GRAPHIC SPECIFICATION OF ABSTRACT DATA TYPES

Pedro Rossel[1] **Ricardo Contreras**[2] **María Cecilia Bastarrica**[3]

Recibido el 5 de diciembre de 2003, aceptado el 7 de mayo de 2004

RESUMEN

La definición formal de requisitos de software usando especificaciones algebraicas tiene todas las ventajas de las especificaciones formales y su sólida base teórica. Este tipo de especificaciones es generalmente textual. La mayor parte de los lenguajes de especificación modernos tienen una representación gráfica para mejorar su usabilidad. Esto también es el caso de las especificaciones algebraicas. En este artículo presentamos una recopilación de las formas en que los tipos abstractos de datos pueden ser representados gráficamente usando especificaciones algebraicas, proponiendo una notación que incluye el conjunto de todas las facetas encontradas en la literatura. También mostramos un ejemplo de aplicación y algunos resultados experimentales de usar esta notación gráfica en la práctica.

Palabras Claves: Ingeniería de software, métodos formales, especificación algebraica, tipos abstractos de datos.

ABSTRACT

Formally specifying software requirements using algebraic specifications has all the advantages of formal specifications. This type of specifications is usually textual. Most modern specification languages have a graphical representation in an attempt to improve usability. This is also the case for algebraic specifications .Here we present a survey on how abstract data types are represented graphically. We propose a structure containing a superset of all elements surveyed. We also show an application example, and we report some experimental results when using this graphical representation.

Keywords: Software engineering, formal methods, algebraic specification, abstract data types, graphic language.

INTRODUCTION

The first and perhaps the most important step for the success of the software development process is the requirements specification. This is critical because it is common to introduce mistakes that are difficult and expensive to remove afterwards. There are several tasks in requirement treatment: requirement elicitation, writing these requirements as a consistent, complete and non-ambiguous document, and requirements analysis. Writing requirements is usually known as requirement specification; different notations have been proposed for this purpose ranging from unstructured and informal text to highly formal mathematical notations. The approach used in each case depends on the goals of the project and the available resources.

Informal specifications are easy to develop but they provide very little support for system analysis. In this way there is no certainty that specifications are complete, non-ambiguous, or that they fulfill all the desired characteristics of functional or non-functional requirements. However, most people use informal specifications because formal methods are still perceived as more difficult and expensive, even though it has been largely proven that this is not true [3], [6], [8], [13]. Formal specifications may require a longer specification time and expert personnel, but they allow and support specification analysis and they reduce testing effort.

Algebraic specifications are a formal way of specifying software systems as heterogeneous algebra's, i.e. a series of sets over which some operations are defined [10]. This technique has been developed for the last three decades and it is widely known.

Besides formality, specifications can also be classified as either textual or graphical. Traditionally, specifications were textual and this allowed a lot of fine grained detail to be specified. However, it is difficult to grasp the meaning of the whole system just looking at a

[1] Universidad Católica del Maule, Departamento de Computación e Informática. Av. San Miguel 3605 Talca, Chile, prossel@hualo.ucm.cl
[2] Universidad de Concepción, Departamento de Ingeniería Informática y Ciencias de la Computación. Edmundo Larenas 215 Concepción, Chile, rcontrer@inf.udec.cl
[3] Universidad de Chile, Departamento de Ciencias de la Computación. Blanco Encalada 2120 Santiago, Chile, cecilia@dcc.uchile.cl

textual specification, so efforts have been made to develop visual notations to overcome this difficulty. Graphical specifications are generally clearer than textual specifications, but they are also less scalable.

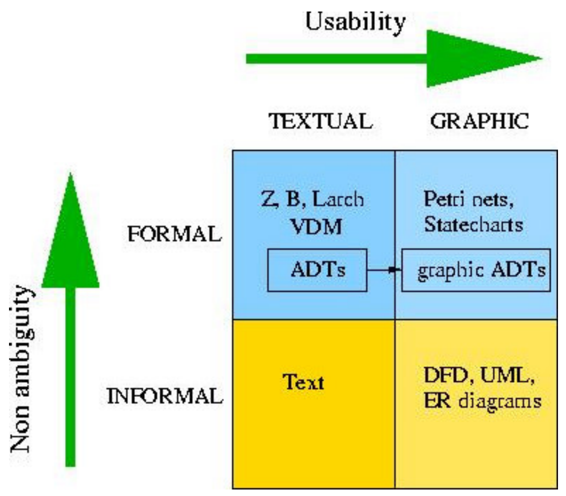


Fig. 1.- Classification of requirements specifications

The textual or graphical characteristic is orthogonal with respect to the formality of the specification. In this way we may have all combinations as shown in Fig. 1. Algebraic specifications are typically textual, but in recent years there has been an effort towards specifying them graphically. Therefore, we can count on all the theory behind algebraic specifications while gaining the usability provided by graphical notations.

