Elaborating, Structuring and Expressing Formal Requirements of Composite Systems

Eric Dubois, Philippe Du Bois and André Rifaut

Institut d'Informatique, Facultés Universitaires de Namur
21 rue Grandgagnage, B-5000 Namur (Belgium)
E-mail: {edu,pdu}@info.fundp.ac.be

Abstract . In this paper, we propose a formal specification language supporting activities performed during the initial requirements engineering phase of the software lifecycle. During this phase, those activities include (i) the elicitation and the capture of the initial description of a given problem, (ii) the expression of requirements associated with a 'composite system' (i.e. a system including manual procedures, hardware devices and software components interacting together) providing a solution to the original problem and (iii) the organization of the requirements document in order to enhance its readibility and to promote its maintenance and reusability.

Keywords : requirements engineering, composite systems, first-order and temporal logic, structuring mechanisms.

1 Introduction

Requirements Engineering (or Requirements Analysis) is now widely recognized as a critical activity in the context of information systems development. Despite this fact, only a few methods and tools, providing a real guidance, supporting verifications and validations, are emerging and used in an industrial environment. We feel that a part of this difficulty is due to the lack of adequacy of the current requirements specification languages. In this paper, we present and illustrate a new specification language based on some recent ideas investigated within the framework of an Esprit II project (called *Icarus*) entirely devoted to Requirements Engineering.

We feel that the first quality of a requirements specification language is its *expressiveness*. One should keep in mind that requirements should not only address the description of the functional behavior of the computerized system but should also encompass the description of a *composite system* (like, e.g., requirements on manual procedures to be installed or on a specific hardware and/or device to be used). Thereby, a suitable language should support the expression of computer artefacts as well as the "natural" (i.e. without any computer bias) description of statements related to the real-world entities. The language, presented in this paper, devotes a particular attention to the expression of *accuracy* requirements (like "the stock's quantities recorded on the computer should reflect the real-world stock's quantities with some delta"), and *real-time* requirements (like "this report has to be issued by the computer within 5 seconds"). On top of its expressiveness, we plead that

a requirements specification language should be *formal* enough so that a number of analysis activities (not only restricted to syntactic checks) can be supported by the availability of (i) rigorous *rules of interpretation* defining precisely the language constructs and (ii) a set of *deductive rules* allowing to reason about pieces of specifications. In Sect. 2, we present the basic features of our language with respect to its expressiveness and its associated formal framework. In Sect. 3, we illustrate them on an excerpt of a library case study.

The requirements engineering activity is a complex activity which starts from incompletely defined wishes expressed by several customers about a desired complex composite system. One could imagine that this activity consists only in the transcription of the customers' wishes in a requirements document. This is far from true because those wishes are in most cases imprecise, incomplete, ill-structured and even inconsistent. Thereby, it is essential to have some methodological guidance in the *progressive* elaboration of the requirements document (see e.g. [Fin89] [Fea89]). For some years, we are experimenting an incremental development strategy (i) starting from a simple specification of the desired external behavior exhibited by the system to be developed and (ii) gradually moving towards a more complex specification of the internal behavior of the system considered as a *composite* one, viz considering the responsibilities associated with the different components as well as the nature of the interfaces existing between them [Dub88b] [Dub90]. In Sect. 4, we illustrate the capability of our language for supporting this incremental elaboration of requirements for composite systems. In particular, we will suggest how the global specification associated with the library case-study can be refined in a more detailed specification where two library's components are identified.

Usually, an important part of the work performed by the requirements analysts relies on the organization of the whole requirements document. This is particularly true for complex descriptions including several thousands of requirements statements. Thereby, we feel essential that the requirements specification language be equipped with *structuring mechanisms* making possible the requirements document be organized into specification units with well-defined inter-units relationships. Structuring mechanisms available in the language are introduced in Sect. 2 and their use illustrated in the other sections. In particular, in Sect. 5, we suggest how the use of the parameterization mechanism may support a requirements elaboration mechanism [Reu89] [Pro89] based on the identification of generic concepts for a given *problem domain* and on their tailoring to the needs of the requirements expressed for a *specific application*. In this section, the approach is illustrated by considering the requirements expressed about one of the library components in terms of basic concepts related to a *resource allocation* problem domain.

Finally, after a short comparison of our language with some other existing approaches, Sect. 6 concludes this paper with some directions for future work.

2 Overview of the Language

As we pointed out in the Introduction, a suitable language for the Requirements Engineering activity should be a general customizable language supporting an incremental elaboration and analysis of the requirements document. To this end, the

language that we have developed has been designed according to three essential features : expressiveness, structuring mechanisms and formality.

2.1 Expressiveness

The language must be sufficiently rich to support some "natural" mapping between all kinds of things of interest and the various language concepts being available. In other words, requirements specification should remain a problem definition activity, not a coding task. In particular :

- The language supports the use of different and possibly mixed styles of specification. At the Requirements Engineering level, we have experimented that there are numerous properties which are not of an *algorithmic* nature, i.e. cannot be expressed in terms of successive transformations applied on arguments to produce results. This is why a more *declarative* style is also supported in the language.
- Immutable values, such as numbers or strings, are not rich enough for modelling 'real-world' dynamic systems. Observations associated with *the state of such systems are intrinsically time-varying.* Thereby, in the language, an important distinction is introduced between *data type* (i.e. immutable values) and *type clusters* (used for recording time-varying states observations).
- With respect to clusters, the language does not only support the expression of constraints on admissible states or on the transition between two successive states (the usual *pre/post*) but also on the *ordering of states.* These constraints make possible to refer future states (like e.g. 'if this property holds in this state, then it holds in all future ones') as well as previous states. Furthermore, *real-time constraints* like 'this property is true during 3 minutes' can also be expressed.
- During the incremental elaboration of a requirements specification, it is usual to deal with *incomplete requirements* (viz. requirements which are in an intermediate – non-finished – stage). The language permits to retain this information so that it may direct the acquisition of further requirements. Furthermore, the semantics of the language supports the handling of incomplete requirements.

