ADTs i.e. the properties stated in the requirements (import section of importing CEM) are satisfied in the ADT in the export of the imported CEM,
- the procedure headings of the imperative view match, i.e. in and inout parameters must correspond,
- the path expressions defined in the interface sections of the concurrency view have to be compatible.

If more than one CEM specification is actually imported, then first the imported CEM specifications are internally "unified" and then used to fulfil the import requirements of the importing CEM specification. This process is based in the type view on those concepts of union and composition of module specifications described later in chapter 3. The match in the imperative view implies a simple syntactic check which is sufficient, since the relevant execution invariant information about an operation is already covered by the involved type view specifications. For the concurrency view the various possible degrees of compatibility are defined in [See 87]. Below a fragment is given which shows how these CEM specification connections are established in the case of our monitored_patient example which assumes that suitable CEMs exist.

```
configuration MONITORED_PATIENT
type configuration
export
   Monitored_patient : Monitor
body
component incarnations
   Monitored_patient:      MONITOR;
   Sc:                     SCANNER_DEVICE;
   Rng:                    TUPLE_OF_MEASUREMENT;
   Bed:                    BED
   Mes:                    MEASUREMENT;
   Pat:                    REAL_PATIENT;
component connections

connection of Monitored_patient
   from Sc import
    sort Scanner is actualized by Scanner_device
    operations
        make_scanner is actualized by new_scanner
        read is actualized by return_measurement
   from Bed import
    sort Bed_alarm is actualized by Bed
    operations
        get_alarm_status is actualized by return_alarm_signal
        make_bed_alarm is actualized by make_bed
        set_alarm is actualized by set_alarm_signal
   from Rng import
    sort Range is actualized by Tuple_of_measurement
    operations
        min is actualized by first
        max is actualized by second
        within is actualized by in_interval
   from Mes import
    sort Value is actualized by Measurement
    operations
        leq is actualized by leq

connection of Pat
   from Sc import
    sort sensor is actualized by Scanner_mod
    operations
        set_sensor is actualized by receive_measurement
....
```

Fig. 10 Fragment of the type connections of the MONITORED_PATIENT configuration

In the above fragment one can see that the same segmentation into sections is maintained. The export is given by an actualized so called incarnation of CEM MONITOR. Thus the export of this actualized CEM is accessible from outside as the component MONITORED_PATIENT. In the body appropriate other component specifications are named and properly connected via specification morphism declaration (which are given by the renaming of sorts and operations).

The crucial sharing of the scanner object and the real patient has to be expressed in the object configuration section of the MONITORED_PATIENT configuration. Below we give the fragment of the object configuration specification stating exactly this.

```
object configuration
export
configuration action new_patient( in r: Rng out
                               m:Monitored_patient )
description  {sets up a new configuration of monitor, patient etc
              and shares the scanner between patient and monitor.}
configuration action remove_patient(...)
..
body
configuration action new_patient( in r: Rng out
                               m:Monitored_patient )
declare
    p: Pat;
end declare
begin
    Pat.new_real_patient(p);
    Monitored_patient.create(r,m);
    share Pat.get_scanner<p> with
        Monitored_patient.select_scanner<m>
end
    {select_scanner of Monitored_patient and get_scanner of
    Real_patient are subcomponent accessing operations not stated
    in the other views for the sake of shortness}
```

Fig. 11 Fragment of the object configuration specification of the MONITORED_PATIENT configuration

The so-called configuration actions are able to create and manipulate configurations of CEMs′ objects. In the example above the configuration action new_patient introduces a new real patient by Pat.new_real_patient and a new object of Monitored_patient by the appropriate create operation. Then the share configuration action declares the scanner component of both objects as a shared object between both newly created objects. Thus the object configuration shown in figure 3 comes into existence.

Configuration actions shall to some degree (in contrast to ordinary operations) resemble those activities in the initial configuration and later management of systems which cannot in principle or not economically implemented by some software based mechanism. In our example this is the connection of the real patient to the scanner device expressed in the configuration action above as the share action.