Formal and Informal Specifications

There are two extreme ways of specifying software requirements: a complete formal specification and an informal specification. There also exist some intermediate representations that have shown to be useful in practice more as a means of communication between users and specialists than as requirement specification languages. These semiformal notations have also been used for documenting design, probably because there is no standard notation, neither for requirements nor for design. Table 1 is taken from [9] and it shows different categories in formality and examples in each category.

In order to have a formal requirements specification we need a formal specification language. A formal specification language provides a notation (syntax domain), an object universe (semantic domain), and the definition of precise rules about which object satisfies the specification [23].

Table 1.- Classification of notations

Informal	Semiformal	Formal
These techniques do not have complete sets of rules to constrain the models that can be created. Natural language (written text) and unstructured pictures are typical instances.	These techniques have a defined syntax. Typical instances are diagrammatic techniques with precise rules that specify conditions under which constructs are allowed and textual and graphical descriptions with limited checking facilities.	These techniques have rigorously defined syntax and semantics. There is an underlying theorical model against which a description expressed in a mathematical notation can be verified. Specification languages based on predicate logic are typical instances.
	Examples	
• Natural Language Specifications	• Data/Control Flow Diagrams • Entity-Relationship Diagrams • Use Case Diagrams	• Petri Nets • State Machines • VDM • Z

Textual and Graphical Specifications

A graphical specification is one whose elements are visual, rather that textual. Despite the linguistic parallel, however, graphical specifications are not easily related to their textual counterparts even in very simple problems. Important efforts are devoted in current research to this correspondence.

In many cases, people perceive information easier when it is presented in a graphic form. Given the comprehensive advantages of pictorial representations for people not trained in formal methods, some efforts have been made to establish a bridge between this kind of users and rigorous formal users. In [17], a language for specification of high level properties of real time systems is presented, based on temporal logic, and to hide obscure formal notation to users, authors employ a two-level approach allowing user-level expression s to have a graphical notation. In this work the idea of a bridge is implemented by considering this approach as a natural division of responsibilities. The use of templates is a partially constructed formula of the underlying temporal logic, similar to the templates we present for graphic notation.

Algebraic specifications are generally textual, but there have been some attempts to make them graphical in order to gain usability and to shorten the learning curve.

In a very interesting work, Neary and Woodward [18] address the problem of comparing the relative comprehensibility of one textual and two visual forms of algebraic specifications. This comparison uses a small specification presented to a test group, and they conclude that the graphical formalisms are easier to manage for novice users.

The Paper

In this paper we present an overview of algebraic specification of abstract data types (ADTs) and their graphical representation. We also show an application example. Fig. 2 sets the context of its contents.

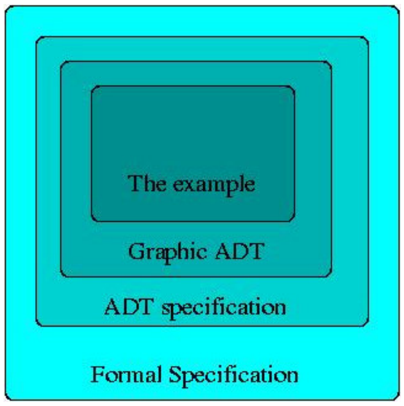


Fig. 2.- Paper context

The next Section presents how ADTs are formally specified using algebraic specifications; it also presents its most relevant characteristics. Then, we show how a simple information system can be specified using ADTs showing both a textual and a graphical version. Finally, we report the results of our experience using graphical ADTs and presents some conclusions.

ABSTRACT DATA TYPES SPECIFICATION

An abstract data type can be considered as a black box, where users can only see the syntax and semantics of its operations. ADT operarations can be classified as either constructors or inspectors [2]. Fig. 3 illustrates this feature.

Operations on the left hand side of the figure are used for constructing objects. Those on the right hand side, access and test, are used for querying information from an abstract object, and modification changes ADTs internal data [2].

Algebraic specification is a formal method used for software requirements specification. They are used since the 1970's as a technique for dealing with data structures in such a way that is independent from their implementation [6], providing a common description for data structures and operations that apply to them [2]. Algebraic specifications are natural for defining ADTs.

