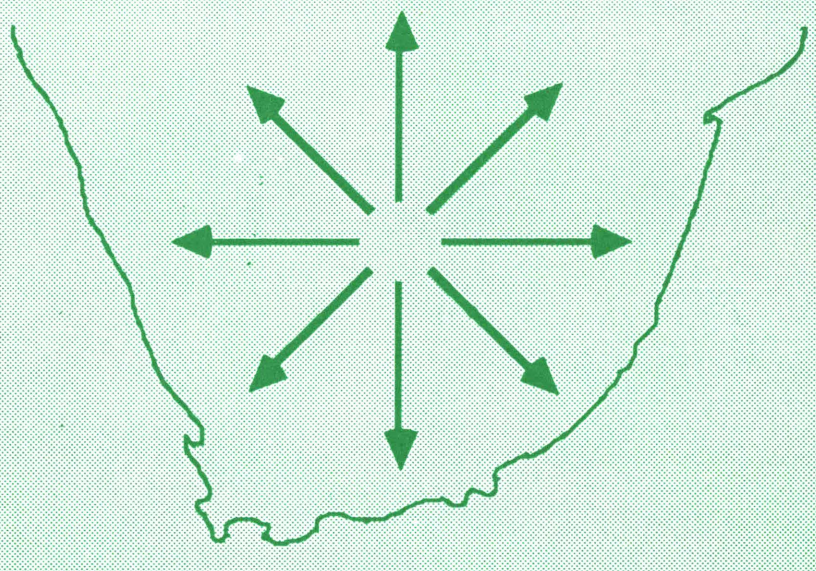
South African Computer Journal
Number 7
July 1992

Suid-Afrikaanse Rekenaar tydskrif
Nommer 7
Julie 1992

SPECIAL ISSUE



7th SA COMPUTER RESEARCH SYMPOSIUM

The South African Computer Journal

An official publication of the South African Computer Society and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif

'n Amptelike publikasie van die Suid-Afrikaanse Rekenaarvereniging en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Subscriptions

	Annual	Single copy
Southern Africa:	R45,00	R15,00
Elsewhere	$45,00	$15,00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

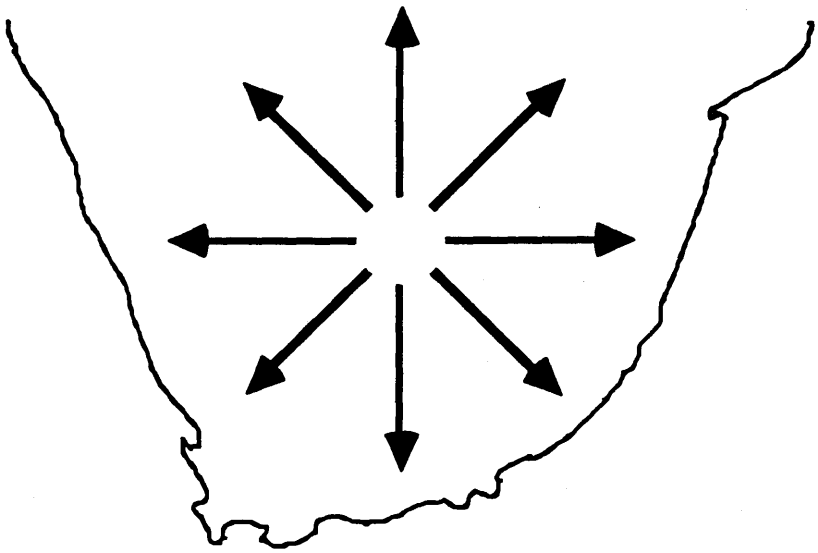
7th Southern African Computer Research Symposium

Karos Indaba Hotel, Johannesburg

1 July 1992

PROCEEDINGS

Guest Editor: Judy M Bishop



PERSETEL

Organised by the SA Institute of Computer Scientists
in association with the Computer Society of SA

Sponsored by Persetel and the FRD

SPECIAL ISSUE - 7th SA COMPUTER SYMPOSIUM

PREFACE

When the first SA Computer Symposium was held at the CSIR in the early eighties, it was unique. There was no other forum at the time for the presentation of research in computer science. In the intervening decade, conferences, symposia and workshops have sprung up in response to demand, and now there are several successful ventures, some into their third or fourth iteration. Each of these addresses a specific topic - for example, hypermedia, expert systems, parallel processing or formal aspects of computing - and attracts a specialised audience, well versed in the subject and eager to learn more. For the main part, the proceedings are informal, and certainly not archival.

SACRS, though, is still unique, in that it deliberately covers a broad spectrum of research in computing, and in addition, seeks to provide a lasting record of the proceedings. To achieve the second aim, we negotiated with the SA Institute of Computer Scientists for the proceedings to form a special issue of the SA Computer Journal, and the copy you have in front of you is the result. The collaboration between the symposium committee and the journal's editorial board placed high standards on the refereeing and final presentation of the papers, to the symposium's benefit, while we were still able to maintain a fresh, audience-oriented approach to the selection of papers.

This is SACJ's first such special issue, and the largest issue (at 145 pages) to date. We hope that it is only the beginning of future such collaborations.

In all 29 papers were received, all were refereed twice, and 19 were chosen for presentation by the programme committee. All the papers were thoroughly revised by the authors on the basis of the referee's comments, and the committee's suggestions aimed at making the material more accessible to a broadly-based audience. Papers had to be new, and not to have been presented elsewhere, a requirement that is still unusual within the SA conference round.

A third goal of SACRS has been to invite keynote speakers, usually from overseas. This year, we are fortunate to present Dr Vinton Cerf, the father of the Internet and a world-renown expert on computer networks. Although his paper is not available for this special issue, it will appear later in SACJ. Through the good offices of Professor Chris Brink of UCT, we also have three other speakers from Germany, Canada and the US adding interest to the event, and two of their papers appear in this issue.