3. Formal Concepts for Functional Semantics of Π as CDL

In this section we describe the semantical basic for the Π-language especially for the type, type connection, imperative, and concurrency view introduced already in section 2. We use algebraic module specifications and their interconnection mechanisms (see [BEP 87], [EM 90] for the type view and type-configuration of the configuration specification, and formal operational semantics for the imperative and concurrency views. The formal basic for all that are algebraic specifications which originally where developed for data type specifications (see [GTW 76]). Therefore, let us start with a short review of algebraic specification.

3.1 Algebraic Specification

A data type in a programming language consists of data structure together with operations that create and modify this structure, where the operations are given by programming language constructs, like functions and procedures. The basic idea of an algebraic specification is to specify data types independent of any specific representation or programming language.

1. <u>Constituent Parts of Algebraic Specification</u>

An algebraic specification SPEC = (S, OP, E) consists of a signature (S, OP) with a set S of sorts and a set OP of constant and operation symbols over S, and a set E of equations or axioms over OP. Each sort represents a domain of a data structure, and each operation symbol represents an operation on that domains. More precisely an <u>operation symbol declaration</u>

$$N{:}s1 \ldots sn \rightarrow s \quad (n \geq 0),$$

consists of an operation name N, a list of argument sorts s1,...,sn and a range (or result) sort s.

So far we only have names for domains of data structures and declarations for operations but no description of what the operations should do. The third component of SPEC, the set E of equations, provides this description in an "axiomatic" or in a "constructive" way.

Algebraic Specifications SPECi=(Si,OPi,Ei) for i=1,2 can be related by a specification morphism f:SPEC1→SPEC2 which is a pair t=(f_S:S1→S2, f_{OP}:OP1→OP2 of functions such that for each N:s1...sn→s in OP1 we have f_{OP}(N):f_S(s1)...f_S(sn)→f_S(s) in OP2 - then t is called signature morphism-, and for each e in E1 the translated equation f#(e) is provable from E2 with the equational calculus.

2. SPEC-Algebras, Data Types and Semantics

Given an algebraic specification SPEC = (S, OP, E) a SPEC-algebra A is a model of the specification SPEC which consists of

- domains A_s for each $s \in S$ (defining the data structure)
- constants $N_A \in A_s$ for each N: → s in OP
- operations $N_A:A_{s1} \times ... \times A_{sn} \to As$ for each N:s1 ... sn → s in OP($n \geq 1$)

such that all equations in E are satisfied.

A SPEC-algebra A can be considered as a data type over SPEC if it is termgenerated, i.e. each $a \in A_s$ ($s \in S$) can be constructed by a term of constants and operations of A.

The initial semantics of an algebraic specification SPEC is represented by the quotient term algebra T_{SPEC} defined as quotient of the termalgebra T_{SIG} of all terms over the signature SIG = (S, OP) by the congruence generated by all the equations in E. A similar construction is possible for positive conditional equations and universal Horn axioms but not for general first order axioms. The initial semantic of SPEC is the abstract data type (ADT) defined by SPEC.

In some cases it is also useful to consider the classical or loose semantics of an algebraic specification SPEC which is given by the class of all SPEC-algebras or - as preferred by some other authors - the class of data types over SPEC.

Main constructions and results for equational algebraic specifications are existence and uniqueness (up to isomorphism) of initial and free algebras, the Birkhoff-Characterization of equational classes, the equational calculus and term rewriting, and correctness and extension criteria of specifications as given in chapters 1 to 6 of [EM 85].

3. Algebraic Specifications with Constraints

The restriction of all algebras to termgenerated algebras (i.e. data types) corresponds to the fact that we have a "termgenerating constraint". Constraints C on a specification SPEC in general are some first or higher order logical conditions for SPEC-algebras A leading to the notion of an algebraic specification with constraints, written SPECC = (SPEC, C), and all SPEC-algebras A satisfying the constraints C are called SPECC-algebras. Other interesting examples of constraints are "initial", "generating", and "free generating" constraints meaning that algebras satisfying these constraints must have certain subalgebras which are initial, or they are generated (resp. free generated) algebras over some data elements. Also first order logical axioms can be used as constraints. So, algebraic specifications with initial or loose semantics can be seen as special case of algebraic specifications with constraints.