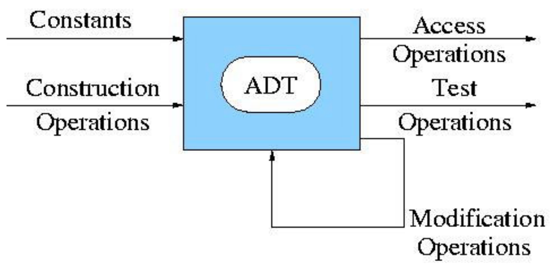


Fig. 3.- ADT Specification

The relationships among operations in an ADT are defined with equations generally called axioms, between terms built using both constants and the aforementioned operations. Each term represents an abstract object and an equation specifies that two terms represent the same object. Axioms are the semantic essence of algebraic specifications [2].

There are several approaches to algebraic specifications, which differ more in the form than in content, as it can be seen in [2], [4], [10], [12], [22], [24]. This article takes the perspective given by [4] and [12], adding some elements that support the characteristics of transaction processing systems [20]. We also enrich the set *E* to include not only axioms but also the whole semantics of the defined ADT.

Formally, an algebraic specification is identified in [6], [7] as a triple:

$$SPEC = \{S \,;\, OP \,;\, E\} \qquad (1)$$

where:

S: set of abstract data types,
OP: set of constant and operation declarations,
E: set of equations or axioms.

Notation

The structure of an algebraic ADT specification may contain the following elements:

ADT name: Beginning of the specification, where the ADT name is indicated.

IMPORT: Specifies other ADTs that can be used in the present specification.

TUPLES: Declares a locally defined data type, that may be composed of a series of fields. Types declared in TUPLES can be used in the current specification for defining variables, as well as in all ADTs that import the current ADT.

SYNTAX: Operation signatures are specified by including their name, domain and range.

SEMANTICS: Section formed by five subsections that define: VARIABLES, AXIOMS, SEQUENCES, DATA and PREDICATES. These parts give meaning to the operations declared in SYNTAX, and also state the constraints and conditions where they are valid. SEQUENCES are used to constrain the order in which operations must be accomplished; the operation on the left of the >> sign must be performed before the one on the right. VARIABLES is used to define local variables. DATA constrains the values that variables can take; interval values can be established, or constants with values of other variables previously restricted in DATA can be considered. AXIOMS are present in almost all algebraic specification languages. PREDICATES adds another level of constraining operation definition. AXIOMS are expressed through equations while PREDICATES establish certain properties through first order logic assertions.

END ADT: Indicates the end of the abstract data type specification.

The elements name, SYNTAX and AXIOMS are identical to those used in [4], [12], [22], [24]. The element IMPORT is used in [4], [12], [22] and TUPLES appears in [12]. SEMANTICS can be found in [4], but only in a restricted format. VARIABLES is used as in [4], [12], [24]. SEQUENCES can be found in [5], [24]. DATA is not used in any of the algebraic specification languages reviewed, but it further specifies and constrains the values variables can take. We use the syntax given by [24] for the PREDICATES. The formal syntax of each specification element is given in the Appendix.

These specification elements were chosen because they provide the expressiveness we need for specifying information systems characteristics, while still keeping the specification simple. All of them are formally defined and they allow us to specify both functional and non functional requirements.

A Graphic Specification for ADTs

We propose to use a graphic notation for some parts of the ADT specification including the same elements defined in the textual algebraic specification. The elements and structure of the graphic specification language are a synthesis of the proposals of [1], [5], [14], [15]. The textual algebraic specification language is isomorphic with the graphic language, so each element of one language has a correspondent element in the other one, and this correspondence is unique. Correspondence between both languages is shown in Table 2. This table also shows the syntax of graphical elements.

In the proposed graphic language, each ADT is defined by two diagrams: one general diagram including all described elements, but SEQUENCES, and another diagram including only the SEQUENCES diagrams. We chose to put SEQUENCES in another diagram to make the specification clearer; each operation may participate in several sequences, making it quite complex. For elements VARIABLES, DATA and PREDICATES we still use only the textual notation. All these features are illustrated in the example in Section **EXAMPLE: INFORMATION SYSTEMS**.

Characteristics

Graphic specifications have some virtues that are not very obvious in a textual specification, although they could be present. We here discuss some of the advantages of having both, a textual and a graphic dual specification.

Modularity

In the context of an ADT graphic specification, an ADT can be seen as a module, which in an isolated way ignores the details and specification of other modules. For a specific ADT, it is not necessary to make explicit the specification of those ADTs in IMPORT and TUPLES. However, in order to integrate a complete system, we should also consider the relationship among these modules.

In order to have a modular specification it is necessary to have a consistent specification within the ADT and with respect to other ADTs specification [20].