2.2 Structuring Mechanisms

In most cases, the specification of requirements results in large documents where complex interactions exist between different pieces of descriptions. Such documents should be *organized* into separate units which can be combined in a controllable way to yield the complete specification. Moreover, structuring mechanisms are also essential to support the *reuse* of specification's components and the *maintenance* of the requirements document. In our language, we have identified four structuring mechanisms :

- A natural part of the specification process includes the identification of various things of interest sharing some common characteristics (like, e.g., the set of admissible values for an account number or the set of admissible behaviors (states)

for a library system). In the language, this activity is modelled by the introduction of *types* which follow the Object-Oriented paradigm [Fia91] [Jun91] by packaging a set of properties together. The language also includes a set of built-in predefined types associated with usual data types (integers, strings, booleans, etc) and with combinators (cartesian product, sequence, set, bag and table) for putting together data specifications.

- The language supports the introduction of *parameterized* type clusters and data types allowing to factor out common properties shared by individual elements (playing the role of instantiated units) at different locations of the requirements document, thus ensuring a better readability of the document. It should be also noted that parameterization is one of the most useful structuring mechanism with respect to reusability.
- Two *inheritance* mechanisms are available in the language. The first one (corresponding to the usual mechanism referred in the literature) is based on the introduction of *sub-types* (like, e.g., 'Employee is a Person'). The second mechanism is less classical and is more syntactic in the sense that it is based on a *cut and paste* of an existing specification piece (like, e.g. 'Properties of an Airbus are a copy of the properties of a Boeing'). The inheritance mechanism supports the definition of a new specification by inheriting another specification and by extending and/or restricting it.
- The 'scoping' mechanism controls the visibility of names in a large document. This is particularly helpful when multiple specifiers have to integrate their specifications into a coherent document. This mechanism is not illustrated in the rest of this paper because no name clashes occur in the small case study considered here.

2.3 Formality

A formal language depends on the availability of rigorous *rules of interpretation* which guarantee the absence of ambiguity. Besides, rules of deductive inference are needed to make possible the derivation of new sentences from given ones. The deductive power supports the analysis, the validation (e.g. with the generation of a prototype) of formal specifications but also gives a handle for a rigorous investigation of the requirements engineering process (see e.g. [Joh88] [Dub91c]).

The choice of a adequate formal framework for our language has been influenced by two conclusions following the study of existing formal specification languages:

- At the expressiveness level, the use of a first-order *logic* framework seems to be a reasonable basis because of the variety and the naturalness of constraints that can be expressed. Moreover, some specific *modal* extensions are interesting because of the availability of specific deductive rules which makes possible to reason on specific concepts and the conciseness reached in the expression of constraints. Examples are Infolog [Ser80] and Erae [Dub88] based on a 'temporal logic' (i.e. a logic dealing with histories) and Mal [Fin87] and [Dub91c] based on a 'deontic logic' (i.e. a logic dealing with actions and agents).
- At the structuring mechanisms level, only a few syntactical structuring mechanisms are available for *first-order logical* frameworks but it appears that, within

the *algebraic* framework, a number of semantic relationships can be envisaged (e.g. parameterization, inheritance, etc). However, in most cases, these mechanisms have been investigated within the framework of an 'equational' logic (i.e. a subset of first order logic), not sufficiently expressive for our requirements modelling purposes.

For our language, the conclusion of these experiences led to the choice of the so-called *loose semantics* formal framework [Ehr90] [Ore89] based on an algebraic framework (with the usual structuring relationships) but where the properties can be expressed in terms of a set of typed first-order formulas.

3 Writing and Structuring Expressive Formal Requirements

The objective of this section is to illustrate the expressiveness and the formality of our language sketched in Sect. 2. To this end and all along the rest of this paper, we will refer to a simplified version of a library problem. The informal description of this case study is the following one:

> We consider a library where users may issue requests for books belonging to the library and become borrowers of these books. The following rules are:
> - the set of books owned by the library and the set of library's users are considered fixed in time. Books can be either available on shelves or borrowed by users. Users are identified by their name and surname;
> - requests are issued by users for books and remained pending up to their satisfaction that should occur without unecessary delay;
> - books can be borrowed by users for a period of maximum 30 days. The borrowing of a book by a user is possible only if this user has issued a request for this book.

3.1 A First Specification

To help in the elicitation and the understanding of a problem, we have found useful to express it in terms of an ERA diagram complemented with constraints (this approach was inspired by some previous experiences made by the authors using the ERAE language [Dub88] [Dub91a]).

Figure 1 proposes a graphical ERA diagram associated with the library case study where:

- Books and Users are considered as sets of entities,
- Requests and Borrowings are relationship between users and books,
- Name and Surname are attributes identifying users.

