HAL open science

Specifying with SACSO

Nicole Levy, Alina Piganiol, Jeanine Souquières

To cite this version:

Nicole Levy, Alina Piganiol, Jeanine Souquières. Specifying with SACSO. [Research Report] RR-0683, INRIA. 1987. inria-00075870

HAL Id: inria-00075870

https://inria.hal.science/inria-00075870v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

UNITE DE RECHERCHE
INRIA-LORRAINE

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 683

SPECIFYING WITH SACSO

Nicole LEVY
Alina PIGANIOL
Jeannine SOUQUIERES

JUIN 1987

SPECIFYING WITH SACSO

SPECIFIER AVEC SACSO

N. Levy, A. Piganiol, J.Souquières

Centre de Recherche en Informatique de Nancy
BP 239 54506 Vandoeuvre-les-Nancy Cedex (France)
Tel: 83.91.20.00

Appeared in Proceedings 4th International Workshop on Software Spécification and Design, IEEE, Monterey (California), April 1987.

15/5/1987

PAPIER RECUPERE ET RECYCLE

ABSTRACT:

The aim of our approach is to improve the requirement specification step, which is very important during system development. But rather than study languages, models and tools in the requirement specification context, we work on the formalization of the design specification we call **method**. Our goal is to describe a multi-method system, a system which allows the user to define his own method or to use predefined ones. During the design specification step, the system will guide the specifier in terms of the chosen method.

RESUME:

Le but de notre approche est de faciliter l'étape de spécification, étape essentielle dans le cycle de développement d'un système. Mais au lieu d'étudier des langages, des modèles et des outils, nous travaillons sur la formalisation du processus de construction d'une spécification, appelé **méthode**. Notre but est de décrire un système multi-méthodes permettant à l'utilisateur de définir sa propre méthode ou d'en utiliser des prédéfinies. Durant l'étape de construction d'une spécification, le système guidera le spécifieur selon la méthode choisie.

15/5/1987

SPECIFYING WITH SACSO

N. Levy – A. Piganiol – J. Souquières

Centre de Recherche en Informatique de Nancy – Tel: 83.91.20.00.
BP 239 54506 Vandoeuvre–les–nancy cedex (France)

Abstract :

The aim of our approach is to improve the requirement specification step, which is very important during system development. But rather than study languages, models and tools in the requirement specification context, we work on the formalization of the design specification we call **method**. Our goal is to describe a multi-method system, a system which allows the user to define his own method or to use predefined ones. During the design specification step, the system will guide the specifier in terms of the chosen method.

1 – INTRODUCTION

Many studies about specifications are presently made. Some of them aim at the implementation of formal results obtained in the study of abstract data types such as ASSPEGIQUE [2], LARCH [10], OBJ [9], ... Other studies stem from more practical considerations dealing with the complexity of large information systems and offer models such as PSL/PSA [15], ACM/PCM [3], GIST [14], ...

Much more emphasis has been laid on specification languages, models and tools than on methods. Specifying a complex system does not just consist in writing a linear sequence of formal statements. It requires creativity: organization of the specification, design decisions. So it seems important to formalize the step of design specification in order to introduce specification methods. Some research teams are now concerned with these method concepts [7], [8].
But this approach can be dangerous if we only suggest one way of doing it: the corresponding system would be too restrictive for a well-informed user. This can be avoided by proposing a set of methods or allowing the user to define his or her own method by means of an appropriate language.

With SACSO, we are concerned with specifications in the most extended meaning. Our work derives from an experience gained in industrial environments such as information systems [6] or image processing [1]. It is built on:

- the use of a formal language based on algebraically specified abstract data types, which allows structuration and verification,
- a set of methods whose goal is to help the user with the construction step,
- a set of automatic tools particularly oriented towards the computer aided typing, the presentation and the validation of the specification. Parameterization by design methods is the main feature of the system.

This paper aims at giving a brief account of our work. In section 2, we introduce the SACSO project. In section 3, we develop the notion of methods, and illustrate it in section 4 with the library system example proposed as a case study for the workshop.

2 – A PRESENTATION OF SACSO

Goal

Helping with the step of design requirement specification is the goal of the SACSO system. It guides the specifier from an informal description towards a precise, complete and formal description of the system to be developped, with all its properties.
The design step is progressive by use of basic construction operators as shown in fig.1. Let us notice that the design process can induce modifications of the initial informal description.