Hierarchy

Modularity is not enough to manage complexity. We need to be able to deal with abstraction and hierarchical specifications in a common approach. Our notation can deal with hierarchical specifications mainly through the IMPORT and TUPLE sections.

When an ADT2 is imported by an ADT1, ADT1 may use the operations and definitions of the ADT2, under the constraints that ADT1 may impose. If there are several hierarchical levels, the higher ones can use the operations defined in all lower level ADTs.

Scalability

Graphic specifications tend to be more intuitively understandable than textual specifications; however, as the complexity of the system grows, this characteristic decreases rapidly. Having a dual textual and graphic notation allows us to use graphic specifications for small and simple pieces, but we can still use the textual notation for very complex ADT specifications. Even though, this is generally accepted as true, we can use the modularity and hierarchical properties to keep all specifications as simple as possible, and continue using the graphic notation as the complete system specification grows in size and complexity.

Learning

In general, we can say that the brain does not code and internalize information unless it finds it significant. Relevant information is then retained through a memorization process [11], [25].

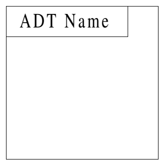
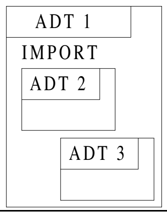
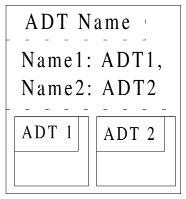
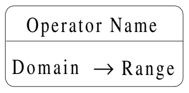
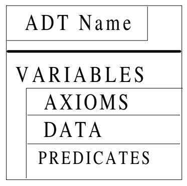
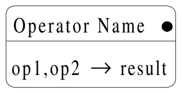
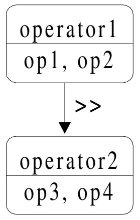
Long term memory is essential for the learning process [25]. This memory stores information in association networks organized in a hierarchical way [11]. Thus, using diagrams for graphical specification, with modularity and hierarchy characteristics as those presented in this article, makes it easier to have them coded in memory, and so it implies an easier use and a faster learning process.

EXAMPLE: INFORMATION SYSTEMS

In this work, we considered the specification of information systems, i.e. transaction processing systems. This kind of systems stands for most of the real world applications [16], [21], so it is an interesting application domain. Transaction processing systems include several activities, mainly data storage, data processing according to business rules, user interface, and data recovery; it may also add a programming interface to allow for extensibility. Fig. 4 illustrates the type of interactions expected for this kind of systems.

According to [19] and [22], information systems have requirements of diverse nature: functional, non-functional, hardware, etc. For this work we focussed on functional requirements and some of the possible non-functional requirements, such as operation precedence, data representation, and constraints on data values.

Table 2.- Correspondence between languages

Algebraic	Graphic	Description
ADT name END ADT		The complete ADT specification is between its name and the END ADT. The complete graphical specification is within the ADT box with the same name.
IMPORT		All imported ADTs are shown within the main box.
TUPLES		The name of the included tuples is included in the upper part of the box, as well as the ADT boxes where they belong.
SYNTAX		Operations defined for the ADT are show as round rectangles where the domain and the range of the operation are stated.
VARIABLES AXIOMS DATA PREDICATES		SEMANTICS is formed by the declaration of VARIABLES, AXIOMS, DATA, PREDICATES and SEQUENCES. VARIABLES, DATA and PREDICATES are declared almost in the same way in both languages. AXIOMS and SEQUENCES are defined graphically in the following rows.
AXIOMS		There is a box for each AXIOM, where the operations involved are stated in the form of equations.
SEQUENCES		There is an arrow with a >> symbol that specifies precedence between declared operations. All operations included must have been declared in SYNTAX. SEQUENCE diagrams are the only ones that are not included within the ADT box for clarity.