The programme committee originally devised a theme for the symposium - "Computing in the New South Africa". We received several queries as to the meaning of this theme, but unfortunately few papers that addressed it directly. One prospective author went as far as to enquire whether computer research would survive in the new South Africa. Another felt that his work was definitely not in the theme, as it was genuine, old world, basic, theoretical science! Neverthless, there are two papers that consider one of South Africa's key issues, that of language. Others look at the success we have achieved in applying technology to mining, and the future of low-cost operating systems. In all, the mix of papers represents a balance between the theoretical and the practical, the past and the future, all firmly based in the computing of the present.

Organising the symposium has involved the hard work of several people, and I would like to thank in particular

- Derrick Kourie, my co-organiser, and the editor of SACJ for his invaluable advice and hard work throughout the planning and implementation stages;
- Riël Smit, the production editor, for attaining such a high standard in such a short time for so many papers;
- Gerrit Prinsloo and the staff at the CSSA for their efficeint and quite delightfully unfussy organisation;
- Persetel for their very generous sponsorship of R25000, and Tim Schumann for taking a genuine interest in our events;
- the Foundation for Research Development for sponsoring Vint Cerf's visit;
- and finally the Department of Computer Science of the University of Pretoria for providing the ideal working conditions for undertaking ventures of this kind, and especially Roelf van den Heever for his unfailing encouragement and support.

Judy M Bishop
Organising Chairman, SACRS 1992
Guest Editor, SACJ Special Issue

Referees

The journal draws on a wide range of referees. The following were involved in the refereeing of the papers selected for this special issue. Their role in certifying the papers and their contribution to enhancing the quality of papers is sincerely appreciated.

John Barrow	UNISA
Ronnie Becker	University of Cape Town
Danie Behr	University of Pretoria
Sonia Berman	University of Cape Town
Liesbeth Botha	University of Pretoria
Theo Bothma	University of Pretoria
Chris Brink	University of Cape Town
Peter Clayton	Rhodes University
Ian Cloete	Stellenbosch University
Antony Cooper	CSIR
Elise Ehlers	Rand Afrikaanse University
Quintin Gee	DALSIG
Andy Gravell	University of Southampton
Wendy Hall	University of Southampton
David Harel	The Weizmann Institute of Science
Scott Hazelhurst	University of the Witwatersrand
Derrick Kourie	University of Pretoria
Willem Labuschagne	UNISA
Doug Laing	ISM
Dave Levy	University of Natal
Graham McLeod	University of Cape Town
Hans Messerschmidt	University of the Orange Free State
Deon Oosthuizen	University of Pretoria
Ben Oosthuysen	University of Pretoria
Neil Pendock	University of the Witwatersrand
D Petkov	University of Natal
Martin Rennhackkamp	Stellenbosch University
Cees Roon	University of Pretoria
Jan Roos	University of Pretoria
John Shochot	University of the Witwatersrand
Morris Sloman	Imperial College, London
Riel Smit	University of Cape Town
Pat Terry	Rhodes University
Walter van den Heever	University of Pretoria
Lynn van der Vegte	University of Pretoria
Herman Venter	University of Fort Hare
Herna Viktor	Stellenbosch University

Semantic Constructs for a Persistent Programming Language

Shane B Sparg Sonia Berman
Database Laboratory, Department of Computer Science, University of Cape Town, Rondebosch, 7700

Abstract

Existing persistent languages cannot express the structure and semantics of the data they work on precisely and simply. As a result programs are more difficult to write and the integrity of the data is compromised. This paper presents a persistent programming language which includes a constraint definition and checking system. The constraint definition aspects of the language PERCI *are explained and the implementation of the constraint handling system is outlined.* PERCI *is compared with existing languages and the advantages of the new language are described.*
Keywords: *Databases, Programming languages, Semantic data models, Persistence.*
Computing Review Categories: *D.3.3, H.2.0, H.2.1, H.2.3*

Received March 1992, Accepted April 1992

1 Introduction

Until recently, research into programming languages and databases has been pursued independently. The interaction between programming languages and database systems has been mainly restricted to the calling of database packages from within programs. Persistent programming languages aim to integrate these two areas to facilitate the creation of programs which can manipulate transient and persistent data in a uniform way.

There are, however, some areas where the existing persistent languages need to be extended. In particular they lack adequate features for the accurate modeling of real life. The structure and semantics of the data cannot be expressed precisely and simply. This makes programs more difficult to write and understand, as well as compromising the integrity of the data.

When a program operates on data which is not accurately described, the data may take on an invalid state or undergo an invalid transition. Traditional type systems cannot adequately express these invalid states. When the type system cannot cope, the programmer is forced to make up for this deficiency by including checks for correctness in the program. This is undesirable because:

- The programmer has to remember to include these checks at all the appropriate points in the program, making the coding process more complicated and less natural.
- Programs become more difficult to comprehend because of the integrity checks inserted into the code.
- The procedure is prone to error. The programmer may forget to include some checks, especially in large programs or when modifying programs written by someone else.
- It is inefficient. The integrity checks can only be executed at run time since they are part of the program. There is no way to statically check any of the constraints.
- It is unsuitable for a persistent programming language since data may easily outlive a program, and there is no guarantee that another program will enforce all the required integrity conditions.

PERCI was developed by the authors at the University of Cape Town. It is a persistent programming language which includes a constraint definition and checking system. The objective of PERCI is to provide a mechanism for describing the constraints on data, and a subsystem to monitor these constraints and ensure that they are not violated. The data definition language of PERCI is based on a semantic data model similar to SDM [6]. Existing persistent languages such as PS-Algol [12], Napier [8] and DBPL [13] do not provide semantic modeling features and constraint enforcement. Taxis [9] and Galileo [1] do provide semantic constructs, but Taxis does not have any means of enforcing constraints. Galileo provides only static integrity constraints which are checked when objects are created, but not on updates.

The major contribution of PERCI is the provision of a simple and consistent constraint handling system in a persistent language which allows the programmer to specify the stage at which constraints are to be checked. This paper focuses on the constraint specification parts of the language and their implementation.