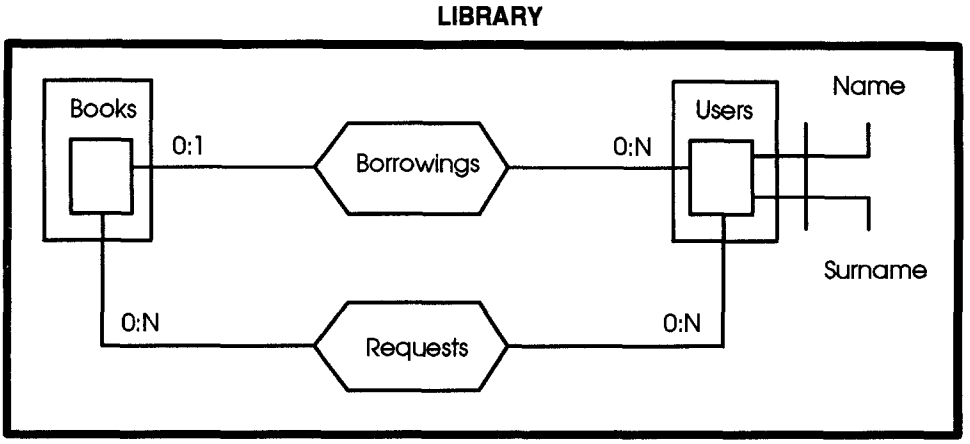

Fig. 1

Hereafter, in Fig. 2, the formal specification of the requirements associated with the library case study is presented.

Type Cluster LIBRARY

State inspection operations

Fixed $Books : BOOK$
$Users : USER$

Varying $Borrowings : BOOK \times USER$
$Requests : BOOK \times USER$

Constraints

* Constraints (1), (2), (3) are connectivity constraints derived from the ERA diagram

* (1) Borrowings are restricted to books and users of the library
$Borrowings(b, u) \Rightarrow Books(b) \wedge Users(u)$
* (2) Requests are restricted to books and users of the library
$Requests(b, u) \Rightarrow Books(b) \wedge Users(u)$
* (3) A book may be borrowed by at most one user
$Borrowings(b, u1) \wedge Borrowings(b, u2) \Rightarrow u1 = u2$

* (4) A user cannot issue a request for book he/she borrows
$Requests(b, u) \Rightarrow \neg Borrowings(b, u)$

* (5) Books available on shelves and for which requests are pending are allocated without unecessary delay
$(\nexists u : Borrowings(b, u)) \wedge (\exists u' : Requests(b, u')) \Rightarrow \bigcirc (\exists u'' : Borrowings(b, u''))$
* It should be noted that no particular assumption is taken about which pending request is served first

* (6) A book can only be allocated to a waiting user
$Borrowings(b, u) \wedge \ominus (\neg Borrowings(b, u)) \Rightarrow \ominus (Requests(b, u))$

* (7) Borrowed books are returned within 30 days
$Borrowings(b, u) \Rightarrow \Diamond_{\leq 30 days}(\neg Borrowings(b, u))$

* (8) A waiting user is waiting until he/she borrows the book he/she is waiting for
$Requests(b, u) \Rightarrow \bigcirc (Requests(b, u) \vee Borrowings(b, u))$

•••

Data Type BOOK

Data Type USER

is $CP[Name : STRING, Surname : STRING]$

Fig. 2

The following features should be noted:

- In the specification, properties are grouped in *type definitions*, respectively associated with the specification of the library *cluster* and with the specification of *data* values.
  Data types are associated with the definition of immutable values, i.e. values which are not supposed to change with time. In our example, data types are associated with BOOK and USER. In the case of USER, the use of the cartesian product (CP) combinator precises that a user is identified with two string values (respectively associated with the name and the surname of a user).
  By contrast, if we consider the set of users belonging to the library case study, data are not sufficient for modelling it since this set will typically vary with time. This is why the library is described as a *type cluster*. In our example, the type cluster is used to characterize the set of possible histories modelling all the admissible behaviors of the library. A history is a discrete sequence of states, each labeled by a time value which increases all along the history. The state can be inspected through a set of so-called *state inspection operations* which are used to return all relevant informations about the system at that moment. These inspections are modelled in terms of data values. In the above example, there are four inspection operations (*Books, Users, Borrowings, Requests*) for the different library state components. It should be noted that these components can be denoted state *varying* (e.g. *Borrowings*) or *fixed* (e.g. *Books*).

- In the specification, there are data types (e.g. STRING) and combinators (e.g. cartesian product) which are built-in in the language. Using these types makes possible to inherit from their predefined associated operations. For example, one may write "Name(u)" to access the name of a user.
- The purpose of first order constraints is to identify the set of admissible histories. Constraints are written according to the usual rules of strongly typed first order logic. In particular, they are formed by means of logical connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies), $\Leftrightarrow$ (if and only if), $\forall$ (for all), $\exists$ (exists). Moreover, it should be noted that the outermost universal quantification of formulas can be omitted. The rule is that any variable, which is not in the scope of a quantifier, is universally quantified outside of the formula. There are different kinds of constraints.
  1. There are constraints which act as *invariants*, i.e. which are true in all states of the system. This is the case in our example for constraints (1) to (4).
  2. There are constraints on the evolution of the system. Writing these constraints require to be able to refer more than one state at a time. This is done in our language by using additional temporal connectives which are prefixing statements to be interpreted in different states. The following table introduces these operators (inspired from temporal logic, see e.g. [Ser80] [Dub91a]) and their intuitive meaning (ϕ and ψ are statements):

$\bigcirc\,\phi$	ϕ is true in the next state
$\ominus\,\phi$	ϕ is true in the previous state
$\Diamond\,\phi$	ϕ is true sometimes in the future (including the present)
$\Diamond\!\!\!-\,\phi$	ϕ is true sometimes in the past (including the present)
$\Box\,\phi$	ϕ is always true in the future (including the present)
$\boxminus\,\phi$	ϕ is always true in the past (including the present)
$\phi\,\mathcal{U}\,\psi$	ϕ is true from the present until ψ is true (strict)
$\phi\,\mathcal{S}\,\psi$	ϕ is true back from the present since ψ was true (strict)

     Constraints can involve two successive states (like in constraints (5), (6) and (8)) or states which are further apart (like in constraint (7))
  3. There are constraints related to the expression of real-time properties. There are needed to express delays or time-outs. For instance, in the library system, the constraint (7) uses an extension of the temporal $\Diamond$ operator subscripted by a time period. This time period is made precise by using predefined functions that can be used to model the usual time units: *Sec, Min, Hours* and *Days*.
- Finally, it should be noted the use of a '•••' notation in the library's specification. This symbol is used to model that the set of properties which have been modelled here is not complete, viz. that the specification document is in an intermediate stage of its elaboration. This means that the analyst needs to further discuss with the customer to elicitate additional requirements. But even when the '•••' occurs, the semantics of our language supports formal checks and analysis on the requirements document and the deduction of properties about the system's behavior.