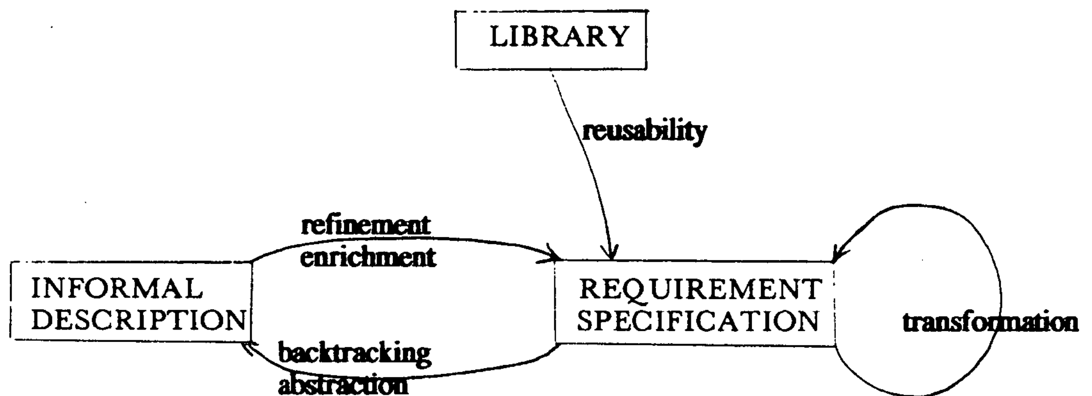

fig.1 : construction operators of a specification

The **enrichment** operator completes the specification from informations picked up in the informal description.
To **refine** a specification is to specify some details that have been left aside undefined in a previous step of the design process. On the contrary, to **abstract** means to "forget" some information from the informal description, considered as over-specification for the actual step.
The **backtracking** allows to review the informal description when some inconsistancies have been detected or to undo some decisions taken during an enrichment step.
The **transformations** do not change the behaviour of the specification, but allow to :
- eliminate redundancies,
- increase the modularity of the specification in order to increase its evolutivity,
- simplify a specification to make it more understandable, or
- increase the appropriateness of the data structure for some of its operations.

The specification model

A specification is defined by the enrichment of a specification called its environment. In particular, any specification enriches a basic specification containing data types such as Integer or Boolean and type constructors (or parameterised data types) such as Cartesian Product, Set Sequence, Table, Union. These built-in types are algebraically specified with their usual operations [12].

A type is defined by the extension of a reused or instanciated data type which is called its expression. It inherits all the operations from the latter. It is enriched by some operations defined in terms of finer ones, which can be predefined or not. The objects of a type can be restricted to those verifying a property or constraint called an invariant, this invariant being a predicate. The objects which do not verify it are equal to an undefined constant.

Helps

We are concerned by two problems: one in the design step with the definition and the use of methods, and the other in the validation step.
The goal of the validation is to prove the appropriateness of the specification to the initial informal description and to the needs of the customer. It can be done :

- by direct interpretation of the functions described in the specification, in order to test its individual behaviour. Functions might be incompletely defined;
- by prototyping the complete specification in order to test it;
- by rephrasing, using several external representations such as graphic, textual, or arborescent ones.

General ideas

The design requirement specification is based on the notion of reusability. Reusability is introduced at language level with parameterized abstract data types, and at model level by modification and enrichment of existing specifications stored in a library. It is performed stepwise by composition of construction operators. The construction is not linear, and at any time incomplete definitions and potential semantic errors are allowed. In order to free the specifier from repeating definitions, even in a different way, the system inferes and proposes definitions. When it has nothing to put forward, it reminds the context giving the data type of the object to define, or the profile of the operation or their use in the current specification, for example.

3 – METHOD

A method is an accurate description of the process to follow for designing a specification: the ordering of the actions to be done and what kind of manipulated objects are concerned. Any action consists in pointing out the next entity to be taken into account. It leads from one specification state to another. Consequently, the specifier can forget the chronological links of the various steps in the thought process. He can focus his attention on the structure of the problem to be solved. Thus the dynamic aspect of reasoning activities is not his concern anymore.

Describing a particular method consists in ordering the set of actions. This leads to the definition of a complete specification. For example, a method will suggest pointing out each entity which seems important and giving its intuitive definition.

Another method will suggest to completely define an entity before introducing a new one.

The method chosen will be strongly dependent on the problem type as well as its environment, such as working habits, the concerned person's previous training, etc.

A multi–methods system

A specification design system which uses a specific approach may be limited in its application area to a special class of problems or users. Furthermore, the complete resolution of a problem is rarely the application of a single method: it will generally be the conjunction of several ones (top–down, reusability...), each of them occuring at one moment of the specification development.