Section 2 introduces the basic concepts of persistence, semantic data modeling and integrity constraints. Section 3 briefly describes PERCI's syntax and how the constraints on data are specified. Section 4 explains how the constraint handling system is implemented. In conclusion, PERCI is compared with the existing persistent languages, and the benefits of the new language are indicated. A small example program is outlined in Appendix A.

2 Background

Persistent languages [2] process database information using the same data types and operations as for transient data; there is no need for a mapping from one representation to another.

A data model consists of rules for defining the logical structure of data and the operations available. A data model should be simple, provide expressive power, and not be dependent on implementation details. A major weakness of the relational model is a lack of semantic expressive power. Concepts must be fragmented and distorted to fit the model, thus losing their simplicity and naturalness [7].

The semantic data model is based on abstract entities (or objects) rather than tuples. There is also the provision for structural constraints and relationships between objects.

Semantic Data Models

In a semantic data model a database is viewed as a collection of *entities* which correspond to the actual objects in the application environment [6]. These entities are arranged into *classes*, which are meaningful collections of entities. Classes are in general not independent, but are logically related by means of *interclass connections*. Entities and classes have *attributes* which describe their characteristics and relate them to other entities. The value of an attribute may be derived from other values in the database. Attributes may be single-valued or multi-valued.

SDM [11] is a high-level semantics-based database model based on classification, aggregation and generalization. *Classification* refers to the member-class relationship. It is a form of abstraction in which a collection of objects is considered a higher level object class. This is essentially an instance-of relationship. *Aggregation* allows the relationship between entities to be treated as an entity itself at a higher level [14]. *Generalization* allows the differences between similar objects to be ignored to form a higher level type in which the similarities are emphasised. This results in a type hierarchy, with the more generalized types at the top and more specialized ones at the bottom.

Data types and attribute options are used to specify constraints in SDM. Attribute options such as maximum and minimum cardinality, unique and distinct, specify the structural constraints of the data.

Persistence

The persistence of an object is the length of time for which it is accessible to the program. This time span can vary from variables which exist only in the block in which they are defined, to data which outlasts the run of the program, or even the existence of the program. In most languages the only long-term persistent structure available is the **file**.

A persistent programming language can be defined as [2]: a language which provides for the longevity of values of all types and does not require explicit movement of data to and from disk. Persistence is orthogonal to type: the code used to manipulate a value is not dependent on the value's persistence.

There are a number of advantages to using orthogonal persistence. Firstly there is a conceptual simplification. Instead of three mappings: between the program and the real world, the program and the database, and the database and the real world, there is only one mapping (between program and real world). Procedures and program modules can be regarded as first class values and may be stored in the persistent store. The persistence mechanism allows for incremental loading and type checking, as well as providing a mechanism for program and library management.

3 The PERCI Language

PERCI is a persistent language which provides features for defining classes, class hierarchies, associations between classes and permissable operations on class members. This section contains a brief description of the data modeling and constraint handling features of the language. A more detailed description of PERCI may be found in [16].

Example database

An example data structure for a parts database is shown in Appendix A. This example will be used to illustrate the descriptions which follow.

Types

The type system serves to give the data a structure and to enforce constraints on the way the data is used. The type system supports structural equivalence between types. Functions are first class objects: they can be used in any context where a type is permitted i.e. they can be bound to identifiers, stored in memory locations, and passed as function parameters.

The basic data types consist of: integers, reals, char, boolean, subranges and enumerations.

Structures

Structures are constructed from base types and other structured types. Structures are generally the type over which classes are defined. A structure has a name and one or more attributes. Attributes consist of name-type pairs and may be constrained by attribute modifiers such as **unique** or **const**.

```
struct PartStructure with
  keys
    P#: Pnum, const;
  attributes
    Type: PartType;
    Name: String[25];
  end;

Part: class of PartStructure;
```

Figure 1. Structure and class definition

Figure 1 contains a structure definition for the data type *PartStructure*. *P#* is defined as a key attribute, which means that it will serve as the key for any class defined over *PartStructure*.

Subtypes

A type T_1 is a subtype of a type T_2 when all the values of type T_1 are also values of T_2. Subtypes are explicitly declared using the **isa** construct. T_1 **isa** T_2 implies that T_1 is a subtype of T_2, its supertype. Subtypes can only be defined from a single supertype (simple inheritance).

Subtypes inherit the features of their supertypes which are added to their own. A subtype thus has at least the same properties as its supertype, and may have additional properties. A value of type T_1 can be used wherever a value of type T_2 could be used when T_1 is a subtype of T_2.

```
struct BasePartStructure isa PartStructure with
   attributes
      Cost: Rands;
      Mass: Grams;
   end;
BasePart: class of BasePartStructure
   subclass of Part
   where Type == BASE;
```

Figure 2. A subtype and subclass definition

Figure 2 shows the definition of a subtype and a subclass. *BasePartStructure* is defined to be a subtype of *PartStructure*, inheriting all its supertype's attributes and adding two new attributes to the subtype.

Classes and Subclasses

The main abstract data type is the class, which represents some meaningful collection of entities. A class can be either a base class or a subclass. A base class is defined independently of all other classes in the database. A subclass does not exist independently; it is defined in terms of other classes. A subclass is a class which contains some, but not necessarily all, the members of its parent class. A subclass inherits all the attributes of its ancestor classes in the generalization hierarchy.

A class is a variable which denotes a particular set of objects, all of the type over which the class is defined. It is possible to define multiple classes over the same type.

Figure 1 defines a class called *Part* over the type *PartStructure*. This class has the member attributes: *P#*, *Type*, and *Name*. The attribute *P#* is a key attribute, which means that it must be unique for each member of a particular class over the type *PartStructure*. A class can be defined over any type, but usually structured types are used. When a class is defined over a simple data type, it is effectively defining a set over the type.

Figure 2 defines a class called *BasePart* which is a class over the type *BasePartStructure*, and a subclass of *Part*. The **where** clause specifies that the *Type* attribute of an object must be *BASE* for it to qualify as a *BasePart*. If the clause **where specified** is used instead, it indicates that an object has to be explicitly placed in the subclass. An object placed in a subclass automatically becomes a member of the parent class. The type of a subclass must be a subtype of the type of its parent class.