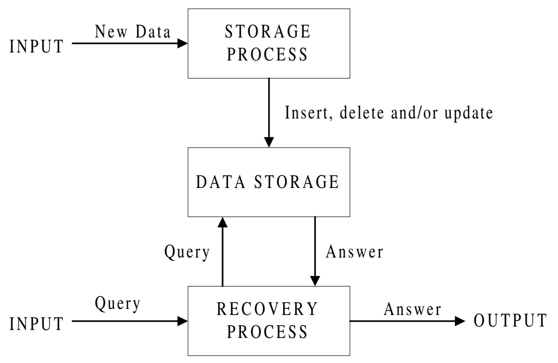


Fig. 4.- Data Storage and Recovery System

```
ADT Toart
IMPORT Book_Collection, Article_Collection
SYNTAX
  consult_book_by_author:
    Document_Author × Book_Collection → Book_Collection;
  consult article by author:
    Document_Author × Article_Collection → Article_Collection
SEMANTICS
  VARIABLES
    ∀ bo ∈ Book; bs ∈ Book_Collection;
    author ∈ Document_Author;
    ar ∈ Article; as ∈ Article_Collection
  AXIOMS
    consult_book_by _author(author, create_bc()) = create_bc();
    consult_article_by_author(author, create_ac()) = create_ac()
  SEQUENCES
    Book_Collection.enter_book(bo, bs) >>
      consult_book_by_author(author, bs1);
    Article_Collection.enter_article(ar, as) >>
      consult_article_by_author(author, as1)
  PREDICATES
    ∃ author ∈ Document_Author
     (∃ bo ∈ Book ( =(author, bo.Author) ⇒
     =(consult_book_by_author(author, enter_book(bo, bs)), bo))
     OR ∃ ar ∈ Article ( =(author, ar.Author) ⇒
     =(consult_article_by_author(author, enter_article(ar, as)), ar)))
END ADT
```

Fig. 5.- Textual Algebraic Specification of ADT Toart

In order to show the application of the proposed specification languages, a simple information system is specified. The system name is "TOART" (TO ARTicle) and it allows to register the papers and books used in the preparation of a scientific article. This system should have functionality for the insertion, deletion and query of books and articles. The system should be able to answer at least two questions (functional requirements):

- Which books did certain author write?
- Which articles did certain author write?

The textual algebraic specification of the system is formed by three ADTs: *Toart*, described in Fig. 5, *Book_Collection*, in Fig. 6, and *Article_Collection*, similar to *Book_Collection*. ADT *Toart* imports and uses the other two in its specification.

In ADT *Toart*, the operation *consult_book_by_author* returns the book or books that certain author has written. Similarly, *consult_article_by_author* returns the article or articles written by the author. In the case of ADT *Book_Collection*, the operation *enter_book* inserts a new book into the collection, and operation *eliminate_book* deletes a book from the collection.

Data types such as Boolean, Natural, Integer and String, are predefined as part of the algebraic specification language. *Document_Author* and *Document_Name* are imported by *Book_Collection* and *Article_Collection* and they are assumed to be defined elsewhere.

```
ADT Book_Collection
IMPORT Document_Author, Document_Name
TUPLES
  Book TUPLE OF
    Code : Natural,
    Author : Document_Author,
    Name : Document_Name,
    Editorial : String,
    Number_Edition : Natural,
    Publication_Year: Natural
SYNTAX
  create_bc: → Book_Collection;
  enter_book: Book ∈ Book_Collection → Book_Collection;
  eliminate_book: Book ∈ Book_Collection → Book_Collection
SEMANTICS
  VARIABLES
    ∀ x ∈ Book; y, y1 ∈ Book_Collection;
      author ∈ Document_Author
  AXIOMS
    eliminate_book(x, create_bc()) = create_bc();
    eliminate_book(x, enter_book(x, y)) = y
  SEQUENCES
    enter_book(x, y) >> eliminate_book(x, y1)
  DATA
    x.Number_Edition ≥ 1;
    x.Year_Edition ≥ 1900
  PREDICATES
    =(enter_book(x, enter_book(x, y)), enter_book(x, y));
    ∃ author ∈ Document_Author ∃ x ∈ Book ( =(x.Author, author)
      ⇒ ∃ y ∈ Books_Collection =(enter_book(x, y1), y) );
    ( =(enter_book(x, y1 ), y) ⇒
    ∃ author ∈ Document_Author =(x.Author, author) )
END ADT
```

Fig. 6.- Textual Algebraic Specification of ADT Book_Collection

The graphic specification of the system contains three diagrams, one for each ADT. The type diagram for ADT *Toart* is shown in Fig. 7. Sequence diagrams in Fig. 8 also belong to ADT *Toart*.