3.2 Organizing the Specification

In the previous sub-section, we proposed a first elicitation of the library problem in terms of an ERA diagram. A next step in the requirements process is to achieve a better structured version of the original specification in order to promote its readibility, maintenance and reusability. To this end, the language offers number of mechanisms for organizing complex requirements descriptions. Hereafter, in Fig. 3, we present a better structured version of the library's specification introduced in the previous sub-section.

Type Cluster LIBRARY

State inspection operations

Fixed $Books : BOOK$
$Users : USER$

Varying $Borrowings : BOOKUSER$
$Requests : BOOKUSER$

Constraints

* Constraints (1), (2), (3), (4) and (7) are similar to Fig. 2

* (5) Books available on shelves and for which requests are pending are allocated without unecessary delay
$OnShelves(b) \wedge (\exists u : Requests(b,u)) \Rightarrow \bigcirc (\exists u' : Borrowings(b,u'))$

* (6) A book can only be allocated to a waiting user
$CheckingOuts(b,u) \Rightarrow \ominus (Requests(b,u))$

* (8) A waiting user is waiting until he/she borrows the book he/she is waiting for
$\neg CheckingOut(b,u) \wedge \ominus (Requests(b,u)) \Rightarrow Requests(b,u)$

•••

Auxiliary operations

Varying $OnShelves : BOOK \rightarrow BOOLEAN$
asserts
$OnShelves(b) \Leftrightarrow \neg In(b, Borrowings)$
* A book is on-shelf if it is not borrowed

$CheckingOut : BOOK \times USER \rightarrow BOOLEAN$
asserts
$CheckingOut(b,u)$
$\Leftrightarrow Borrowings(b,u) \wedge \ominus (\neg Borrowings(b,u))$
* A user is checking out a book if he/she is borrowing a book that was not borrowed in the previous state

Data Type BOOK

Data Type USER

is $CP[Name : STRING, Surname : STRING]$

Data Type BOOKUSER

is $CP[BOOK, USER]$

Operations

$In : BOOK \times SET[BOOKUSER] \rightarrow BOOLEAN$
asserts
$In(b, bu) \Leftrightarrow (\exists u : < b, u > \in bu)$
* Test the membership of a book to the relationship between books and users

$In : USER \times SET[BOOKUSER] \rightarrow BOOLEAN$
asserts
$In(u, bu) \Leftrightarrow (\exists b : < b, u > \in bu)$
* Test the membership of a user to the relationship between books and users

Fig. 3

This new organization has been achieved by using two mechanisms.

- The first mechanism is based on the introduction of *auxiliary intermediate operations* inside the type cluster. These new operations help to better structure and clarify the set of constraints. Each operation is defined in terms of an assertion specifying the relationship that must hold between arguments in the domain and results in the range.
- The second mechanism proposes an organization of the new introduced operations following the O-O paradigm. To this end, the language offers the possibility to define additional *auxiliary types* on top of already existing ones by using the predefined types constructors. In our example, a new intermediate data type cluster (called BOOKUSER) has been introduced. This cluster associates two new intermediate operations with a new intermediate type defined as a set of tuples.

4 Specifying Composite Systems

In some recent work, we are investigating an incremental elaboration *process* starting with the specification of a problem considered as a monolithic one and gradually moving towards the description of a more complex composite system [Dub88b] [Dub91c]. More precisely, we propose:

1. to express specifications about the **goals** assigned to the system to be installed and to its environment (considered as a whole – black-box approach –) to be developed;
2. to express specifications about **finer requirements** that are assigned to the different components (e.g. a software component, a hardware piece, or a manual procedure) and to verify that the set of all requirements attached to individual components meet the goals originally introduced.

To support this approach, we need a language where, at some stage of the process, it is possible to capture descriptions of composite systems rather than to consider the system as a monolithic one. Thereby, we have extended our language so that it offers mechanisms for combining several single components together and for specifying properties characterizing the individual behavior of each separate component as well as the interactions taking place between the different components.

Let us refer to the library problem again. Up to now (see Fig. 2), we have considered this system as a monolithic one. At a more finer level, one could imagine a more detailed organization making the distinction between:

- The environment of the library system composed of users issuing requests and having books in their possession.
- The system itself where a librarian is in charge of managing the set of books on shelves in the library as well as the allocation of these available books according to the requests issued.
- The introduction of the system and its environment goes with the identification of the nature of the communications that should take place between them. A priori, each component only has access to the informations that it manages. Clearly some *communication medium* (i.e. *interface* [Doe91]) has to be installed between the different components so that a component may offer some services to another one.

In Fig. 4, we introduce a graphical ERA representation of the new situation where the system (LIBRARY-S) and its environment (LIBRARY-E) are made distinct. It should be noted that:

1. the library system is embedded in its environment. This is due to the fact that we want to indicate that changes occurring in the environment (i.e. requests and returns made by users) should not be constrained by the system behaviour.
2. we introduce events (graphically depicted with ovals) to describe the nature of the *interface* between the system and its environment. A *visibility* relationship (graphically depicted with arrows) makes precise the perception of a component with respect to events happening in the other component.