Although most of the results for equational algebraic specifications mentioned above are no longer valid for specifications with constraints these more general specifications are most important for all kinds of applications in the software development process. See chapter 7 of [EM 90] for more details and some basic results concerning algebraic specifications with constraints.

3.2 Foundations of a Formal Semantics of the Type and Type Connection View

A functional semantics of the type view and type-configuration of Π may be given directly by a denotational semantic description or indirectly by a translation to the module specification and interconnection language ACT TWO (see [Fey 88]). Because both are based on algebraic module specifications and their interconnection mechanisms we introduce here these concepts.

3.2.1 Module Specification

The importance of decomposing large software systems into smaller components, called modules, to improve their clarity, facilitate proofs of correctness, and support reusability has been widely recognized within the programming and software engineering community. For all stages within the software development process modules resp. module specifications are seen as completely self-contained components which can be developed independently and interconnected with each other. Algebraic module specifications can be used to define the functional semantics of the type view of CEMs.

1. An algebraic module specification MOD = (EXP, IMP, PAR, BOD) consists of four algebraic specifications

MOD:

PAR	EXP
IMP	BOD

which are normally related by inclusion morphisms. The export EXP and the import IMP represent the interfaces of a module while the parameter PAR is a part common to both import and export and represents a part of the parameter of a whole modular system. These interface specifications PAR, EXP, and IMP are allowed to be algebraic specifications with constraints (see 3.1.3) in order to be able to express requirements and properties for operations and domains in the interfaces by suitable logical formalisms. Note, that - as in Π - the import interface describes only a formal import, i.e. the resources it requires rather then naming specific modules which provide those resources. The body BOD, which makes use of the resources provided by the import and offers the resources provided by the export, represents the constructive part of a module.

2. The semantics of a module specification MOD is given by the loose semantics with constraints of the interface specifications PAR, EXP, and IMP, a "free construction" F from import to body algebras which defines for each IMP-algebra A a BOD-algebra F(A) freely constructed over A, and a "behaviour construction" from body to export algebras given by restriction of the free construction to the export part.

A module specification is called (internally) correct if the free construction "protects" import algebras and together with the behaviour construction transforms import algebras satisfying the import constraints into export algebras satisfying the export constraints.

3. In the Π-language the (abstract)syntax of such module specifications is defined by

```
<type_view_specification> ::=
            "export"                 eq-SPECT1
            "import"                 eq-SPECT2
            "common parameters"      eq-SPEC
            "body"                   eq-SPECT3
```

where eq-SPEC is an equational algebraic specification (see 3.1.1) for the common parameters and eq-SPECT$_i$=(S_i,OP_i,E_i) for i=1,2,3 are equational algebraic specification torsos for the export, the import and the body respectively. These torsos are algebraic specifications but the operation symbol declarations OP_i and the equations E_i need not be over the sorts S_i and OP_i respectively. For convenience we have here abstracted from the concrete syntax for algebraic specifications which can be seen in the example of section 2.

4. Such type view specifications can be given a formal semantics by algebraic module specifications via the following semantical construction mod-unit defined on the domains eq-SPECT of algebraic specification torsos, eq-SPEC of algebraic specifications, and mod-SPEC of algebraic module specifications together with the undefined module specification mod-undef.

mod-unit : eq-SPECT x eq-SPECT x eq-SPEC x eq-SPECT → mod-SPEC

defined by

```
mod-unit (eq-SPECT$_1$, eq-SPECT$_2$, eq-SPEC, eq-SPECT$_3$) =
    **if**   eq-SPEC $\cup$ eq-SPECT$_1$ $\in$ eq-SPEC,                          (1)
          eq-SPEC $\cup$ eq-SPECT$_2$ $\in$ eq-SPEC,                          (2)
          eq-SPEC $\cup$ eq-SPECT$_2$ $\cup$ eq-SPECT$_3$$\in$ eq-SPEC,            (3)
          SIG (eq-SPECT$_1$) $\subseteq$ Sig (eq-SPECT$_2$ $\cup$ eq-SPECT$_3$)       (4), and
          equations $E_1$ are provable from equations $E \cup E_2 \cup E_3$       (5)
    **then** (eq-SPEC $\cup$ eq-SPECT$_1$, eq-SPEC $\cup$ SPECT$_2$, eq-SPEC,
          eq-SPEC $\cup$ eq-SPECT$_2$ $\cup$ eq-SPECT$_3$)
    **else** mod-undef.
```