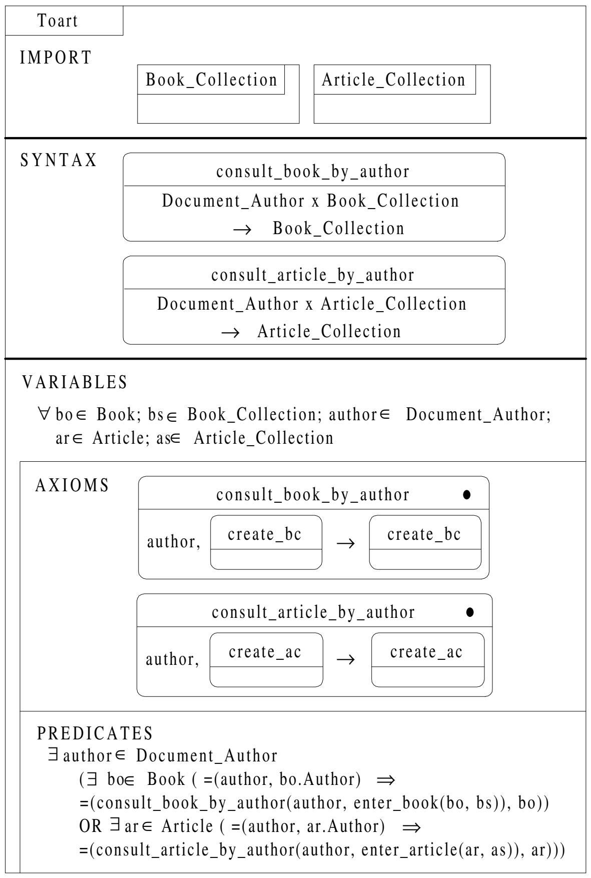

Toart
IMPORT
Book_Collection
Article_Collection
SYNTAX
consult_book_by_author
Document_Author x Book_Collection → Book_Collection
consult_article_by_author
Document_Author x Article_Collection → Article_Collection
VARIABLES
∀ bo∈ Book; bs∈ Book_Collection; author∈ Document_Author; ar∈ Article; as∈ Article_Collection
AXIOMS
consult_book_by_author ●
author, create_bc → create_bc
consult_article_by_author ●
author, create_ac → create_ac
PREDICATES
∃ author∈ Document_Author
(∃ bo∈ Book (=(author, bo.Author) ⇒
=(consult_book_by_author(author, enter_book(bo, bs)), bo))
OR ∃ ar∈ Article (=(author, ar.Author) ⇒
=(consult_article_by_author(author, enter_article(ar, as)), ar)))

Fig. 7.- Graphic Specification of ADT Toart

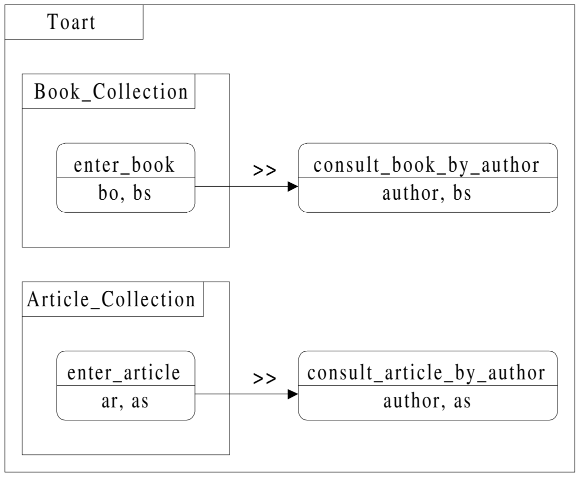


Fig. 8.- Graphic Specification of the Sequences in ADT Toart

Even though the system specified in the example is simple, it shows all the specification elements in the textual and graphic languages. We can intuitively compare the understandability of both approaches, where it is clear that it is easier to identify operations, axioms and sequences in the graphic notation, but we can also see that graphic notation is less scalable when the specification is long and complex.

CONCLUSIONS

Algebraic specifications are traditionally textual but they can also be specified graphically. Many approaches have been proposed in this direction. We presented a superset of all the features proposed in order to show the potential expressiveness of the technique, including those especially suited for specifying transactional systems.

As it happens with most graphical notations, graphical ADTs allow us to get a general understanding of a system easier than the textual specification for the same system. However, developing the most complex parts of the specification as axioms or predicates remains very close to text or completely textual, so there is no improvement.

We have had the experience of applying this two forms of algebraic specifications for two semesters for teaching formal methods to fourth year computer science students. They unanimously declared that the graphical form was easier to understand than its textual isomorphic specification. Even though this was not a controlled experiment, it gives us some intuition about the usability and understandability of graphical ADT specifications.

As expected, a graphical isomorphic representation of a textual algebraic specification can only improve understandability, even though it does not remove the complexity of developing the specification.

REFERENCES

[1] H. Ben-Abdallah, I. Lee, and J.Y. Choi; “GSCR: a Graphical Language with Algebraic Semantics for the Specification of Real-Time Systems”. Technical report, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, United States, 1995.