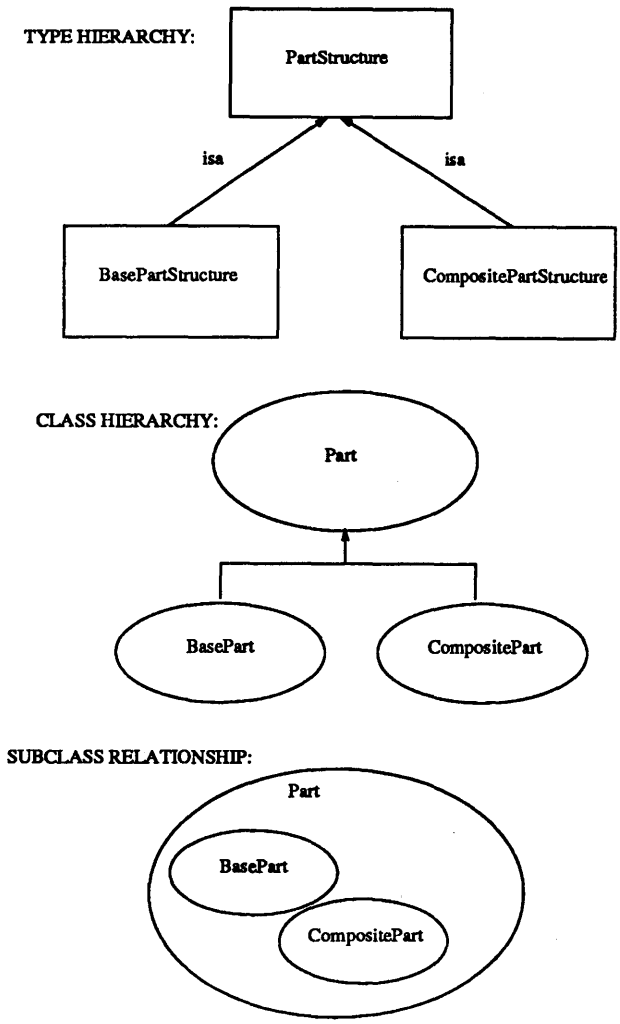


Figure 3. Type and class hierarchies for Parts, BaseParts, and CompositeParts

Figure 3 shows the relationship between types, subtypes, classes and subclasses. An instance of the class *BasePart* is also an instance of the class *Part*, since the class *BasePart* is a subset of the class *Part*. We could define another class over the type *BasePartStructure*, call it *SomePart*. These two classes, *BasePart* and *SomePart*, have the same type, but contain different objects i.e. they provide different sets of *BasePartStructure*.

Attributes of a class may be either data-valued (DVA) or entity-valued (EVA) [6]. A DVA attribute describes some property of each element in a class by associating the entity with a value or set of values from some domain. An EVA describes a property of each entity by relating it to an entity or entities of another or the same class. EVA's serve to relate classes to each other.

The modifiers **req** (required), **uniq** (unique), **inv** (inverse) and **const** (unchangeable)may be applied to any attribute. Multi-valued relationship are indicated by defining an attribute as a class variable over some type. For example: the attribute *Uses* of structure *CompositePartStructure* defines a multi-valued relationship with *UsesStructure*.

A pair of attributes can be related by means of inversion. Attribute A_1 of class C_1 can be specified as the inverse of attribute A_2 of class C_2. This means that the value of

A_1 for member M_1 of C_1 consists of those members of C_2 whose value of A_2 is M_1.

An attribute may be derived from other data, either in the same structure or in some related entities (EVA's).

Constraints

Integrity constraints are rules which limit the allowed values of a data type and the transitions which may take place between different values [10], [5], [3]. Constraints can be grouped into a number of classes:

- Attribute constraints
- Entity constraints
- Constraints on collections of entities
- Types
- Dependencies
- Preconditions and postconditions

The first four can be found in PERCI [15]. Constraints can be either implicit or explicit. Implicit constraints arise out of the constructs of the data model itself. An example of this type of constraint is that of type hierarchies. Explicit constraints are defined using constructors e.g. single-valued or multi-valued attributes.

Explicit constraints consist of static and dynamic constraints. Static integrity constraints serve to restrict the possible values of a class. Dynamic integrity constraints place restrictions on the way data values may be changed.

In PERCI a structure's allowed values can be specified by means of a constraint expression. This expression may be a simple expression involving only constants, or it may refer to other values in the database. Constraints are classified as initial, final or always, depending on when they are to be checked. Initial constraints are checked when an object is added to the class. Final constraints must be true when an object is removed from the class. Always constraints must be true at all times for any object in the class.

```
struct Customer with
  attributes
    name: string[25];
    address: Address;
    amountDue: Rands;
    creditLimit: Rands;
    creditRating: Rands;
  initially
    startClean: (amountDue == 0);
  finally
    noDebt: (amountDue == 0);
  always
    underLimit: (amountDue <= creditLimit);
end;
```

Figure 4. Structure with initial, final, and always constraints

In the structure shown in figure 4 three constraints are defined. The first is an initial constraint and says that the value of *amountDue* must be 0 when an object is added to any class of type *Customer*. The second constraint will not allow an object to be removed from a class of this type unless it has a value of 0 for *amountDue*. The third constraint specifies that the value of *amountDue* must always be less than or equal to the value of *creditLimit*. An assignment which attempts to violate this constraint will not be allowed.