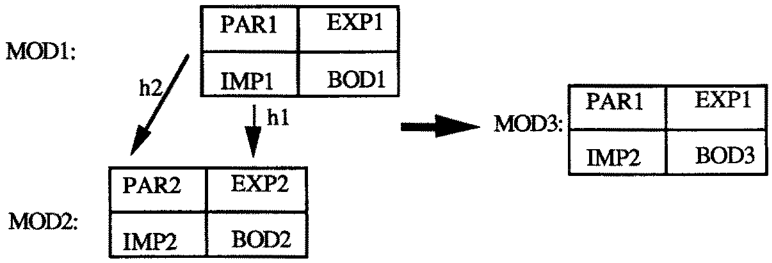
where (1), (2), and (3) are conditions for getting (complete) algebraic specification for the export, import and body respectively, condition (4) reads that all the sorts and operation symbols of the export should also be declared in the body or already required by the import, and condition (5) requires that all the equation of the export should be provable from the equations of the body, the import and the parameter.

The result of the semantical construction mod-unit (eq-SPECT$_1$, eq-SPECT$_2$, eq-SPEC, eq-SPECT$_3$) is a module specification if all the conditions (1)-(5) above are satisfied.

3.2.2 Interconnection Mechanisms

Basic interconnection mechanisms to structure modular system specifications are composition, union and actualization. Such mechanisms can be used to define the type-configuration of the configuration specification of the Π-language. Other interconnections are extension, recursion, product and iteration (see [EM 90]). But in the following we only explain the basic interconnections. Each of them gets their algebraic semantic via a module specification constructed by "flattening" the structure.

1. Composition or import actualization: the import part of module MOD1 is connected to the export part of module MOD2. The connection is established by a specification morphism h=(h1,h2) which maps sorts and operations in the import part of MOD1 to sorts and operations in the export part of MOD2 by h2 such that it is compatible with the map of the parameter part of MOD1 to the one of MOD2 by h1.



The result MOD3 of the composition is denoted by MOD1 $\circ_h$ MOD2.

For the semantics the corresponding module specification MOD3 has the same import part as MOD2, the same export and parameter parts as MOD1, and a body BOD3 which can be constructed by textual substitution of IMP1 in BOD1 by BOD2, i.e. BOD3 = BOD1 $+_{IMP1}$ BOD2.

The (abstract) syntax of such a composition of a module specification mod-SPEC1 with name mod-name1 by another module specification mod-SPEC2 with name mod-name2 via a signature morphism sigmor, -relating the import of mod-name1 to the export of mod-name2-, is defined in the ∏-language within the <type-configuration> view by

```
<component_connection_description> ::=
            "connection of" mod-name1
                      "from" mod-name2 "import" "sigmor
```

Note, that we have here abstracted from the concrete syntax of ∏ by taking terminals for signature morphisms and names of module specifications.

Such a component connection description denotes the module specification Import-Actualize(mod-SPEC1, mod-SPEC2, sigmor) got by the following semantical construction defined on the domains of module specifications mod-SPEC and signature morphisms SIGMOR

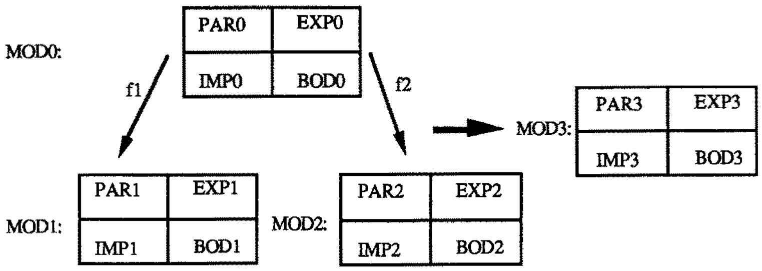
Import-Actualize : mod-SPEC x mod-SPEC x SIGMOR $\rightarrow$ mod-SPEC