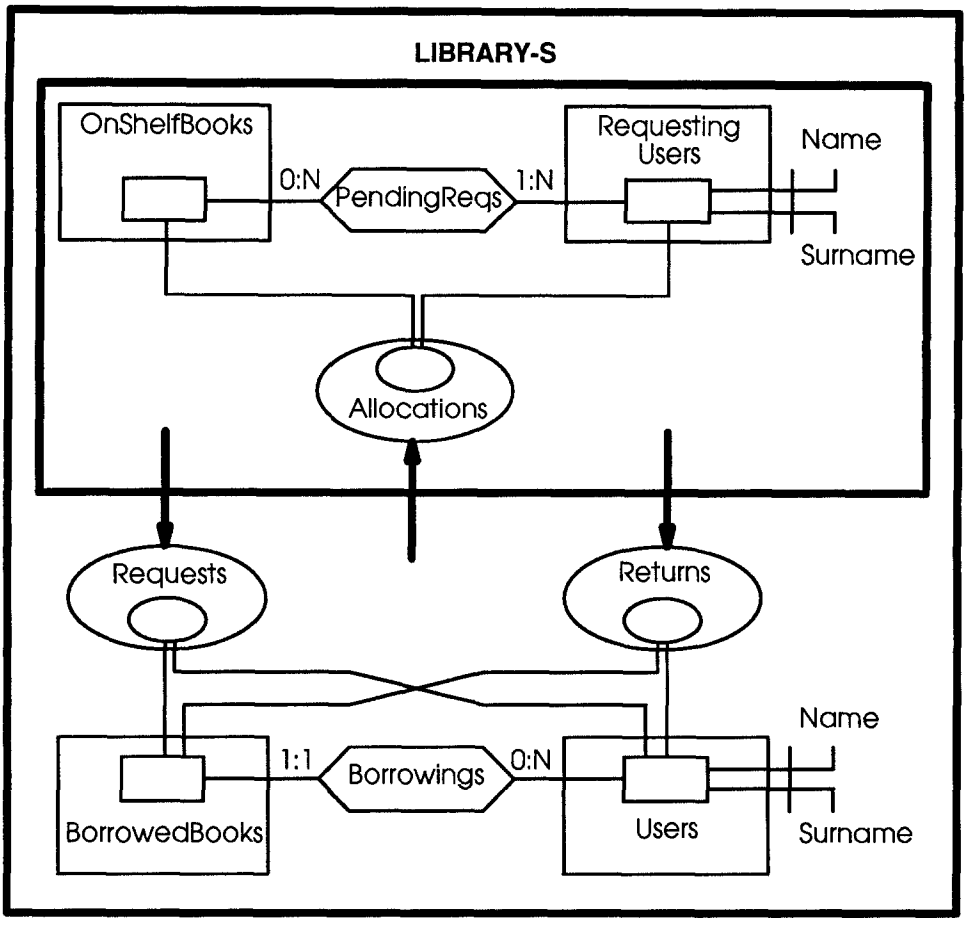

Figure 4

In Fig. 5, we present the formal specification associated with the graphical representation presented above.

Type Cluster LIBRARY

Composed of

LIB-S is SYSTEM
LIB-E is ENVIRONMENT

Type Cluster LIB-S

Interface events

$Allocations : BOOK \times USER$

State inspection operations

Varying $OnShelfBooks : BOOK$
$RequestingUsers : USER$
$PendingReqs : BOOK \times USER$

Constraints

* The three following constraints are derived from the ERA diagram

* A check-out is related to an on-shelf book and to a requesting user
$Allocations(b, u) \Rightarrow OnShelfBooks(b) \wedge RequestingUsers(u)$
* A pending request exists between a book of the library (i.e. which is or has been on-shelf) and a requesting user
$PendingReqs(b, u) \Rightarrow \Diamond (OnShelfBooks(b)) \wedge RequestingUsers(u)$
* A user is requesting if and only if he/she has at least one pending request
$RequestingUsers(u) \Leftrightarrow \exists b : PendingReqs(b, u)$

* A request is pending from the time it is issued in the environment and until the book is allocated
$Requests(b, u) \Rightarrow \bigcirc (PendingReqs(b, u)\ \mathcal{U}\ Allocations(b, u))$

* A pending request and a book on shelf are removed if and only if the corresponding book is allocated
$Allocations(b, u) \Rightarrow \bigcirc (\neg PendingReqs(b, u) \wedge \neg Onshelf Books(b))$

* A book can only be allocated to a waiting user
$Allocations(b, u) \Rightarrow \ominus (PendingReqs(b, u))$

* Books available on shelves and for which requests are pending are allocated without unecessary delay
$(\exists b : OnShelfBooks(b)) \wedge (\exists u : PendingReqs(b, u))$
$\Rightarrow \bigcirc (\exists u' : Allocations(b, u'))$

Type Cluster LIB-E

Interface events

$Requests: BOOK \times USER$
$Returns: BOOK \times USER$

State inspection operations

Fixed $Users: USER$

Varying $BorrowedBooks: BOOK$
$Borrowings: BOOK \times USER$

Constraints

* The five following constraints are derived from the ERA diagram

* A request is issued by a user of the library
$Requests(b, u) \Rightarrow Users(u)$
* A return happens for a borrowed book of the library and is performed by a user of the library
$Returns(b, u) \Rightarrow BorrowedBooks(b) \wedge Users(u)$
* A borrowing links a borrowed book of the library to a user of the library
$Borrowings(b, u) \Rightarrow BorrowedBooks(b) \wedge Users(u)$
* A book is borrowed if and only if it is linked by a borrowing
$BorrowedBooks(b) \Leftrightarrow \exists u: Borrowings(b, u)$
* A book may be borrowed by at most one user
$Borrowings(b, u1) \wedge Borrowings(b, u2) \Rightarrow u1 = u2$