A function may suspend a particular constraint while it modifies the data, reinstating the constraint when it returns to the calling function. This allows a function to temporarily place the data in an inconsistent state. This is necessary because certain data transformations between consistent states requires the data to be temporarily inconsistent. The function *DebitAccount* in figure 5 suspends the constraint *inBalance*. This allows the function to debit a value to the account without causing a constraint violation.

```
struct AccountStruct with
  attributes
    credit: Rands;
    debit: Rands;
    balance: Rands;
  always
    inBalance: (balance == credit - debit);
end;

Account: class of AccountStruct;

function DebitAccount (acc, amount ) with
var
  acc: AccountStruct;
  amount: Rands;
suspend
  acc.inBalance
actions
  acc.debit := acc.debit + amount;
  acc.balance := acc.balance - amount;
end;
```

Figure 5. Suspending a constraint

A set of constraints is contradictory if they cannot be simultaneously satisfied. It is, in general, not possible to statically detect contradictory constraints, since they can be arbitrarily complex and may depend on data values which cannot be statically determined. There are two types of contradictory constraints: always contradictory (which are always contradictory for any data values), and intermittently contradictory (only contradictory for some sets of data values).

Problems are introduced when there is more than one way of accessing a particular data object. Aliasing makes programs less clear and makes constraint checking more complex.

The constraint verification system should ensure that the data are never allowed to achieve a state which violates any declared constraint. The system must verify all constraints and must do so at the earliest possible time. If constraint checks are not factored out as soon as possible, the overhead of making all the checks may cause great inefficiency. Run-time checks must only be produced for those checks which could not be performed at compile-time.

Constraint verification may be necessary in the following cases:

- After a constraint is declared, to ensure that there is no violation of the constraint at the point of its creation.
- After an assignment the new value must be checked.
- When a new instance of a class is created.
- When returning from a function call. A function may temporarily suspend a constraint until after the function has executed.

4 Implementation of Constraints

In order to implement constraint checking a method is needed to produce and store a function which will check that the data obeys the constraint definitions. The ideal solution is to store such the constraint function in executable form. The PERCI compiler [15] makes use of a (dynamically) callable compiler to construct the constraint checking functions. (See figure 6.) It is written in Napier and produces code which runs on the Napier Persistent Abstract Machine (PAM).

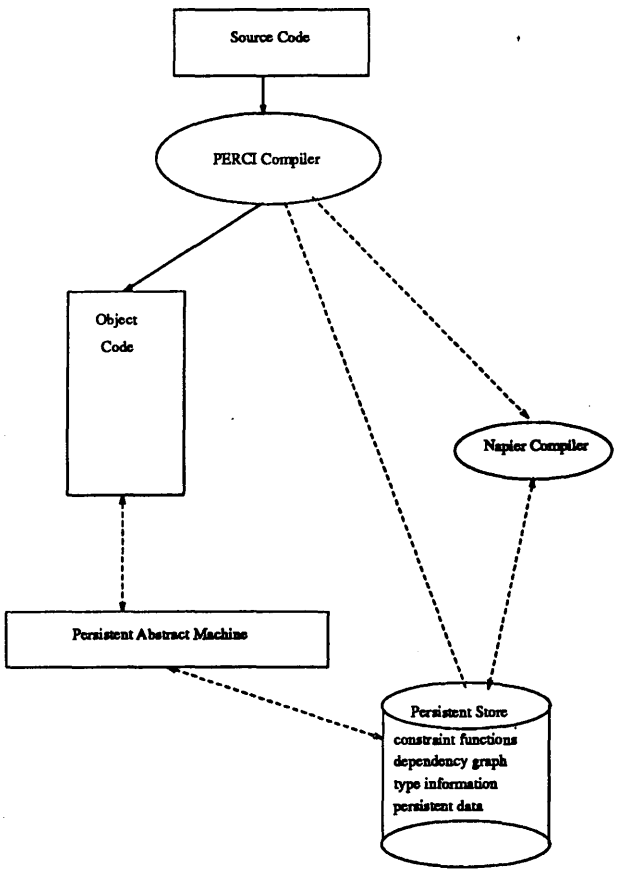


Figure 6. The compiler and constraint subsystem

The Napier [8] system consists of the language and its persistent store. The Persistent Abstract Machine executes Napier programs and accesses the persistent store. The persistent store contains the Napier compiler as a procedure. This means that a program can generate another program by supplying the compiler with a string on which to act.

The advantage of using Napier to implement the PERCI compiler is that we can use its callable compiler to create an executable function which is then placed on the persistent store. It is for this reason that Napier was chosen as the implementation language.

When the compiler encounters a constraint declaration it parses the declaration and checks its validity. The compiler then constructs a function which will be used to verify the constraint. This function is compiled by the Napier callable compiler and stored on the persistent store.

The compiler constructs a dependency graph (Figure 7) of the type hierarchies so that dependencies between types can be checked and inherited constraints enforced. This dependency graph is also placed on the persistent store so that it is available while compiling and when running the compiled program.

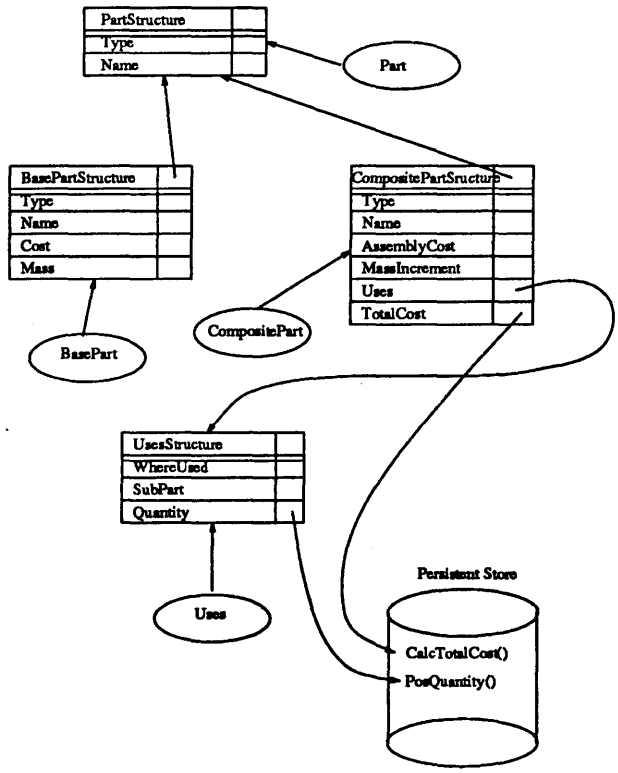


Figure 7. Dependency graph for the example program

Each class object has a pointer to its type object. The type object is part of the dependency graph and also contains references to any constraint functions acting on variables of that type. When a variable is assigned a value, the constraint management system checks whether the variable is constrained by consulting its type object in the dependency graph. The appropriate constraint functions are retrieved from the persistent store and called to check whether the requested changes to the variables will violate any of the declared constraints. If a violation is detected, the update is abandoned and an error condition is generated. Figure 8 contains an example of the creation and execution of a simple constraint on the values of *Quantity* in the class *Uses*. Figure 9 shows the life-cycle of a typical constraint.