[2] E.A. Boiten, H.A. Partsch, D. Tuijnman, and N.Völker; “How to Produce Correct Software – An Introduction to Formal Specification and Program Development by Transformations”. The Computer Journal, Vol. 35, No. 6, pp.547 – 554, December 1992.

[3] J.P. Bowen and M.G. Hinchey; "Seven More Myths of Formal Methods". IEEE Software, Vol. 12, No. 4, pp. 34 – 40, July 1995.

[4] I.M. Bradley; "Notes on Algebraic Specifications". Information and Software Technology, Vol. 31, No. 7, pp. 357 – 365, September 1989.

[5] L.M.F. Carneiro, D.D. Cowan, and C.J.P. Lucena; "Introducing ADVcharts: a Visual Formalism for Describing Abstract Data Views". Technical report, Computer Science Departament & Computer Systems Group, University of Waterloo, Waterloo, Canada, 1993.

[6] H. Ehrig, B. Mahr, I. Classen, and F. Orejas; "Introduction to Algebraic Spe cification. Part 1: Formal Methods for Software Development". The Computer Journal, Vol. 35, No. 5, pp. 460 – 467, October 1992.

[7] H. Ehrig, B. Mahr, and F. Orejas; "Introduction to Algebraic Specification. Part 2: From Classical View to Foundations of System Specifications". The Computer Journal, Vol. 35, No.5, pp. 468 – 477, October 1992.

[8] K. Finney; "Mathematical Notation in Formal Specification. Too Difficult for the Masses?" IEEE Transactions on Software Engineering, Vol. 22, No. 2, pp. 158 – 159, February 1996.

[9] M.D. Fraser, K. Kumar, and V.K. Vaishnavi; "Strategies for Incorporating Formal Specifications in Software Development". Communications of the ACM, Vol. 37, No. 10, pp. 74 – 86, October 1994.

[10] C. Ghezzi, M. Jazayeri, and D. Mandrioli; "Fundamentals of Software Engineering". Prentice-Hall International Editions, 1991.

[11] T.L. Good and J. Brophy; "Contemporary Eductional Psychology". Addison Wesley, 5 edition, January 1995.

[12] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing; "Larch: Languages and Tools for Formal Specification". Springer-Verlag, 1993.

[13] A. Hall; "Seven Myths of Formal Methods". IEEE Software, Vol. 7, No. 5, pp. 11 – 19, September 1990.

[14] D. Harel; "On Visual Formalisms". Communications of the ACM", Vol. 31, No. 5, pp. 514 – 530, September 1988.

[15] J. B. Johnston; "The Contour Model of Block Structured Processes. SIGPLAN Notices, Vol. 6, No. 2, pp. 55 – 82, 1971.

[16] K. E. Kendall and J. E. Kendall; "System Analysis and Design". Prentice Hall, 4 edition, 1999.

[17] I. Lee and O. Sokolsky; "A Graphical Property Specification Language". In Proceedings of 2nd IEEE Workshop on High-Assurance Systems Engineering. IEEE Computer Society Press, 1997.

[18] D. Neary and M. Woodward; "An Experiment to Compare the Comprehensibility of Textual and Visual Forms of Algebraic Specifications". Journal of Visual Languages and Computing, Vol. 13, No. 2, pp. 149 – 175, April 2002.

[19] R. S. Pressman; "Software Engineering. A Practitioners Approach". McGraw-Hill, 5 edition, June 2000.

[20] P.O. Rossel; "A Language for Formal Requirements Specification with a Graphic Representation", (In Spanish). Master's thesis, Universidad de Concepción, Concepción, Chile, January 2000.

[21] J. A. Senn; "Analysis and Design of Information Systems". McGraw-Hill, 2 edition, January 1989.

[22] I. Sommerville; "Software Engineering". Addison-Wesley Publishing Company, 6 edition, 2000.

[23] J. M. Wing; "A Specifier's Introduction to Formal Methods". IEEE Computer, Vol. 23, No. 9, pp. 8 – 24, September 1990.

[24] J. Woodcock and M. Loomes; "Software Engineering Mathematics". Addison-Wesley Publishing Company, 1989.

[25] A. E. Woolfolk; "Educational Psychology". Allyn & Bacon, 7 edition, 1998.

APPENDIX: FORMAL SYNTAX

The formal syntax of the algebraic specification has been defined in [20], and we summarize it here. Figs. 9, 10, and 11 show this definition.