defined by

```
Import-Actualize (mod-SPEC1, mod-SPEC2, sigmor) =
    if   mod-SPECi ≠ mod-undef for i=1,2,                              (1)
         PAR1 ⊆ PAR2                                                    (2)
         sigmor induces h2:IMP1 → EXP2 s. th. h2/PAR1 is inclusion     (3)
         bodies BOD2 and BOD1 without IMP1 have nothing in common      (4)
    then mod-SPEC1 ∘sigmor mod-SPEC2
    else mod-undef
```

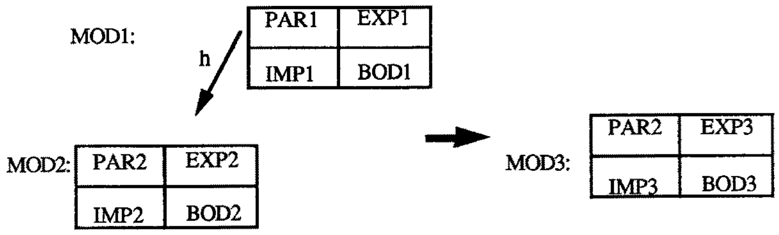
If the condition (1)-(4) above are satisfied the composition is defined and results in a module specification unequal to the undefined one mod-undef.

2. Union: Two module specifications MOD1 and MOD2 are connected via a shared submodule specification MOD0 indicated by "module specification morphisms" f_i:MOD0→MOD$_i$ for i=1,2.



For the algebraic semantics the corresponding module specification MOD3 is exactly the set theoretical union in each component, if MOD0 is equal to the intersection of MOD1 and MOD2. Otherwise all those parts in the intersection of MOD1 and MOD2, which are not in MOD0, are duplicated. In other words each component SPEC3=SPEC1+$_{\text{SPEC0}}$SPEC2 is constructed as disjoint union of SPEC1 and SPEC2 where, however, the SPEC0 part of SPEC1 and SPEC2 are "glued together".

3. Actualization: The parameter part of a module specification MOD1 is connected by a specification morphism to the export part of a module specification MOD2.



For the semantics the corresponding module specification MOD3 has the same parameter part as MOD2, but the other parts are constructed as "unions" EXP3=EXP2+$_{\text{PAR1}}$EXP1, BOD3=BOD2+$_{\text{PAR1}}$BOD1 and IMP3=IMP2+$_{\text{IMP10}}$IMP12, where however IMP1=PAR1+$_{\text{IMP10}}$IMP12 with IMP10 subspecification of IMP2 is assumed.

The last two module interconnection mechanisms - union and actualization of the parameter - are only implicitly available in Π. Therefore we gave no syntax and semantical construction here.

4. As main results for module specifications we can show that the basic interconnection mechanisms are operations on module specifications which are preserving correctness and which are compositional w.r.t. the semantics. This means that correctness of modular system specification can be deduced from correctness of its parts and its semantics can be composed from that of its components using their interconnections. Moreover, there are nice compatibility results between these operations which can be expressed by associativity, commutativity and distributivity results and allow the restructuring of modular systems (see chapters 2, 3, 4, and 8 of [EM 90]).

3.3 Towards a Formal Semantics of the Imperative, (and Concurrency) View

In the preceding sections a formal semantics for the type view and type-configurations of the Π-language has been introduced. The aim of a semantical foundation of the whole

Π-language is to define a formal semantics for each view, such that all views can be shown to be compatible; i.e. common features specified in different views have the same abstract semantics. Therefore it is necessary to have a well defined mathematical model for the semantics of a view specification.

In this section we sketch an approach for a formal operational semantics of an the type view. This is a first step towards a formal semantics of the imperative view of the Π-language where the execution of operations - specified in the type view - is specified via imperative procedures. They define threads of control, i.e. possible algorithmic concurrency. Objects are introduced as local variables to support concurrency and distributed execution of operations. The computations defined in the imperative view by algortihms must comply with the equational rules defined for the operations in the type view.