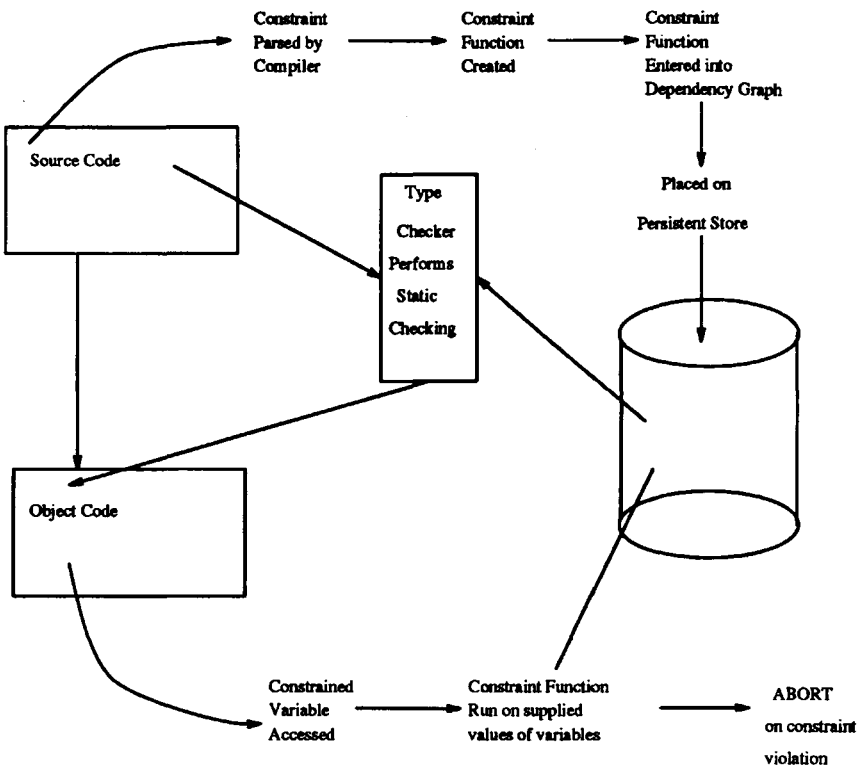


Figure 9. The life-cycle of a constraint

Creating the constraint:

```
constraint:
  PosQuantity: (Quantity > 0);

compiler:
  parses constraint PosQuantity
  creates function PosQuantity()
  compiles function PosQuantity()
    using Napier callable compiler
  stores PosQuantity()
    on Persistent Store in dependency graph
```

Executing the constraint:

```
program:
  X.Quantity := X.Quantity – 999;

compiler:
  looks up UsesStructure in the dependency graph
creates object code:

  tmp := X.Quantity – 999;
  if (not PosQuantity(tmp)) ABORT;
  X.Quantity := tmp;
```

Figure 8. Example of creation and execution of a constraint.

5 Comparison with existing persistent languages

Galileo

Galileo is based on Semantic Data Modeling concepts and provides subtypes and subclasses. The type system is partially derived from an early version of ML [2].

The language is strongly typed, statically type-checked and has constructs for persistence, modularisation, higher-order functions and data modeling. It is an expression-based, interactive database query language.

User-defined abstract data types are allowed. These data types have the same status as primitive data types. Abstract data types are used to provide unique operations for each type. These operations are only defined on the abstract type to which they belong. This allows a program to be independent of changes in the representation of the data.

Galileo supports subtypes. They are automatically inferred for concrete types, but have to be specified for abstract types.

Galileo provides mechanisms for transaction handling.

Environments are used to model a database schema. Such a unit encapsulates the data and the operations available on it. Environments allow for persistence, views, and logical data independence. To provide persistence, a global environment is assumed in which all values are maintained by the system. The environment can be expanded by adding new bindings to it. A new environment can be generated by extending the current environment with new definitions.

The static type checking can be a problem when the database has to be extended or modified [4]. It would be difficult to transfer data between different databases or use

the same program on a different database.

Persistence is an orthogonal property of data and is implemented with the help of environments. The only way data can be made to persist is by saving the entire workspace.

The language provides for type hierarchies and classes with inheritance. The naming of types and their extent is separated, but multiple extents over the same type are not allowed.

TAXIS

TAXIS [9] was designed for the creation of interactive database systems, with large volumes of short, update intensive transactions. Classes are used to model passive and active objects, such as transactions, expressions, constraints and exceptions. It makes extensive use of inheritance over classes. Programming is done by defining classes.

There are three types of objects in TAXIS: *tokens* (representing constants), *classes* (describing collections of tokens), and *metaclasses* (describing collections of classes). Classes and tokens have properties through which they can be related to other classes and tokens. Properties of classes give information about the structure of instances of the class, while properties of tokens specify the structure of the token itself.

By using metaclasses with inheritance, new classes can be created with different features. All programming in TAXIS is done by defining classes. Procedures are represented by instances of a **Transaction Class**. A **Variable Class** is similar to a type with an associated extent. The **Exception Class** is used to raise an exception when a result or expression has a specified value. The IS-A relationship is defined over classes and metaclasses and is used to create a hierarchy of classes.