* A book is borrowed from its allocation to a user until it is returned to the library
$Allocations(b, u) \Rightarrow \bigcirc (Borrowings(b, u)\ \mathcal{U}\ Returns(b, u))$

* A borrowing and a borrowed book are removed if and only if the corresponding book is returned
$Returns(b, u) \Rightarrow \bigcirc (\neg Borrowings(b, u) \wedge \neg BorrowedBooks(b))$
$\ominus (Borrowings(b, u)) \wedge Returns(b, u) \Rightarrow Borrowings(b, u)$
$\ominus (BorrowedBooks(b)) \wedge (\not\exists u: Returns(b, u)) \Rightarrow BorrowedBooks(b)$

* A book can be returned only if it was borrowed
$Returns(b, u) \Rightarrow \ominus (Borrowings(b, u))$

* Borrowed books are returned within 30 days
$Allocations(b, u) \Rightarrow \Diamond_{\leq 30 days} (Returns(b, u))$

* In the initial state, the set of borrowed books is empty
$\neg Empty?(BorrowedBooks) \Rightarrow \Diamond\, Empty?(BorrowedBooks)$

Data Type USER

is $CP[Name : STRING, Surname : STRING]$

Data Type BOOK

Fig. 5

Finally, due to the formality of our language, it would be possible to give a formal proof that the joint behavior of the two components meet the behavior of the original system. For example, in our first specification (Fig. 2), the set of books was declared fixed in time. This property is preserved by the combination of the behaviors of the two components of Fig. 5 from which it results that:

$$Books = BorrowedBooks \cup OnShelfBooks$$

(i.e. the set of books of the library is the union of both sets of borrowed and on-shelf books).

5 Capturing Problem Domain Knowledge

In Sect. 3, we have suggested how the use of *types* may provide help in the organization of a large requirements document. However, we have experimented that these mechanisms are not yet sufficient for promoting the use of a formal specification language. A major drawback relies on the number of formal statements that should be written so that the requirements be completely and consistently expressed. In particular, when we consider two applications belonging to the same application domain, it is definitively tedious to have to encode similar specifications twice. Those are conclusions which are shared for example by [Die89] and [Reu91] and which have led to the introduction of libraries of reusable cliches for some application domain. Analogously to the paradigm considered in the Esprit Project Ithaca [Pro89] [Per90], we feel essential to be able to distinguish between two roles for the analyst, namely the "application engineer" and the "application developer". The application engineer is responsible for providing generic concepts for a given application domain while the application developer is in charge of reusing and tailoring these generic concepts to the needs of the requirements expressed for a particular application.

In our language, the introduction of structuring mechanisms (in particular, the *parameterization* mechanism associated with the *syntactical inheritance* mechanism) follows the same objective, i.e. to make distinct the modelling of requirements typical

of some application domain from the modelling of requirements specific to a particular application. In Fig. 6, we have illustrated the use of this mechanism by making distinct the part of the requirements related to a general allocation of resources problem from the specific requirements associated with the borrowing of books in the library. These specific requirements are obtained by instantiating the parameterized RESOURCE-ALLOCATION specification (Fig. 6a) with substitution of BOOK and USER for RESOURCE and CONSUMER respectively. In the resulting instantiation (Fig. 6b), it should be noted the renaming of the operations inherited from the parameterized cluster.

Doing so, it should be noted how the specification of the LIBRARY-S has been considerably simplified with respect to its previous specification presented in Fig. 5. Moreover, we have experimenting the usefulness of the RESOURCE-ALLOCATION parameterized type by considering it in a completely different case study related to a telephone switching network system.

Type Cluster RESOURCE-ALLOCATION[RESOURCE,CONSUMER]

Interface events

$Grants : RESOURCE \times CONSUMER$

State inspection operations

Varying $Resources : RESOURCE$
$WaitingConsumers : CONSUMER$
$PendingRequests : RESOURCE \times CONSUMER$

Constraints

* The three following constraints are derived from the ERA diagram

* A grant occurs to an available resource and for a waiting consumer
$Grants(b, u) \Rightarrow Resources(r) \wedge WaitingConsumers(c)$
* A pending request links a resource of the system (i.e. which is or has been available) and a waiting consumer
$PendingRequests(r, c) \Rightarrow \Diamond (Resources(r)) \wedge WaitingConsumers(c)$
* A consumer is waiting if and only if he/she has at least one pending request
$WaitingConsumers(c) \Leftrightarrow \exists r : PendingRequests(r, c)$

* A request is pending until the resource is granted
$PendingRequests(r, c) \Rightarrow \bigcirc (PendingRequests(r, c) \; \mathcal{U} \; Grants(r, c))$

* A pending request and an available resource are removed if and only if corresponding resource has been granted
$Grants(r, c) \Rightarrow \bigcirc (\neg PendingRequests(r, c) \wedge \neg Resources(r)$
$\ominus (PendingRequests(r, c)) \wedge \neg Grants(r, c) \Rightarrow PendingRequests(r, c)$
$\ominus (Resources(r)) \wedge (\nexists c : Grants(r, c)) \Rightarrow Resources(r)$

* A resource can only be granted to a waiting consumer
$Grants(r, c) \Rightarrow \ominus\ (PendingRequests(r, c))$

* Available resources for which requests are pending are granted without unnecessary delay
$(\exists r : Resources(r) \wedge (\exists c :\ PendingRequests(r, c)) \Rightarrow \circ\ (\exists c' :\ Grants(r, c'))$