The SACSO system is meant to provide the user with a set of design methods rather than to favour a specific approach. Several methods are available; the user can link them to his liking for a given problem. In order to achieve that purpose, a language for describing methods has been studied: it enables the definition of new methods at a low cost. It takes into account the dialogue between user and system, integrating indications about external presentation (such as the information to be given to the user) and questions–answers (such as a choice among a given list of entities likely to admit a definition).

Describing a method

A method is conceived as a set of production rules. The formalism of the production system has been selected on account of the modular expression of the knowledge and the uniformity fixed for their expression (a single structure for the rules).

In order to take into account the temporal aspect linked to the method, the rules are summed up inside a "decision" tree, which describes two kinds of relations between rules:

- an obligation relation: the rule j will be activated at the time t if and only if the rule i has been activated at the time t–1 (for example an answer can be given only after a question has been asked),

- a conjunction relation: the rule i and the rule j have their condition part simultaneously confirmed at the moment t;

The condition part of a rule is a logical formula of the proposition calculus dealing with the actual specification and the record of the session.
The action part of a rule is concerned by two aspects:
- the modifications of the specification to be done,
- the dialogue between the user and the SACSO system.

The description of a method is made from the model of a SACSO specification. It is seen as a parameterizable module: the parameters point to aspects which are not directly linked to the construction steps (whatever the actual value, the ordering is not changed). The parameters are:
- the helps to be given by the system (inferences and propositions),
- the display (choice of the external representation),
- the checks to be made.

4 – CASE STUDY: A LIBRARY

We are going to study the example of the library, and how a method guides the user during the construction of his or her specification. The method we follow is divided into three main actions:

First action: point out all the essential operations of the future system. For each of them, give its name and its profile as precisely as possible, and its informal definition. All these operations will have in their domain a type which will be the most general type of the system (an object of that type will contain all the information about a system). Among the operations, modifiers and observers will be distinguished.

Second action: structure the introduced types, beginning with the most general one. Each of them will be structured by a predefined parameterized type and will use simpler types.

Third action: define all the introduced operations as a composition of predefined operations and of simpler operations to be defined later on. Start with a modifier operation.

Fourth action: When all the introduced operations have been defined, complete the specification by giving the invariants of the types and the details left aside during the three first actions. Start with the last defined types and operations.

CONSTRUCTION OF THE LIBRARY SPECIFICATION

4.1 – First action

From the statement of the problem, we point out the main functions required, with their profile (they can be incomplete).
Let us introduce the type "**LIBRARY**" as a general type whose objects contain all the information about a library. The functions

- check-out, return, add-a-copy and remove-a-copy change the library state: they are the modifiers.

- get-from-author, get-from-subject, find-out-books and find-out-borrow get their information in a library: they are observers.

As only the staff users are allowed to perform the functions 1, 2, 4 and 5 below, a user will have to give his or her name to perform these functions.

1. check-out : **LIBRARY, A-COPY, NAME --> LIBRARY**
   return : **LIBRARY, A-COPY, NAME --> LIBRARY**
2. add-a-copy : **LIBRARY, A-COPY, NAME --> LIBRARY**
   remove-a-co: p **LIBRARY, A-COPY, NAME --> LIBRARY**
3. get-from-author : **LIBRARY, AUTHOR --> LIST-OF-BOOKS**
   get-from-subject : **LIBRARY, SUBJECT --> LIST-OF-BOOKS**
4. find-out-books : **LIBRARY, BORROWER, NAME --> LIST-OF-BOOKS**
5. find-out-borrow : **LIBRARY, A-COPY, NAME --> BOR**

4.2 – Second action

We structure the types.
The library contains books and knows the borrowers; therefore, it has at least two components:

LIBRARY == CARTESIAN-PRODUCT [books: **BOOKS**,
borrow: **BORROWERS**]

The borrowers are referred to by their name and characterized by two things:
- their status: "staff" or "ordinary",
- the set of the books they have borrowed.

The **BORROWERS** type is structured as a table which associates the information related to a borrower with his or her name.

BORROWERS == TABLE [**NAME**, **TWO-THINGS**]

TWO-THINGS == CARTESIAN-PRODUCT
[status: **STATUS**, bc: **BC**]

STATUS == {"staff", "ordinary"}

BC == SET [**BOOK-CHARACT**]

The objects of **BOOK-CHARACT** are the characteristics of a book that particularize it.