TAXIS tries to use inheritance to support all aspects of database programming. There is some doubt [2] as to whether this can be done without requiring a large number of special classes and a resulting increase in complexity.

All database programming in TAXIS is performed by defining classes. Classes are objects themselves and are instances of the metaclass ANY. This is useful for attaching properties to classes themselves instead of instances. TAXIS allows multiple inheritance (a class may be a subclass of more than one class).

There is very little non-persistent data in TAXIS. All types which have an associated extent persist, and there is no mechanism to create subsets of those extents (useful for storing transient data).

PSAlgol

PS-algol [12] is a block structured language derived from the language S-algol, which was in turn based on the language Algol-s. The emphasis of the language is on simplicity. There are few constructs available, but the power of the language has been increased by extending the scope of the features. The language is strongly typed and, as far as possible, statically type checked.

Structures are complex objects similar to Pascal records. The fields of the structure may be of any type. A class constructor may be defined. It consists of a name for the class and a set of names and types for the fields of the class. The constructor is used to generate an instance of a class, the type of which is **pntr**. The individual fields of the object can be dereferenced and assigned to. The copy operation of objects is pointer based, therefore a copy of an object creates another reference to the same object and not a new object.

A **pntr** object may reference an object of any class. It can be used to refer to objects without knowing their class. The **pntr** object allows for delayed binding and the modeling of complex data. Because the language is strictly type-checked, each object has to be specified before it is used. It is possible, however, to write a procedure which handles a number of different types by putting these objects in structures and passing pointers to the structures.

Procedures are first-class and thus may be manipulated in the same way as any other data objects. In particular, they may be placed in structures and thus allow the use of abstract data types where the access to data is restricted to a set of operations defined on that type.

A callable compiler is available which takes a string representing a procedure and returns the compiled procedure which may then be used in the same way as any other procedure.

Exceptions are provided to allow events which cannot be handled by the current procedure to be passed to successive outer blocks until they can be handled.

Persistence is provided through the use of tables. A table is a set of one-to-one mappings between strings or integers (the keys) and **pntr** objects. They are used to build keyed sets of objects. The operations available on tables are: create an empty table, insert a key-value pair into a table, retrieve an object using a key, and apply a procedure to all the objects in a table.

To allow data to persist, it must be entered into a structure which is reachable via **pntr** chains from a persistent root of type **pntr**. Any data can persist since any data can be placed in a structure. The persistent root is called a database.

PS-algol provides uniform persistence. Since it is data type complete, data of any type is allowed to persist. Many databases may be accessed at the same time and data can be transferred between them or cross-references between databases set up.

Because there are no higher-level data types (bulk data types) such as those provided in Pascal/R and DBPL, programs may require too much detail.

The **pntr** data type provides for polymorphism, but this use of untyped pointers is inelegant and a potential source of errors.

Napier

Napier was developed from PS-algol. The Napier system consists of the language and its persistent store. The system is built on a layered architecture containing the Persistent Abstract Machine Layer and the Stable Persistent Storage Layer [8].

The Persistent Abstract Machine (PAM) provides the

ability to execute Napier programs, handle error conditions, and monitor interactions with the file system. The persistent store must be stabilised (transformed from one consistent state to the next), in order to preserve the persistent data.

The type system allows for flexible and eager incremental binding. Types are sets of values from the name space. These sets may be predefined, like integer, or built using one of the predefined type constructors, like structure. These constructors obey the principle of data type completeness, which allows a rich type system to be described with a small number of rules. Type checking is mostly static, except where required to support orthogonal persistence. The type system is polymorphic and uses existentially qualified types to provide for abstract data types.

The structures **env** and **any**, the infinite union of all types, are used to support persistence. The Napier type system allows the user to indicate whether static or dynamic binding should be used, by choosing the appropriate constructor. Eager type-checking is used so that types are checked as early as possible. The type constructors **env** and **any** are dynamically checked; all other type checking is static.

All data objects in Napier may persist. There is a root of persistence called 'PS'. Objects which are to persist beyond the life of their creator must be reachable from the persistent root. This persistent root is of type *environment*. Objects of this type consist of name-value pairs. They differ from structures in that bindings can be dynamically added to objects.

Perci

PERCI provides semantic data modeling features similar to Galileo and TAXIS. However, PERCI obeys the principles established for persistent programming languages in [2], which the other two do not. It is also a true programming language: TAXIS and Galileo were intended to be database design languages.

Persistence in PERCI is provided by classes. Since any type may be placed in a class, any type may persist. Since functions are first-class, they may placed in classes and made to persist as well. This allows data and the operations allowed on the data to be stored together. Types and extents are separate, as in Galileo, but PERCI also allows multiple extents over the same type.

PERCI provides a more generalised constraint handling system than Galileo. Galileo provides for only static constraint checks which are made when data is created. PERCI can enforce constraints at any stage: at initialisation, deletion, or throughout the life of the data.

PERCI provides the bulk data modeling features which PS-algol and Napier lack. This means that less detail is required when modeling complex data, and the representation of the data is closer to the real system.

PERCI does not provide for concurrency, exceptions or environments.

6 Conclusion

The PERCI system provides a persistent programming language with a semantic integrity constraint handling system. It allows programmers to concentrate on the program without being concerned about the translation and storage of data. The constraint system allows the semantic information to be placed where the data is defined, making for a safer, more secure system.

The use of Napier to implement the PERCI system has a number of advantages, the most important being the availability of the callable compiler. This allows the compiler to automatically generate the constraint checking functions.

PERCI provides for the semantic content and constraint handling which languages like PS-Algol and Napier lack. While both Galileo and PERCI provide for semantic modeling and constraint checking, the system provided by PERCI is less restrictive. Galileo does not allow types and extents; it also provides only static constraint checking. Furthermore, Galileo is not a true persistent programming language; it is a conceptual modeling language and as such does not obey all the principles established for persistent programming languages.