The semantical model is constructed analogously. For a given operation - specified in the type view - an algebraic graph grammar [Ehr 79] specifies its concurrent evaluation, i.e. we use a form of concurrent term rewriting as semantical model. A term is assigned to a local variable (or object) which manages its evaluation. It may invoke simple rewriting steps on the whole term, or create new objects to delegate the parallel evaluation of some of the subterms.

A graph grammar rule (see figure below) is given by three parts: a left handside which describes the subgraph which will be substituted by the right hand side. The middle part of the rule identifies a subgraph of the left hand side which remains intact by the graph substitution. Thus it is called glueing part. Boxes denote labelled nodes in the graph, while thin arrows denote the connecting edges. Bold arrows denote graph homomorphisms which define how to replace subgraphs.

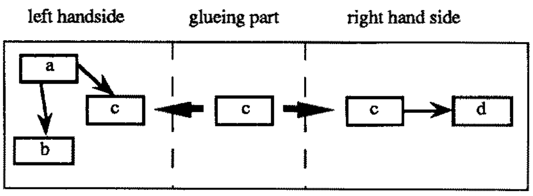


Figure 12 Sample Graph grammar rule

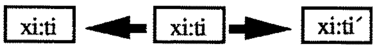
The algebraic graph grammar for the evaluation of a given term t0 is defined as follows.

START GRAPH

x0 an identifier

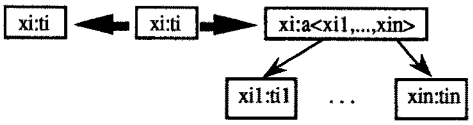
START GRAPH associates the identifier x0 with t0

REWRITE RULE



where ti → ti´ is a rewriting step induced by the equations of the type view

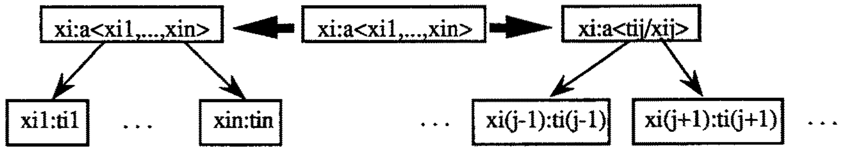
SPLIT RULE



if a<ti1/xi1,...,tin/xin> = ti; here a<ti1,...,tin> denotes a term with outermost operation symbol a and subterms ti1,...,tin; in contrast with a(ti1,...,tin) the subterms in a<ti1,...,tin> need not be direct subterms and can occur several times.

The SPLIT RULE replaces some of the subterms of the current term ti by local variables which are therefore created and associated with the corresponding subterms.

JOIN RULE



The JOIN RULE substitutes the term variable xij in the current term ti by the subterm tij which is associated with xij.

A derivation in this algebraic graph grammar has a <u>result</u>, if the last graph is a single node $\boxed{x0:t0'}$ and t0′ contains no variables (object identifiers). t0′ is then called the result of the distributed rewriting of t0.

The parallelism theorem in [EBHL 87] shows that the rewrite rule can be applied to all subterms in parallel, and that the result of this distributed rewriting is not affected by the ordering of the local evaluations.

Since distributed rewriting defines a congruence on terms it can be compared with the congruence on terms defined in the type view specification. In other words: we have mathematical models for an operational version of the type view which can be compared with the mathematical model of the imperative view based on the same formalism to show their compatibility.

The 'distributed rewrite graph grammar' is based upon simple term rewriting for algebraic specifications and thus suitable to specify operational aspects of the type view. To find appropriate rules for the specification of algorithms, however, is future work.

Having defined a formal semantics for each view in such a way, also the object-configuration aspect of configuration-specifications should be specified and integrated to obtain a formal semantical foundation of the whole language. A hint towards this aim may be found in [GM87], [Gro89], [GE90]. However, this is problem is currently investigated and its solution depends on a detailed, worked out semantical model of the other views.