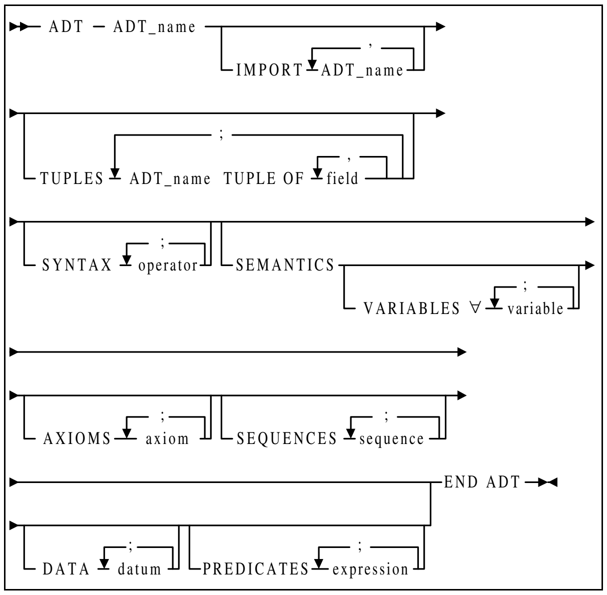


Fig. 9.- Formal definition of algebraic specification syntax (a)

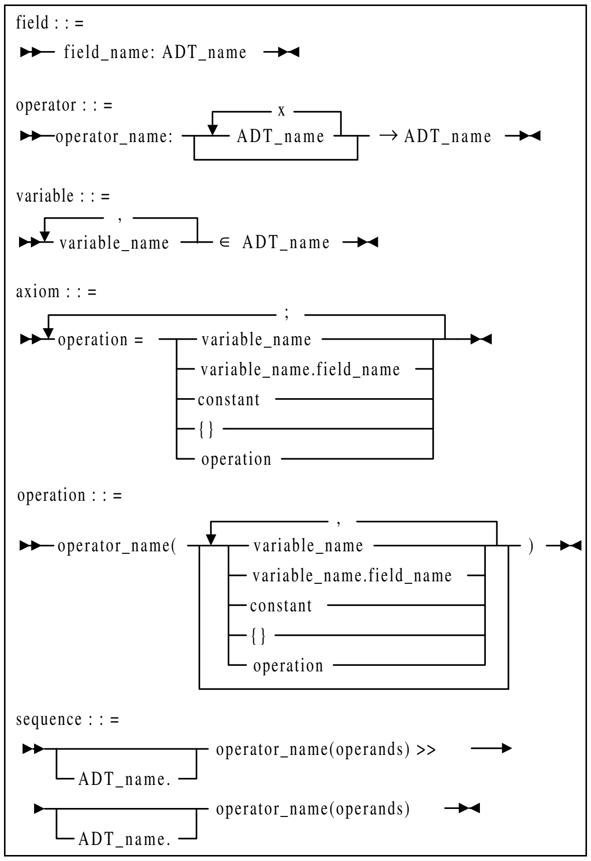


Fig. 10.- Formal definition of algebraic specification syntax (b)

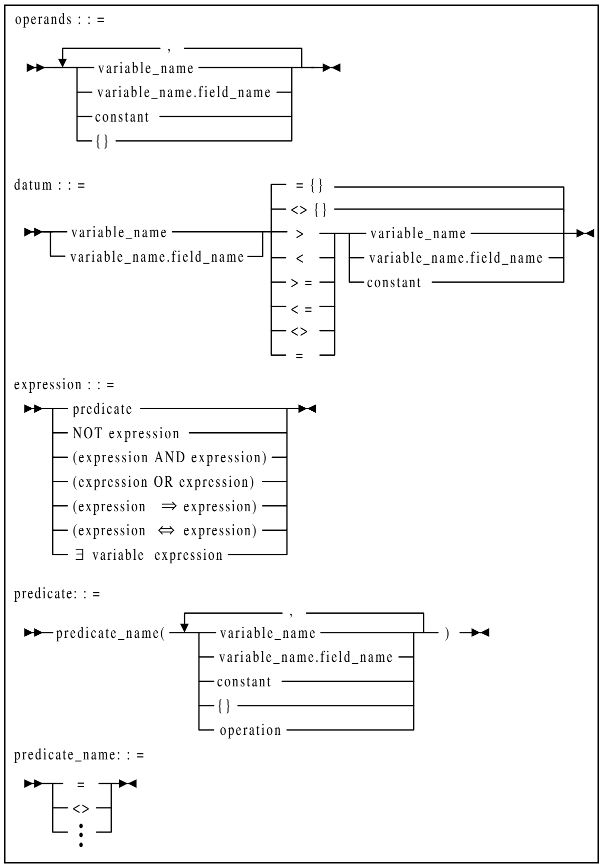


Fig. 11.- Formal definition of algebraic specification syntax (c)