Data Type RESOURCE

Data Type CONSUMER

Fig. 6a

Type Cluster LIBRARY-S is RESOURCE-ALLOCATION[BOOK,USER]

Interface events

CheckOuts **for** *Grants*

State inspection operations

Varying *OnShelfBooks* **for** *Resources*
RequestingUsers **for** *WaitingConsumers*
PendingReqs **for** *PendingRequests*

Constraints

* Link with the requests coming from the environment
$Requests(b, u) \Rightarrow \circ\ (PendingReqs(b, u))$

Fig. 6b

6 Conclusion

For more than fifteen years, a number of requirements specification languages have been proven useful in industrial environments. These include, e.g., PSL/PSA [Tei77], SADT [Ros77], SREM [Alf77] or REMORA [Rol82]. We feel however that such languages present limitations of several natures. First, they lack a sound theoretical basis and the semantics of the various language constructs is in some cases ill-defined. Thereby, the accompanying tools provide limited support (essentially editing, storage and manipulation facilities) and analysis capabilities are restricted to syntactic checks. Second, they also have a limited expressiveness because only some aspects of the requirements can be formulated, like data-structures, data flows, and limited additional properties. In particular, it should be noted that, in most case, the dynamic aspects of the system evolution can only be captured with algorithmic descriptions, i.e. with the risk of introducing over-specifications.

By contrast, new emerging requirements languages (e.g. LARCH [Gut85], RML [Gre86], Z [Suf86] GIST [Fea87], MAL [Fin87], OBLOG [Ser89], ERAE [Dub88, Dub91a] or TELOS [Myl90]) are based on logical/mathematical semantics (e.g. initial algebras, first order logic) and exhibit two essential features, i.e. *expressiveness* and *structuring mechanisms.*

- At the expressiveness level, we have drawn the following conclusions:
  - the use of a first-order *logic* framework seems to be a reasonable basis because of the variety and the naturalness of constraints that can be expressed;
  - things of interest have not only to be expressed in terms of *data* but also in terms of *clusters* when we want to model real-world persistent entities;
  - dynamic aspects of the system can be modelled using a *state-based* view (with transitions explaining state changes) or an *event-based* view (with events supporting the description of interactions). We feel that the latter supports the description of interactive systems whereas the former is more suited for the description of sequential systems;
  - the description of a system can be *snapshot* or *history* oriented. The snapshot view only supports descriptions expressed in terms of two successive states of an history. By contrast, an historical perspective allows to refer to the whole history of states. The historical view supports a more declarative view than the snapshot view;
  - only a few approaches permit the reference to (i) real-time aspects without the introduction and the management of somewhat artificial clocks and to (ii) organizational aspects related to the responsibility and the cooperation of different agents within the system.
- At the structuring mechanisms level, it appears that many approaches only offer a *syntactical* scoping mechanism to deal with name's clashes in large specification. By contrast, within the *algebraic* framework, a number of more *semantic* relationships are envisaged (e.g. parameterization, inheritance, etc). However, in most cases, these mechanisms have been investigated within the framework of an 'equational' logic (i.e. a subset of first order logic), not sufficiently expressive for our requirements modelling purposes.

In the *Icarus* project, the conclusion of these experiences have led to the development of the GLIDER language (a General Language for an Incremental Definition and Elaboration of Requirements) [Dub91b] supporting the expression of the different kinds of requirements presented above and offering powerful semantic and syntactic structuring mechanisms (e.g. parameterization and inheritance). The language we have introduced in this paper is a dialect of this GLIDER language also inspired by some recent experiences of two of the authors with the ERAE language.

Our research plans are in four directions:

- the enhancement of the language with the expression of *organizational* requirements (like "this department is responsible for producing data to be processed by the computer system"). Preliminary experiences [Dub91c] consider the introduction of the notion of agent and action in order to model and to reason on a responsibility relationship as well as to be able to express requirements on performances, reliability and security aspects;
- the validation of the methodology proposed for composite systems through the study of the conclusions resulting from large experiments currently done in different industrial environments;
- the development of an integrated environment of tools made of textual and graphical syntactic editors, an object-oriented repository for managing intermediate specifications fragments and semantic analyzers for verifying consistency and completeness of requirements fragments and also for deriving some new relevant properties about them;
- the development of a *requirements assistant* supporting the *process* followed by the analysts during the elaboration of the requirements document, as well as the study of the *rationale* that have led to the choice of a particular process.

Acknowledgment: This work was partially supported by the European Community under Project 2537 (ICARUS) of the European Strategic Program for Research and Development Technology (ESPRIT). We are indebted to J.P. Finance, A. van Lamsweerde, F. Orejas, J. Souquières and P. Wodon who participated in the design of the GLIDER language. We are also grateful to J. Hagelstein for his basic contribution in the design of the ERAE language.

References

[Alf77] M.W. Alford, "A Requirements Engineering Methodology for Real-time Processing Requirements," *IEEE Trans. Soft. Eng.*, SE-3(1), pp. 60-69, 1977.

[Die89] N.W.P. van Diepen and H.A. Partsch, "Some Aspects of Formalizing Informal Requirements," Department of Computer Science, University of Nijmegen, The Netherlands, 1989.

[Doe91] E. Doerry, S. Fickas, R. Helm and M. Feather, "A Model for Composite System Design," in *6th Int. Workshop on Software Specification and Design*, Milano, October 1991.

[Dub88] E. Dubois, J. Hagelstein and A. Rifaut, "Formal Requirements Engineering with ERAE," *Philips Journal of Research*, 43, nos. 3/4, 1988.

[Dub88b] E. Dubois, "Logical Support for Reasoning about the Specification and the Elaboration of Requirements," in *The Role of Artificial Intelligence in Databases and Information Systems*, WG2.6/WG8.1 Conference, Guangzhou, China, pp. 29-48, July 1988.

[Dub90] E. Dubois, "Supporting an Incremental Elaboration of Requirements for Multi-agent Systems," in Draft Proceedings of *International Working Conference on Cooperating Knowledge Based Systems*, University of Keele (England), October 3-5, pp. 130-134, 1990.

[Dub91a] E. Dubois, J. Hagelstein and A. Rifaut, "From Natural Language Processing to Logic for Expert Systems. Chapter 6: a Formal Language for the Requirements Engineering of Composite Systems," A. Thayse (Editor), Wiley, 1991, 535 pages.

[Dub91b] E. Dubois, Ph. Du Bois, A. Rifaut, P. Wodon, "GLIDER User Manual," SpecFunc Deliverable, ESPRIT Project Icarus 2537, June 1991.

[Dub91c] E. Dubois, "Use of Deontic Logic in the Requirements Engineering of Composite Systems," *First International Workshop on Deontic Logic in Computer Science*, Amsterdam, The Netherlands, 11-13 december, 1991.

[Ehr90] H. Ehrig and B. Mahr, "Fundamentals of Algebraic Specifications : Module Specifications and Constraints," *EATCS Monographs on Theoretical Computer Science*, W. Brauer, G. Rozenberg, A. Salomaa (Eds), Springer-Verlag, 1990.

[Fia91] J. Fiadeiro and T. Maibaum, "Describing, Structuring and Implementing Objects," in *Proc. Foundations of Object-Oriented Languages*, Noordwijkerhoud (The Netherlands), LNCS 489, Springer Verlag, pp. 275-310, 1991.

[Fin89] A. Finkelstein and H. Fucks, "Multiparty Specification," in *Proc. Fifth International Workshop on Software Specification and Design*, pp. 185-195, 1989.

[Fea87] M.S. Feather. "Language Support for the Specification and Development of Composite Systems," in *ACM TOPLAS*, vol. 9, 2, pp. 198-234, April 87.

[Fea89] M.S. Feather. "Constructing Specifications by Combining Parallel Elaborations," in *IEEE Trans. Soft. Eng.*, vol. 15 (2), February 1989.

[Fin87] A. Finkelstein, C. Potts. "Building Formal Specifications Using 'Structured Common Sense'," in *Proc. Fourth International Workshop on Software Specification and Design*, pp. 108-113, 1987.

[Gre86] S.J. Greenspan, A. Borgida and J. Mylopoulos, "A Requirements Modeling Language and its Logic," *Information Systems*, vol 11(1), pp. 9-23, 1986.

[Gut85] J. Guttag, J. Horning and J. Wing, "Larch in Five Esasy Pieces," Research Report 5, Digital Systems Research Center, 1985,

[Joh88] W. L. Johnson, "Deriving Specifications from Requirements," in *Proc. 10th Int. Conf. on Software Engineering*, Singapore, pp. 428-438, 1988.

[Jun91] R. Junglaus, G. Saake and C. Sernadas, "Formal Specification of Object Systems," in *Proc. TAPSOFT'91*, Brighton (UK), LNCS 494, Springer-Verlag, pp. 60-82, 1991.

[Myl90] J. Mylopoulos, A. Borgida, M. Jarke and M. Koubarakis, "Telos : A Language for Representing Knowledge about Information Systems," *ACM Trans. Information Systems*, 1990.

[Ore89] F. Orejas, V. Sacristan and S. Clerici, "Development of Algebraic Specifications with Constraints," in *Categorical Methods in Computer Science*, Springer LNCS, 1989.

[Per90] B. Pernici, "Class Design and Metadesign," in *Object Management*, D. Tsichritzis (ed), Geneva University, p. 117-132, 1990.

[Pro89] A. Profrock, D. Tsichritzis, G. Muller and M. Ader, "ITHACA: an integrated

toolkit for highly advanced computer application," in *Object Oriented Development*, D. Tsichritzis (ed), Geneva University, pp. 321-344, 1989.

[Reu89] H.B. Reubenstein and R. C. Waters, "The Requirements Apprentice : An Initial Scenario," in *Proc. Fifth International Workshop on Software Specification and Design*, pp. 211-218, 1989.

[Reu91] H.B. Reubenstein and R. C. Waters, "The Requirements Apprentice : Automated assistance for requirements acquisition," in *IEEE Trans. Soft. Eng.*, 17(3), March 1991.

[Rol82] C. Rolland and C. Richard, "The Remora Methodology for Information Systems Design and Management," in *Information Systems Design Methodologies: A Comparative Review*, T.W. Olle, H.G. Sol, A.A. Verrijn-Stuart (eds), North-Holland, pp. 369-426, 1982.

[Ros77] D.T. Ross and K.G. Schoman, "Structured Analysis for Requirements Definition," *IEEE Trans. Soft. Eng.*, SE 3(1), pp. 1-65, 1977.

[Ser80] A. Sernadas, "Temporal Aspects of Logic Procedure Definition," *Information Systems*, vol. 5, pp. 167-187, 1980.

[Ser89] A. Sernadas, C. Sernadas and H.-D. Ehrich, "Abstract Object Types: a Temporal Perspective," *Colloquium on Temporal Logic and Specification*, B. Banieqbal, H. Barringer and A. Pnueli (eds), LNCS 398, Springer-Verlag, pp. 324-350, 1989.

[Suf86] B. Sufrin (ed), "Z Handbook," Oxford Programming Research Group, 1986.

[Tei77] D. Teichroew and E.A. Hershey, "A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Soft. Eng.*, SE-3(1), pp. 41-48, 1977.

This article was processed using the LaTeX macro package with LMAMULT-LNCS style