For the further development of formal semantical models for the other views we may derive the following guidelines:

- A view may refine other views: The algebraic semantics of the type view defines the abstract semantics of operation, the distributed rewrite graph grammar defines how the operations are evaluated. It does not affect the semantics of the type view, but adds a further aspect.
- A view may restrict other views: The path expressions of the concurrency view may be used to exclude in certain circumstances the application of particular operations (e.g. reading from an empty list). Thus it restricts the semantics of the type view in that it discards terms or elements from the corresponding data types.
- A view may extend other views: The object-configuration aspect of configuration-specifications defines the creation and connection of objects, configuration actions define how objects communicate and the distribution of operations. The invariants of these operations are already specified in the type view; the object-

configuration aspect extends the type view taking into account possible distributions, dependency on the states of objects etc.

In each case, however, we must have a formal criterion to check whether the views fit together, refining, restricting or extending each other.

4. Conclusion

Above we introduced the concepts of the Π-language. As was shown it exhibits a number of properties required to be a CDL. Worth noting in this context is the modular approach and especially the formal import together with the configuration configuration specification capabilities. They offer flexible mechanisms to build systems from components. The possibility to state syntactic and semantic properties in the interfaces gives rise to the required precision necessary for building large and reliable software systems. The possibility to specify a system incrementally by employing the available views provides an approach to specify the components´ properties covering a wide range of abstraction levels in a structure preserving way. Moreover for each abstraction level a suitable representation scheme is offered by the respective view. Thus the entire specification is obtained by the superposition of all views. In this way the multiple representation schemes can be used as a mental vehicle to find the desired component properties. In the second part of the paper we outlined a formal foundation for the Π-language. However, much work in this direction has still to be done. The presented results for the type view will be used as a starting point and by extending the algebraic framework using the graph grammar approach the algorithmic aspect including concurrency will be defined. Thus a formally based CDL for incremental development of software components will be obtained. In addition the production of suitable tools to support the application of the Π-language is in progress.

References

[Bei 88] Beierle C. *PROTOS-L: Design and Implementation* in: Proceedings of the first workshop of the EUREKA project EU56 PROTOS, Lugano-Morcote, Switzerland, September 1988

[BEP87] E.K.Blum, H.Ehrig, F.Parisi-Presicce *Algebraic Specification of Modules and Their Basic Interconnections.* Journal of Computer and System Sciences Vol.34,Nos.2/3, NewYork-London 1987, pp. 293-339

[Boo 87] Booch, Grady *Software Components with ADA Structure, Tools, and Subsystems* Benjamin Cummings , 1987

[CDGJKN 89] Cardelli L., Donahue J., Glassman L., Jordan M., Kaslow B., Nelson G. *Modula-3 Report (revised)* Digital Systems Research Center, Technical Report , Oct 1989

[CH 74] Campbell,R.H. Habermann,A.N. *The Specification of Process Synchronization by Path Expressions* Lecture Notes in Computer Science Vol 16, pp 89-102 Springer Verlag New York 1974

[Cox 86] Cox B.J. *Object-Oriented Programming: An Evolutionary Approach* Addison-Wesley Publishing Company 1986

[Die 89] Dietrich R. *A Preprocessor Based Module System for Prolog* in: Proceedings of the TAPSOFT, Barcelona, 1989

[EBHL87] H. Ehrig , P. Boehm, U. Hummert, and M. Loewe *Distributed parallelism of graph transformation.* in 13th International Workshop on Graphtheoretic Concepts in Computer Science, pages 1-19, Springer Lecture Notes in Computer Science 314, Berlin, 1988

[EM85] H. Ehrig, B. Mahr *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer-Verlag (1985)

[EM90] H.Ehrig, B.Mahr *Fundamentals of Algebraic Specifications 2 : Modules and Constraints.* EATCS Monographs on Theoretical Computer Science, Vol. 21, Springer-Verlag (1990)

[Fey 88] W. Fey *Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language.* Diss. TU Berlin 1988; also Techn. Report No. 1988/26, TU Berlin, FB 20

[FGMO 87] K. Futatsugi, J.Goguen, J.Meseguer, K. Okada *Parameterized Programming in OBJ2* in Proc. 9th Intl. Conf. on Software Engineering , ACM 1987 pp51-60

[FJ 89] Feijs L.M.G., Jonkers H.B.M. *METEOR and Beyond: Industrializing Formal Methods* in: K.H. Bennett (ed.): Software Engineering Environments: Research and Practice John Wiley & Sons 1989

[FKG 90] Finkelstein,A. Kramer,J. Goedicke, M. *ViewPoint oriented Software Development* in Proc. 3rd Intl Workshop Software engineering & its Applications, Toulouse 1990

[GDS 89] Goedicke M., Ditt W., Schippers H. *The Π-Language Reference Manual - Version 0.1* Research Report No. 295, University of Dortmund, Department of Computer Science, January 1989

[GE90] M. Große-Rhode, H. Ehrig *Transformation of Combined Data Type and Process Specifications Using Projection Algebras.* Technical Report No. 1990/1, TU Berlin, FB 20

[GM87] J.A.Goguen, J.Meseguer *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics.* in: Research Directions in Object-Oriented Programming, ed. by B.Shriver and Peter Wegner, MIT Press, pp. 417-477, 1987

[Goe 90] Goedicke,M *Paradigms of Modular Software Developmen*" in Mitchell R.J. (Ed); Managing Complexity in Software Engineering; IEE Computing Series, Vol 17 Peter Peregrinus, Stevenage, England 1990

[Gro89] M.Große-Rhode *Parameterized Data Type and Process Specifications Using Projection Algebras.* in: Categorical Methods in Computer Science with Aspects from Topology, H.Ehrig, M.Herrlich, H.J.Kreowski G.Preuß (eds.), LNCS 393, Springer-Verlag (1989), pp. 185-197

[GTW76] J.A. Goguen, J.W. Thatcher, E.G. Wagner *An initial algebra approach to the specification, correctness and implementation of abstract data types* IBM Research Report RC 6487, 1976. Also: Current Trends in Programming Methodology IV: Data Structuring (R. Yeh, ed.), Prentice Hall (1978), 80-144

[Hei 89] Heimbigner D. *P4: A Logic Language for Process Programming* in: Proceedings of the 5th International Software Process Workshop

[KMS 89] Kramer,J. Magee,J. Sloman,M. *Constructing Distributed Systems in Conic* in IEEE Transactions on Software Engineering, Vol SE 15 No 6 June 1989

[Kra 90] Kramer, J. *Configuration Programming - A Framework for the Development of Distributable Systems* Proc. of IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro 90), Tel-Aviv, Israel, May 1990, 374-384.

[Mey 88] Meyer,B. *Object-oriented Software Constrcution* Prentice Hall Intl. Series in Computer Science, London , 1988

[MHT 90] R. Milner, M. Tofte, R. Harper. *The Definition of Standard ML* MIT Press, 1990

[Moo 89] Moon, D.A. *The CommonLisp Object -Oriented Programming Language Standard* in Kim, W. Lochovsky,F. (eds) Object Oriented Concepts, Databases, and Applications. ACM Press, Addison Wesley, New York 1989

[Mye 75] Myers G.J. *Reliable Software Through Composite Design* Van Nostrand/Reinhold 1975

[PEA 88] The Peacock Project *The Peacock Language Reference Manual* Deliverable, Brussels, March 1988

[Res 89] Resnikoff, H.L. *The Illusion of Reality* Springer Verlag,New York 1989

[Roh 90] Rohen,.M. *Semantics of composed modular logic programs in a progamming environment with integrated object inheritance mechanisms* (in german), PhD.Dissertation forthcoming University of Dortmund, Dept. of Computer Science 1990

[See 87] Seehusen,S. *Determination of Concurrency Properties in Modular Systems with Path Expressions* Dissertation, University of Dortmund, Fachbereich Informatik, 1987 (in german)

[Sim 84] Simon,H.A. *The Sciences of the Artificial* 2nd Edition, The Mit Press , 1984

[Str 86] Stroustrup B. *The C++ Programming Language* Addison-Wesley, Menlo-Park (California), 1986

[Wir 86] Wirsing M. *Structured algebraic specifications: a kernel language* Theoretical Computer Science 42, 1986

[Wir 88] Wirth N. *The Programming Language Oberon* Software Practice and Experience, No. 18, 1988