The books of a library will be acceded individually or globally: we structure them as a table. A book is particularized by its author and a title. Here we introduce the component "title" to characterize each book of an author. A subject is associated with each book.
A book will have several copies, each one particularized by a number, for example. A copy of a book can be "available" or "checked-out"

BOOKS == TABLE [**BOOK-CHARACT**, **BOOK**]

BOOK-CHARACT == CARTESIAN-PRODUCT
[author: **AUTHOR**, title: **TITLE**]

BOOK == CARTESIAN-PRODUCT
[subject: **SUBJECT**, copies: **COPIES**]

COPIES == SET [**COPY**]

COPY == CARTESIAN-PRODUCT
[num: **NUMBER**, state: **STATE**]

STATE == { "available", "checked-out"}

The types **AUTHOR**, **TITLE**, **SUBJECT** and **NUMBER** will not be defined in this specification: they belong to the environment specification which will be defined later during the construction process.

The next actions are to define formally all the functions in terms of predefined operations. We will study only one.

4.3 – Third action

Let us choose "add–a–copy" of a book which is a modifier of the **LIBRARY** type. The copy is characterized by 3 or 4 pieces of information: the author, the title of the book and the number of the copy. If the book is new in the library, it will be necessary to give its subject. The type **A–COPY** is thus the union of 2 different types **A–COPY–3** and **A–COPY–4**.

A–COPY == UNION [**A–COPY–3**, **A–COPY–4**]

where

A–COPY–3 == CARTESIAN-PRODUCT
[author: **AUTHOR**, title: **TITLE**, num: **NUMBER**]

A–COPY–4 == CARTESIAN-PRODUCT
[author:**AUTHOR**, title:**TITLE**,num:**NUMBER**,subject:**SUBJECT**]

See the description of add–a–copy in fig.2.

```
FUNCTION:   add-a-copy  : LIBRARY, A-COPY, NAME  --> LIBRARY
            add a copy of a book to a given library
```

```
data
  2    lib: Library given
  3   copy: A-COPY of a book to add
  4     un: user's NAME
result
  1   lib': LIBRARY

  6   bks': BOOKS of lib'
  7    bw': BORROWERS of lib'
  9   bks : BOOKS of  lib
 15    bw : BORROWERS of lib
 10   ust : user's STATUS
 11    a  : AUTHOR of the copy
 12    t  : TITLE of the copy
```

```
5  lib' = < bks', bw' >

8  bks' = [ bks
              if non (equal ("staff", ust)) ]
       or [ [ modify-book
              (bks, conv-A-COPY-3(copy))
              if in-library (bks, <a,t>) ]
       or [ add-book
              (bks, conv- A-COPY-4(copy))
              else ] else ]
19 bw'  = bw
13 bks  = books (lib)
16 bw   = borrow (lib)
14 ust  = type (acct (bw, un))
17 a    = author (copy)
18 t    = title  (copy)
```

fig.2: description of add-a-copy

Let us note that bks is not modified if the user does not belong to the "staff". It is enriched by just a new copy if the book is already in the library or by a new book with just one copy otherwise.
During the definition of the operation "add–a–copy", the system induced all the types of the objects introduced. The user gave only the formal and informal definitions. The numbers show the order in which the definitions may have been made. The profile and the informal definition of "add–a–copy" have been given during the first action.
The definition of "add–a–copy" introduces three new operations: modify–book, in–library and add–book.

The functions :

conv-A-COPY-3: **A–COPY** –-> **A–COPY–3**
conv-A-COPY-4: **A–COPY** –-> **A–COPY–4**

convert an object of the union type to the types **A–COPY–3** or **A–COPY–4**. They are predefined operations of the parameterized type "union".
The method suggests to define the three introduced operations. Their profile has been induced by the system. See their descriptions in fig.3, 4 and 5.

```
FUNCTION:   modify-book : BOOKS, A-COPY-3 --> BOOKS
            modify the component "copies" of a book by adding a copy
--------------------------------------------------------------------------
data                                    | bks' = mod (bks, <a,t>, <s,c'>)
    bks : BOOKS of a library            |
  copy3 : A-COPY-3                      | s    = subject (acct (bks, <a,t>))
result                                  | c'   = add-set (c, <n, "available">)
   bks' : BOOKS                         |
                                        | c    = copies (acct (bks, <a,t>))
      s : SUBJECT of the book           | a    = author (copy3)
     c' : COPIES of the book in bks'    | t    = title  (copy3)
      c : COPIES of the book in bks     | n    = number (copy3)
      a : AUTHOR of the book            |
      t : TITLE of the book             |
      n : NUMBER of the copy            |
```

fig.3: description of modify-book

```
FUNCTION:   in-library : BOOKS, BOOK-CHARACT --> BOOL
            is the given book in the library ?
--------------------------------------------------------------------------
data                                    |
    bks : BOOKS of a library            | in   = appt (bks, cb)
     cb : BOOK-CHARACT author and       |
          title of the given book       |
result                                  |
     in : BOOL true if the book         |
          belongs to bks                |
```