References

1. A Albano, L Cardelli, and R Orsini. 'Galileo: A Strongly-Typed, Interactive Conceptual Language'. *ACM Transactions on Database Systems*, **10**(2):230–260, (June 1985).
2. M Atkinson and O P Buneman. 'Types and Persistence in Database Programming Languages'. *ACM Computing Surveys*, **19**(2):105–190, (June 1987).
3. F Bancilhon and P Buneman. *Advances in Database Programming Languages*, chapter 12, pp. 187–217. Addison-Wesley, 1990.
4. R Cooper. *On the Utilization of Persistent Programming Environments*. PhD thesis, University of Glasgow, Department of Computing Science, September 1989. 90/R12.
5. C J Date. *An Introduction to Database Systems*, volume 2 of *The Systems Programming Series*, chapter 2, pp. 35–81. Addison-Wesley, 1984.
6. M Hammer and D McLeod. 'Database Description with SDM: A Semantic Database Model'. *ACM Transactions on Database Systems*, **6**(3):351–386, (September 1981).
7. W Kent. 'Limitations of Record-Based Information Models'. *ACM Transactions on Database Systems*, **4**(1):107–131, (March 1979).
8. R Morrison, F Brown, R Connor, and A Dearle. 'The Napier88 Reference Manual'. Persistent Programming Research Report 77, Universities of Glasgow and St Andrews, (July 1989). PPRR-77-89.
9. J Mylopoulos, P A Bernstein, and H K T Wong. 'A Language Facility for Designing Database-Intensive Applications'. *ACM Transactions on Database Systems*, **5**(2):185–207, (June 1980).

10. S H Nienhuys-Cheng. 'Classification and Syntax of Constraints in Binary Semantical Networks'. *Information Systems*, **15**(5):497–513, (1990).
11. J Peckham and F Maryanski. 'Semantic Data Models'. *ACM Computing Surveys*, **20**(3):153–189, (September 1988).
12. Persistent Programming Research Group. *PS-algol Reference Manual*, 2nd edition, 1985. PPRR-12-85.
13. J W Schmidt, H Eckhardt, and F Matthes. *DBPL Report*. Fachbereigh Informatik, Johann Wolfgang Goethe-Universitat, draft edition, November 1988.
14. J M Smith and D C P Smith. 'Database abstractions: Aggregation and generalization'. *ACM Transactions on Database Systems*, **2**(2):105–133, (June 1977).
15. S B Sparg. 'The implementation of constraints in the persistent programming language Perci'. Technical Report in preparation.
16. S B Sparg. 'Syntax of the language Perci'. Technical Report, March 1992.

Appendix A Example Database

```
struct PartStructure with
 keys
   P#: Pnum, const;
 attributes
   Type: PartType;
   Name: String[25];
 end;

struct BasePartStructure isa PartStructure with
 attributes
   Cost: Rands;
   Mass: Grams;
 end;

struct CompositePartStructure isa Part with
 attributes
   AssemblyCost: Rands;
   MassIncrement: Grams;
   Uses: class of UsesStructure;
   TotalCost: CalcTotalCost(this);
 end;

struct UsesStructure with
 attributes
   WhereUsed: CompositePartStructure inv of Uses;
   SubPart: Part;
   Quantity: integer;
 constraint
   PosQuantity: (Quantity > 0);
 end;

struct CostAndMassStructure with
 attributes
   Cost: Rands;
   Mass: Grams;
 end;

struct Customer with
 attributes
   name: string[25];
   address: Address;
   amountDue: Rands;
   creditLimit: Rands;
   creditRating: Rands;
 initially
   startClean: (amountDue == 0);
 finally
   noDebt: (amountDue == 0);
 always
   underLimit: (amountDue <= creditLimit);
 end;

struct AccountStruct with
 attributes
   credit: Rands;
   debit: Rands;
   balance: Rands;
 always
   inBalance: (balance == credit - debit);
 end;

Account: class of AccountStruct;

function DebitAccount (acc, amount ) with
 var
   acc: AccountStruct;
   amount: Rands;
 suspend
   acc.inBalance
 actions
   acc.debit := acc.debit + amount;
   acc.balance := acc.balance - amount;
 end;

Pnum = integer[0..100000];
PartType = (BASE, COMPOSITE);
Rands = integer[0..10000];
Grams = integer[0..100000];

TempParts: PartStructure;
TempBaseParts: BasePartStructure;
```

```
Part: class of PartStructure;

BasePart: class of BasePartStructure
 subclass of Part
 where Type == BASE;

CompositePart: class of CompositePartStructure
 subclass of Part
   where (Type == COMPOSITE);

Uses: class of UsesStructure;

CostAndMass: class of CostAndMassStructure;

function CalcTotalCost(thisPart) with
 var
   thisPart: Part;
   usedPart: Part;
   cost: Rands;
 actions
   if thisPart.Type == BASE then
     cost := thisPart.Cost;
   else
     begin
       cost := 0;
       foreach usedPart in thisPart.Uses do
         cost := cost + CalcTotalCost(usedPart);
         cost := cost + thisPart.AssemblyCost;
     end
 return
   cost;
 end;
```

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for *review* should be prepared according to the following guidelines.

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1, 2, 3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) LaTeX file(s), either on a diskette, or via e-mail/ftp – a LaTeX style file is available from the production editor;
2. In camera-ready format – a detailed page specification is available from the production editor;
3. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

   Further instructions on how to reduce page charges can be obtained from the production editor.
4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for LaTeX or camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 3 and 4 above will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

1. E Ashcroft and Z Manna. 'The translation of 'goto' programs to 'while' programs'. In *Proceedings of IFIP Congress 71*, pp. 250–255, Amsterdam, (1972). North-Holland.
2. C Bohm and G Jacopini. 'Flow diagrams, turing machines and languages with only two formation rules'. *Communications of the ACM*, 9:366–371, (1966).
3. S Ginsburg. *Mathematical theory of context free languages*. McGraw Hill, New York, 1966.

South African Computer Journal

Suid-Afrikaanse Rekenaar-tydskrif

Number 7, July 1992
ISSN 1015-7999

Nommer 7, Julie 1992
ISSN 1015-7999

Contents