fig.4: description of in-library

```
FUNCTION:     add-book : BOOKS, A-COPY-4 --> BOOKS

              add a new book to the library
---------------------------------------------------------------------------
data                                     | bks' = insert (bks, <a,t>, <s,c>)
    bks : BOOKS of a library             |
  copy4 : A-COPY-4                       | c    = add-set (empty-set,
result                                   |                 <n, "available">)
   bks' : BOOKS bks with a new book      | a    = author(copy4)
                                         | t    = title(copy4)
      c : COPIES of the book in bks'     | n    = number(copy4)
      a : AUTHOR of the book             | s    = subject(copy4)
      t : TITLE of the book              |
      n : NUMBER of the copy             |
      s : SUBJECT of the book            |
```

fig.5: description of add-book

4.4 – Fourth action

All the definitions have been completely given so there is nothing to complete. Any type introduced has an invariant: they are equal to "true".

5 – CONCLUSION

The SACSO system allows the design of a specification in a non linear way. The user can introduce incomplete definitions at any time. Deductions and data type checkings are done according to the context. Semantics mistakes are indicated to the user, who can ignore them and correct them later on.

More generally, the design of the SACSO system takes into account the following concepts:

- modularity and reusability by means of a hierarchical specifications data base,
- ease of use with guidelines,
- flexibility of the environment and user-friendliness of the interface,
- validation by interpretation,
- rephrasing for a better understanding of the specification.

Several external representations are proposed:
- graphic representations in different formalisms (entity-relationship models [5], data-type visualisations [12]).
- textual ones.

The system is under development in Le-Lisp [4] (CEYX) [11] on a SM90.

Acknowledgements

This work is partially supported by A.D.I. and the C.N.R.S. (Greco de Programmation).
The authors are extremely grateful to professor Jean-Pierre Finance for initiating and supervising this research.
The development of SACSO is a team effort. The authors acknowledge the contribution of each team member: Eric Dubois, Habdelkrim Hadjadji and Agnes Valdenaire. Many discussions with Axel Van Lansweerde allowed us to have an external view of our work.

6 – BIBLIOGRAPHIE

[1] Belaid A., Levy N. : Spécification de types de données images et leur intégration dans des langages spécialisés. Bigre+Globule 45, Actes des journées AFCET Informatique, 1985.

[2] Bidoit M., Choppy C. ASSPEGIQUE : an Integreted Environment for Algebraic specifications. TAPSOFT proceedings, Berlin, pp.246-260, march 1985.

[3] Brodie M.L. and E. Silva : Active and Passive Component Modelling: ACM/PCM, In "Information System Design Methodologies: A Comparative Review", T.W. Olle, H.G. Sol, A.A. Verrijn-Stuart (eds), North-Holland, p.41-91, 1982.

[4] Chailloux, J.: Le_LISP de l'INRIA, Le Manuel de référence, 1984.

[5] Chen, P.P.: The Entity-Relationship Model: Towards a Unified View of Data, ACM TODS 1, pp. 9-36, 1976.

[6] Dubois E.: Cadre et méthode de spécification de systèmes d'information fondés sur les types de données, Thèse de Docteur-Ingénieur en Informatique, Nancy, 1984.

[7] Dubois, E., Levy, N. et Souquières, J. SACSO : Methodes et outils de construction de spécifications de systèmes, Actes CGL3, Versailles 1986.

[8] Dubois E., Van Lamsweerde A. Making specification processes explicit, 4th International Workshop on Software Specification and Design, Monterey, California, 1987.

[9] Futatsugi K., Goguen, J.A. and J. Jouannaud . Principles of OBJ2, ACM, Proc. Symp. Principles of Programming Languages, Assoc. for Computing Machinery, vol. 12, pp. 52-66, 1985.

[10] Guttag, J.V., and J.J. Horning . Report on the Larch Shared Language, Science of computer programming, vol 2, no 2, 1986.

[11] Hullot, J.M.: CEYX A multiformalism Programming Environment, Information Processing, 1983.

[12] Levy, N.: Outils d'Aide à la Construction et Transformation de Types Abstraits Algébriques, Thèse de troisième cycle, Université de Nancy-1, 1984.

[13] Ross, D.T. and K.G. Schoman. Structured Analysis for Requirements Definition, IEEE Trans. Soft Eng., SE 3(1), pp. 1-65, 1977.

[14] Swartout, W. , GIST English Generator, in Proceedings, AAAI-82, pp. 404-409, 1982.

[15] Teichroew, D. and Hershey, E.A. : A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems, IEEE Trans. Soft. Eng., SE-3(1), pp.41-48, 1977.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique