Abstract
Requirements engineering is the activity investigating the user's needs in the context of a software development project. This activity is critical because it takes place early in the project's history, and its errors are often discovered late, most of the time only when the software system is delivered. Despite this fact, it currently relies more on interviewing techniques and ambiguous drawings than on a sound technical basis.
In the context of the ESPRIT project METEOR, we have developed an approach to requirements engineering based on an expressive and formal language, called ERAE, complemented with methodological guidance. The most influential idea in this project has been that better requirements will be obtained by not focusing on the needed software system, but rather on the larger system formed by the computer and its environment. Two other specificities of our approach are the expressiveness and the formality of the ERAE language. The language is conceptually close to the natural way users express themselves, and therefore requires little paraphrasing when recording the requirements. Its formality not only ensures its unambiguity but also provides for deductive power, which helps detecting a number of misunderstandings induced by the unreliable communication with customers (ambiguities, contradictions, etc). This paper presents the language, and illustrates the key role of formality in the methodology.

Keywords: Formal languages, methodology, requirements engineering, specifications.

1. Introduction
The production of software is nowadays the main bottleneck for the use of computers\(^1\). The discrepancy between the need for good software and

\(^1\) This work was partially supported by the Commission of the European Communities, under Project 432 (Meteor) of the ESPRIT programme.
the inability to meet that demand has even given rise to the term 'software crisis' in the 70s. The major cause of that crisis is the poor understanding and control of the production process, i.e. the sequence of actions taken to create a software system).

1.1. Process models

A process model is an abstract and generic description of a process, providing some guidance for organizing and controlling it. The first process models for software development, also called life-cycle models, decomposed the activity into a sequence of phases: requirements engineering, main design, detailed design, programming, verification, and maintenance. In case of problems during a phase, it may be necessary to back up one or more phases. Unfortunately, such a sequential organisation naturally limits the amount of this back up, and in particular leads to perform the maintenance at the program level, whereas the design or even the requirements should be questioned. Another drawback is that the strong separation between phases suggests using different, potentially not integrated, techniques in each.

Later models have introduced the rapid prototyping paradigm, which consists in developing quickly an inefficient version of the system to obtain feedback from the users, before developing an efficient version with the same behaviour. However, it may be costly to develop a prototype, which is not the ultimate solution anyway: users are assumed to accept the prototype when they actually only accept its behaviour in the cases they have tried.

Since a few years, the favour of the academic community goes to process models adhering to the transformational paradigm, where correctness-preserving transformations are applied to convert formal specifications into programs, and where maintenance is performed on the specifications. This covers well the activity leading from formal specifications to programs, but does not address the activity of obtaining the initial specifications, i.e. requirements engineering, or more shortly RE.

The METEOR ESPRIT project, started in October 1984, includes the extension of the transformational paradigm to RE. The Philips Research Laboratory in Brussels is in charge of this part of the project, as well as of the overall coordination. The resulting RE methodology and language is described in this paper.

1.2. The purpose of requirements engineering

It is wrong to assume that RE is a non-inventive activity, which discovers and records wishes of the future users of the system (hereafter called cus-
Indeed, the primary interest of customers is not in a computer system, but rather in some overall positive effects resulting from the introduction of a computer system in their environment. These positive effects, which we call ‘objectives’, must be achieved by a proper interaction of the computer with the rest of the environment.

Actually, a first part of RE is non-inventive: identifying objectives, but also expliciting needed application domain knowledge. A second part of RE is clearly inventive: in order to feed designers with an appropriate input, RE must go beyond objectives and define the borderline between the computer system and the environment, as well as a specification of the two. In this part, the requirements engineers propose solutions and negotiate them with customers and designers.

This view of RE is not reflected in current practice, where RE still focuses very much on the required computer system. As a result, users frequently complain that they were not made aware of the organizational changes required to run the new computer system.

1.3. The specific problems of requirements engineering

The two main specificities of RE, compared to other activities of software development are that
(1) it starts from inherently informal knowledge, and
(2) it involves people not literate in software.
These two aspects contribute to make the validation of information a crucial problem. How can requirements engineers avoid misunderstanding customer’s statements, and ensure that customers will understand the proposed solutions?

Unfortunately, these misunderstandings that are likely to occur are also particularly costly if not detected during RE. Indeed, the next occasion for checking the customers’ satisfaction is much later, when the resulting software is delivered. Any change of requirements at this point will impact on numerous design and implementation decisions.

Two other problems are worth mentioning, which are not so specific to RE but impact on the language and methodology. First, there will not be one customer, but many of them, with different – often contradictory – views and interests. Second, requirements are typically large, and must be cut into manageable parts.
1.4. A suitable language for requirements engineering

In current practice, the validation problem is addressed by basing RE on documents written in English and in informal graphical languages, in the hope that customers are better able to check these. The resulting ambiguity and lack of sophisticated checks has already led to disasters. An alternate solution is the prototyping paradigm, but its limitations, mentioned previously, prevent it from being the ideal solution.

We propose to address the validation problem by using a formal and expressive language. A language is formal if its syntax is recognizable by a machine, and if its semantics is given by a rule associating a suitably defined mathematical object to every expression. The advantage of formality is to guarantee the absence of ambiguity, but also to permit deductions. The importance of the latter will be illustrated in sec. 3, but we may already mention the possibility to rigorously prove that objectives are met by a proposed behaviour of the computer system and of the environment. Of course, a formal language is probably not readable by customers. The requirements engineers are assumed to convert parts of the requirements towards more suited forms when communicating with customers. Again, this is further discussed in sec. 3.

Expressiveness is the ability to easily describe typical RE knowledge, such as application domain knowledge and environment's behaviour knowledge. The language features which favour expressiveness are detailed in refs 6 and 7. Among others, it is essential to take into account the temporal dimension of the system under description, to describe both possible states of affairs and possible changes, and to allow the flexible expression of such changes. Abstraction and structuring mechanisms are also needed for mastering complexity, but this is less specific to RE and will not be developed here.

In trying to combine formality and expressiveness, we have been inspired by ideas which matured in various computer disciplines faced with the problem of expressing some form of knowledge and reasoning about it: RE itself, database conceptual modelling, and knowledge representation in artificial intelligence. We will review them in turn.

While most current practice RE languages are based on informal graphical notations [8,9], there is a clear trend towards formality in more advanced methods [10-12] (see ref. 13 for a survey of the field). The outlook of these methods depends on their choices regarding expressiveness, and their stress on deductiveness. Actually, deductions are not presented by the current approach as the essential means for tackling validation, and expressiveness is often limited as far as dynamic aspects are concerned. Typically, the de-
scription of dynamics is very operational, sometimes even directly inspired by programming concepts.

Database conceptual modelling is an activity similar to the engineering of requirements for a database system. It consists in analysing the part of the world on which information will be kept – the ‘universe of discourse’ – in order to define a representation-independent view of the database structure. This discipline developed powerful tools for describing structural aspects of an application domain, such as the entity-relationship model\(^4\), and the abstraction mechanisms of classification, specialisation and aggregation\(^5\). Later languages made extensions, often based on logic, to capture more semantics of the universe of discourse\(^6-18\). Still, these languages suffer from basic assumptions, which need not be valid when developing systems other than databases: identification of the structure of the system with the one of the universe of discourse, evolution of the system often described by means of transactions, omission of non-functional requirements (performance, actuality of data, etc).

Knowledge representation is an established subfield of artificial intelligence, which has inspired several recent RE languages\(^19,20\). Yet, as for database conceptual modelling, the difference in purpose does not make knowledge representation techniques directly applicable in requirements engineering. Indeed, these techniques represent knowledge for automatic use, for instance by an expert system. A compromise is therefore needed between expressiveness and efficient processing, which is not necessary in RE. Nevertheless, relevant contributions may be found in semantic networks, frame systems, and the use of logic\(^7\).

The various influences mentioned above agree on a number of key points: representation of knowledge as objects linked by associations, use of logic for expressing constraints, importance of structural knowledge (classification, generalisation, aggregation). The main uncovered area is the flexible expression of the system's evolutions. It is taken into account in the ERAE model, presented in the next section. Section 3 is devoted to the associated methodology, including one example of use of deductions in the language. Section 4 will point out the status of industrialisation of ERAE, and the future directions of development.

2. The ERAE language

A formal language must establish a correspondence between what it describes and some properly chosen mathematical objects. The purpose of RE is to describe complex systems formed by computers and their environment.
E. Dubois, J. Hagelstein and A. Rifaut.

Therefore, the mathematical objects provided by ERAE are 'histories', i.e. sequences of states with sets of events labelling the transitions between states. Each state is a configuration of existing objects and associations between them. Each state is also labelled by a time value, which increases along the history.

As a consequence, developing an ERAE specification involves two activities:
- identifying the application-dependent history components, i.e. the various kinds of objects and associations in the states, and the various kinds of events;
- expressing constraints which exclude unwanted histories.
These two activities produce the two parts of an ERAE specification: the 'declarations' and the 'statements'.

2.1. Declarations

This subsection will first delimit the information expressed in declarations, then illustrate the syntax on an example, and finally discuss the concept of groups.

The purpose of declarations is to fill the general structure of histories by identifying application-dependent components: objects and associations forming the states, as well as events labelling the transitions. Actually, ERAE provides the following concepts:
- Objects model phenomena perceived individually, without any needed reference to others. They are further subdivided as follows:
  - events, used to label transitions of histories, should be perceived as instantaneous phenomena;
  - entities, which exist in a number of consecutive states in the histories, should correspond to phenomena with a certain duration;
  - values are abstract concepts used to characterize entities or events, but are not of interest by themselves; actually, ERAE is not meant to define values, but rather to use predefined value types, such as integers or strings.
- Associations are not perceived as individuals, but rather as state-dependent links between objects. They are subdivided as follows:
  - relations are associations between entities and/or events;
  - attributes are associations between an entity or event, and a value.

The name ERAE is derived from the names of the various concepts: Entity, Relation, Attribute, Event. Of course, the classification of a phenomenon is not objectively decidable. Such a classification scheme is only a tool for clarifying one's own perception, and is necessarily subjective and depending on the intended purpose.
The declarations may identify both individual objects and (more often) sets of related objects, which are called ‘groups’ in ERAE. These may have a state-dependent membership and may intersect under certain circumstances detailed later. A group is also an object itself, in fact an entity, which may participate in associations. In particular, a group may be contained in another one because membership is reflected by a predefined relation called ‘is-in’.

Associations are declared between specific groups, value types or individuals. An association may never hold between objects which, in that state, are not in the required groups or are not the required individuals.

Declarations only mention entities that may exist, events that may occur, and associations that may hold. It is the role of statements to go further and state that some associations are mandatory (e.g. every customer must have an identification number).

Example

The syntax of an ERAE declaration may be either textual or graphical. We will illustrate the latter on part of a ‘transit node’, in a telephone network. Here is a (slightly edited) excerpt of an existing informal description:

‘The transit node contains a permanent input control port, up to N input data ports which are added dynamically, and as many output data ports. All ports are concurrent to all others.

The input control port accepts and treats messages of the form ‘Add-port-msg(n)’, which gives the node knowledge of a new input data port and a new output data port, both numbered ‘n’. The node then starts to accept and treat messages sent to the input data port, as indicated below.

An input data port accepts and treats only messages of the form ‘Data-msg(d)’. Such messages arrive from the environment only when the input port is available. The transit node routes the data ‘d’, unchanged, to any one (non-determinate) of the output data ports’.

This case is treated more extensively in ref. 21. The declaration in fig. 1 corresponds to the excerpt above.

An individual entity is represented by a squared box with a double outline (e.g. Control-port-in). An individual event is represented by an oval with a double outline (not illustrated). A group of entities is represented by a simple squared box (e.g. Data-ports-in). A group of events is represented by a simple oval (e.g. Data-msgs). A relation applicable between the elements of two groups is represented by an arrow linking them (e.g. port); an attribute is represented by a simple line (e.g. in-port-nr). A value type is represented
by some text at the end of a line, which may be a predefined name such as 'Data', or a subtype such as '[1..N]'.

Some relations or attributes are applicable to the elements of a group, whereas others are applicable instead to the group, considered an object (e.g. is-in on its ending side). We indicate the latter case by putting a circle around the corresponding end of the line (except for individuals, where there is no ambiguity).

Two frequent kinds of statements may be expressed graphically, among the declarations: state-dependence and cardinality.

The state-dependence of associations is indicated by using dashed lines, whereas invariant associations use plain lines. A dashed individual entity denotes one which needs not always exist. A dashed group has a state-dependent membership. Individual events are never dashed, because events have no existence. In the example, all relations and attributes are constant, but the membership in the groups 'Data-parts-in' and 'Data-parts-out' is state-dependent. Note that associations may only hold as long as the involved objects are in the right groups. Even invariant associations stop holding as soon as this condition is not met.

The numbers at the end of the associations are cardinality constraints. A
symbol ‘i’ at the end of a line means that, in every state, each object at that end participates exactly in \( i \) occurrences of the association. When a range of values needs be specified, the notation ‘\( i:j \)’ is used. It means that the objects at that end participate to at least \( i \) and at most \( j \) occurrences. The absence of information is equivalent to ‘0:∞’.

Groups

The concept of group is more subtle than one may expect, as explained in ref. 22. In particular, the possibility of state-dependent membership in groups lead to associating two sets of objects to any group: its population, i.e. the objects belonging to it in a given state, and its type, i.e. the union of its population over all states. The two concepts coincide for state-independent groups.

Two ERAE groups normally have disjoint instantaneous populations, except if they are allowed to intersect because of the occurrence of is-in arrows in the diagram. For instance, in the declaration above, ‘Data-msgs’ may share elements with ‘Data-arrivals’. (From the cardinality information, we may even deduce that the latter is included in the former.) In fact, ‘Data-arrivals’ and ‘Data-sendings’ may also share elements, provided these common elements are in ‘Data-msgs’. The absence of other is-in arrows implies that these are the only possible instantaneous intersections: for instance, there is no intersection between ‘Data-parts-in’ and ‘Data-parts-out’.

The instantaneous disjointness of two groups does not prevent entities from moving from one group to the other. This allows, for example, to define the disjoint groups ‘children’ and ‘adults’, while permitting objects to pass from the former to the latter.

The inheritance of properties between elements of various groups is known to be an interesting abstraction mechanism. It allows to define a group as a specialization of another, in the sense that the elements of the former may participate in all the associations granted to the latter, and in some additional ones. This is provided in ERAE by the possible intersection between groups. For instance, the attribute ‘data’ is inherited by every element of ‘Data-arrivals’ because this group is instantaneously included in ‘Data-msgs’. However, only some members of ‘Data-msgs’ may be at the origin of the ‘corresponds’ relation, namely those also in ‘Data-arrivals’.

2.2. Formal interpretation of declarations

Declarations define the mathematical symbols usable in statements. These are chosen to reflect the intuitions presented in sec. 2.1.

First, a specific logic is used to reflect the distinction between categories
of objects: entities, events, and groups for a finer classification. ERAE maps the concept of 'type' informally introduced in the previous section into the same concept available in multi-sorted logics. In such a logic, every constant and variable is given a type, and can only denote an object in the part of the domain corresponding to that type. Similarly, functions and predicates are declared to take their arguments in some types, and functions produce values of a certain type. A formula where a mismatch occurs is not 'well-formed', i.e. it is not considered part of the language.

The types associated to a given declaration are derived as follows:

- The types 'Entity', 'Event' and the various value types are predefined. Entity contains all entities that will exist during an history, and Event contains all events that will occur during it.
- Every group of entities or events defines a type, containing all objects that may, in some state, be in this group. This type bears the same name as the group.
- Every individual entity or event defines a type, containing it as only element. The type of an individual 'Foo' is named '{Foo}'. As mentioned in sec. 2.1, every group (of events or entities) also counts as an individual entity.
- Any union of types is a type. This is to handle overloaded symbols.

Syntactically, each term has exactly one type, but, semantically, the parts of the domain corresponding to two types are not necessarily disjoint. Two types which may have a non-empty intersection are said to be 'compatible'. Any union of types is a type. This is to handle overloaded symbols. When deciding on the well-formedness of formulas, a term needs only have a type which is compatible with the position it occupies. In this way, any term which has a chance of denoting an appropriate object is accepted as well-formed.

The compatibility of the predefined value types is determined by their definition. These are not provided in ERAE, but in a prologue of the ERAE specification, where any appropriate mathematical technique may be used. In any case, these types are incompatible with the types containing entities or events.

The compatibility of entity and event types is deduced from the declarations as follows:

- each type containing entities is compatible with the type 'Entity';
- each type containing events is compatible with the type 'Event';
- two types are compatible when they may have an instantaneous intersection according to the occurrence of is-in arrows;
- the types of any pair of time-varying groups of entities (respectively events) are compatible.
– any union is compatible with its components, and the types compatible with them.

Because compatibility is interpreted as the possibility of intersection, it is a symmetric relation, but not a transitive one. Notice that a type containing entities is always incompatible with one containing events.

Once the types are determined, we must define some constants, functions, and predicates on them. These are predefined for the value types and include such symbols as ‘0’, ‘+’, ‘<’, etc.

The user-defined symbols and their associated type are derived from the declaration as follows:

– individual entities and events define constants, of the type containing them as only element (e.g. Foo is of type {Foo});
– an association (relation or attribute) defines a predicate taking its arguments in the types associated to its end points.

Here are some of the constants and predicates, with their types, defined by the example in sec. 2.1:

Constants:
- Control-port-in: \( \rightarrow \{\text{Control-port-in}\} \)
- Add-port-msgs: \( \rightarrow \{\text{Add-port-msgs}\} \)
- Data-ports-in: \( \rightarrow \{\text{Data-ports-in}\} \)

Predicates:
- corresponds: Data-arrivals \( \times \) Data-sendings
- data: Data-msgs \( \times \) Data

‘Port’ is an example of an overloaded relation name, a situation which is perfectly admitted in ERAE. The type of the overloaded name is the union of its different types.

In this example, the compatibility between types is as follows:
– Data-ports-in, Data-ports-out, \{Control-port-in\}, \{Data-ports-in\}, \{Data-ports-out\}, \{Add-port-msgs\}, \{Data-msgs\}, \{Data-arrivals\}, \{Data-sendings\} are all compatible with Entity;
– Add-port-msgs, Data-msgs, Data-arrivals, Data-sendings are all compatible with Event;
– Data-msgs, Data-arrivals, Data-sendings are mutually compatible;
– Data-ports-in and Data-ports-out are mutually compatible.

Sometimes, functions are more pleasant to use than predicates that are known to be functional. Such functions may be defined textually. For instance, in addition to the predicate ‘in-port-nr’, we define a function ‘in-port-nr’ as follows:

\[ \text{‘in-port-nr’: Data-ports-in} \rightarrow [1..N] \]
and we state that it verifies the following property:

$$\forall x,y: \text{in-port-nr}(x) = y \iff \text{in-port-nr}(x,y)$$

Such functions are used without notice in the rest of the paper.

Additional predefined predicates and functions are provided to reflect the intuitive properties of the various types:

- **exists:** Entity
- **occurs:** Event
- **time:** Event $\rightarrow$ Time
- **time:** $\rightarrow$ Time
- **is-in:** $\alpha \times \{\alpha\}$ (for every group $\alpha$)

'Exists' is true for each entity in some set of consecutive states. It reflects the intuitive fact that entities exist for some time. 'Occurs(ev)' is true in a state if 'ev' denotes one of the events associated to the transition leading to that state. It is true for each event in a unique state, but there may be states where it is not true for any event. The function 'time' applied to an event gives the time value associated to the state where 'occurs' is true for that event. (The type 'Time' is among the predefined value types.) Whereas 'time(ev)' is state-independent, the function 'time' without arguments is state-dependent. Its value in a state is the time value labelling the state. 'is-in' corresponds to the instantaneous membership of objects in groups. Of course all these predefined functions and predicates are implicitly constrained to behave as expected (see 'implicit statements' in the next section).

### 2.3. Statements

The statements are written in a typed real-time temporal first-order logic with equality. The constants, functions and predicates of the logic are the ones defined in the declarations, plus the predefined ones and '='. Variables may be typed by expressions of the form

```
<variable> : <type>;
```

**Static properties**

A first set of operators are the usual ones for first order logic:

- $\forall$ (for all), $\exists$ (there exists), $\land$ (and), $\lor$ (or), $\Rightarrow$ (implies), $\neg$ (not)
As predicates and functions may be state-dependent, the truth of a formula may vary from state to state. A formula stated to be true must be so in every state, and is thus an invariant. The following one is an example:

'When an input data port with some number is available, then an output data port with the same number is also available.'

\[
\text{dpi: Data-ports-in; dpo: Data-ports-out; i: [1..N]; } \\
\forall \text{dpi, i: in-port-nr(dpi, i) } \Rightarrow \exists \text{dpo: out-port-nr(dpo, i)}
\]

**Dynamic properties**

The usual first-order logic operators allow to state only properties which refer to exactly one state. This covers some constraints of a dynamic nature, namely the interaction between events and components in the state just following their occurrence:

'An add-port-msg ensures the existence of an input data port with the appropriate number.'

\[
\text{apm: Add-port-msgs; dpi: Data-ports-in; } \\
\forall \text{apm: occurs(apm) } \Rightarrow \exists \text{dpi: in-port-nr(dpi) = port-nr(apm)}
\]

We have used here the functional form of 'in-port-nr' and 'port-nr'.

Such statements are, however, not sufficient to complete the requirements. For instance, they do not permit to state preconditions of events, or constraints involving two successive states, and of course not those involving events and states which are further apart. Temporal operators 23) are a classical means to reference several states in the same formula. The truth of the whole formula still depends on the state in which it is considered, but subformulas may be interpreted in another state by prefixing them by one of the following operators:

- \(\circ \phi\) means that \(\phi\) holds in the next state,
- \(\diamond \phi\) means that \(\phi\) holds either in the current state or in some future one,
- \(\square \phi\) means that \(\phi\) holds in the current state and in all future ones,
- \(\phi \sqcup \psi\) means that \(\phi\) holds in all states from the current one until just before the first next one where \(\psi\) holds.
There are corresponding operators referring to the past:

- \( \bullet \phi \) means that \( \phi \) holds in the previous state,
- \( \blacklozenge \phi \) means that \( \phi \) holds in the current state or has held in some previous state,
- \( \square \phi \) means that \( \phi \) holds in the current state and has held in all previous states,
- \( \phi \& \psi \) means that \( \phi \) holds in all states between the current one and just after the last preceding one where \( \psi \) held.

Here are some typical statements using these operators:

'A data arrival must be followed later by some corresponding data sending.'

'da: Data-arrivals; ds: Data-sendings;
\( \forall da: \text{occurs}(da) \Rightarrow \exists ds: \text{corresponds}(da,ds) \land \blacklozenge \text{occurs}(ds) \)

'When an add port message occurs, there may not be any port with the corresponding number.'

'apm: Add-port-msgs; dpi: Data-parts-in;
\( \forall apm: \text{occurs}(apm) \Rightarrow \neg \exists dpi: \bullet \text{in-port-nr}(dpi) = \text{port-nr}(apm) \)

Real-time properties

The expression of real-time properties requires a new set of operators, which are in fact simple extensions of the temporal ones \(^{24}\). Indeed, a duration may be associated to any temporal operators, except '\( \bullet \)' and '\( \circ \)'. This duration is interpreted as illustrated below:

- \( \Diamond_{3'} \phi \) means that \( \phi \) will be true in some future state, exactly 3 minutes later than the current one,
- \( \Diamond_{>5'} \phi \) means that \( \phi \) will be true in some future state, more than five seconds from the current one,
- \( \square_{\leq 2h} \phi \) means that \( \phi \) will be true in all future state, up to 2 hours from the current one,
- \( \phi \land_{< 1 \text{ day}} \psi \) means that \( \phi \) will remain true until \( \psi \) becomes true, or for one day if \( \psi \) does not become true before.

Here is an example of a real-time statement:

'Two successive add port messages are separated by at least 10 milliseconds.'
Formal requirements engineering with ERAE

\[ \text{apm}_1, \text{apm}_2: \text{Add-port-msgs}; \]
\[ \forall \text{apm}_1, \text{apm}_2: \text{apm}_1 \neq \text{apm}_2 \land \text{occurs}(\text{apm}_1) \Rightarrow \Box_{\leq 10ms} \neg \text{occurs}(\text{apm}_2) \]

Further examples of statements may be found in refs 25 and 21.

Implicit statements

When reasoning about ERAE statements and making deductions, the explicitly given statements may be combined with other implicit ones, automatically derived from the declarations. These express additional properties that we would like to assume about the declared mathematical objects.

For instance, associations not involving events can only hold when the involved entities exist and are in the required group. (Typing on the group only guarantees that the entities will sometime belong to it.)

An object of type 'Data-parts-in' may only have an attribute 'in-port-nr' when it actually is in the group 'Data-parts-in'.

\[ \text{dpi}: \text{Data-parts-in}; i: [1..N]; \]
\[ \forall \text{dpi}, i: \text{in-port-nr} (\text{dpi}, i) \Rightarrow \text{is-in} (\text{dpi}, \text{Data-parts-in}) \]

Such implicit statements, combined with the rules of interpretation of declarations, form the axiomatic semantics of ERAE, as given in ref. 26.

3. The ERAE methodology

The RE process should be systematized, made explicit and formalized for a number of good reasons: premature decisions are avoided by expliciting the choices and justifying them; an historical recording of the process allows to better control its progress and to backtrack to the right point in case of need; past RE processes may be reused more effectively than past RE specifications; etc. To this end, we need an abstract description of the process, i.e. a process model.

The execution of RE in a specific case may be viewed as a directed graph of actions performed by requirements engineers (henceforth called 'analysts') and transforming a set of requirements documents. In addition, an action may be decomposed into a whole subgraph of more detailed sub-actions.

A process model provides a supply of components for building such a graph:
E. Dubois, J. Hagelstein and A. Rifaut

- a definition of the RE database which contains the requirements under elaboration as well as information about the status of the current RE process (managerial information, past actions and their justification),
- action types, organised in a hierarchy, characterized by the actors performing them and their effect on the RE database,
- heuristics for controlling the application of actions.

Our long-term goal is to formalize the various components of the process model, and, in particular, to provide a meta-language for reasoning about the running process and for expressing heuristics\(^{27}\). However, the model presented in this paper is less ambitious. It only presents two of the levels in the hierarchy of actions types:
- a strategical level, identifying activities of negotiation between the analysts and customers or designers.
- a tactical level or 'interaction cycle' describing one interaction between an analyst and another actor.

3.1. Strategical level

At the higher level, RE is an activity of negotiation between analysts, customers and designers\(^{28,29}\). As in any negotiation, all parties must finally accept some duties, which consist here in 'implementing' (or guaranteeing the correctness) of part of the requirements:
- customers are responsible for the application domain knowledge and must implement the requirements on the environment's behaviour;
- designers must implement the requirements on the computer system.

Actually, the negotiation is necessary because at the starting point of RE one usually finds some objectives which will result from the cooperation of the future system with its environment. The essence of RE is therefore to define environment requirements and system requirements which, together, establish the objectives while being acceptable by, respectively, customers and designers. There is actually a third responsibility involved:
- analysts are responsible for the meeting of the objectives, provided the environment and system requirements are met.

Let us illustrate this on a fragment of the case study presented in sec. 2. A basic objective of the transit node is that a data message arriving on an input data port should be routed to any of the output data ports. Customers will obviously not accept to implement this requirement, but designers will not either, because the transit node can only process messages arriving on a port that it knows to be available. This knowledge must first be communicated by the environment.
In such a conflictual case, strategical heuristics are available, that suggest to define two sets of requirements, which can be assigned respectively to the two parties, and which, together, imply the original objective. In our case, the following couple of requirements is a candidate:

- data message may only be sent to an input data port if the latter has been previously installed by means of an add port message.
- data message arriving on an available input data port should be routed to any of the available output data ports.

The first requirement is assigned to the customers and the second one to the designers. The deductive system associated to the ERAE language allows to prove that the initial objective follows from these two requirements.

We are currently working on the formalization of methodological information such as the responsibility of an agent (customer or designer), the concept of interface between a computer and its environment, and the relationship connecting requirements to the objective they are meant to achieve. More details are given in ref. 30.

3.2. Tactical level

At a lower level of the process, RE consists in performing repeatedly activities belonging to four categories: acquisition, modelling, analysis and conversion. This sequence of activities is called the interaction cycle because one such cycle represents the work performed by the analysts between one acquisition of some informal knowledge and the next one (see fig. 2).

This concept was first introduced in ref. 28 and will be illustrated here on the case of the transit node.

Acquisition

Acquisition corresponds to the informal part of the interaction cycle, and involves analysts and either customers or designers. It may take various forms: interviews and discussion (open or directed), observation, experimentation, study of documentation, etc. It only interests us here to the extent that ERAE is not meant to be generally used during this phase. The result of acquisition should be represented in ERAE during the modelling activity, and the input for the next acquisition should be derived from the ERAE requirements by means of the conversion activity.

Modelling

Modelling is an action performed by analysts, which consists in converting the informal input material into its formal counterpart in the ERAE lan-
guage. It is admittedly a subjective activity which should be helped by heuristics and by an orthogonal choice of language concepts.

For a given perception of the world, i.e. for a given analyst at a given time, the decision to map a phenomenon to a certain mathematical object should be based on well defined properties of that object, not possessed by the alternative ones. This is why events, entities, values, relations and attributes result from different combinations of properties, including the autonomy of a phenomenon, its temporal dimension and the question whether it is of interest by itself or not. Some modelling heuristics take profit of that orthogonality by suggesting, for instance:

A phenomenon, that may be seen as instantaneous and is a topic of discourse independently of others, may be properly modelled as an event. The instantaneous nature must be understood in the sense that one is not interested in discussing its overlap with other phenomena.

There are many complementary heuristics, for instance the ones based on the grammatical structure of the informal starting material, such as:

Phenomena denoted by nouns in the discourse are more likely to correspond to objects, whereas verbs tend to correspond to associations. However, verbs might also correspond to events, especially when they have circumstantial complements, which then correspond to associations granted to these events.
Other heuristics are specific to the user of some constructs of ERAE, such as sub-grouping, or to specific kinds of statements. In particular, templates are provided for statements expressing a necessary precondition of an event, a sufficient precondition, the immediate effect of an event, etc.

Analysis

This activity, performed by analysts, heavily relies on the deductive power of the ERAE language. It is the main topic of ref. 25. The aim of the analysis is to detect the problems arising from the imperfect communication between analysts and customers, and to prepare the next interaction. These problems include, of course, misunderstandings, but also incompletences, contradictions, ambiguities, non-minimality, i.e. bias towards a particular implementation, and redundancy, i.e. repetition of information.

Specific heuristics address these various deficiencies. For instance, the following one addresses the issue of possible misunderstandings:

Instead of validating directly the statements produced during modelling, apply first some derivation rule to transform them. The following derivations may be tried:
- replace $P \Rightarrow Q$ by $\neg Q \Rightarrow \neg P$
- (others omitted)

Let us illustrate this on the first dynamic statement given in sec. 2. The sentence 'an add-port-msg ensures the existence of an input port with the appropriate number' corresponds closely to the formula

$$\forall \text{apm} : \text{occurs}(\text{apm}) \Rightarrow \exists \text{dpi} : \text{in-port-nr}(\text{dpi}) = \text{port-nr}(\text{apm})$$

which by applying the heuristics, is transformed into

$$\forall \text{apm} : (\neg \exists \text{dpi} : \text{in-port-nr}(\text{dpi}) = \text{port-nr}(\text{apm})) \Rightarrow \neg \text{occurs}(\text{apm})$$

Assuming that this statement is rephrased in English for validation, we obtain 'if there is no input data port bearing the number associated to a given add port message, then we may be sure that this message was not just received'.

If the derived statement is rejected by the customer, the original one must be considered suspect. If it is accepted, the confidence that there is no misunderstanding is increased more than if the customer had just agreed with a rephrasing of its initial statement. In the case above, it is likely that the customers will agree, but another positive effect will also be experienced. There are good chances that this new formulation leads the customer to
E. Dubois, J. Hagelstein and A. Rifaut

complete the information by saying ‘actually, I am sure that the add port message was never received up to then’.

Other analysis heuristics are those used to ensure the completeness of the requirements. Some of these address the a posteriori verification of completeness, but others suggest procedures for systematically obtaining the statements complementing a given declaration. An example of the latter kind is:

Consider every possible happening – occurrence of an event, change of value of a predicate – and try to identify prior necessary happenings. The corresponding statement will normally be of the form:

\(<\text{happening}> \Rightarrow \langle\text{past operator}\rangle \langle\text{necessary condition}\rangle\)

Conversion

Customers and designers are not required to understand the ERAE language. Communication with them uses a more suitable form, and therefore requires a conversion from ERAE towards other notations.

For customers, conversion may result in natural language, tables, drawings, existing informal languages (e.g. data-flow diagrams), execution traces or even a prototype. Each of these alternatives is less expressive, or more ambiguous (or both) than the formal RE language. Therefore, various conversions should be used, and cross-referenced. Up to now, experiments have been limited to the direct use of ERAE declarations, and the rephrasing of ERAE statements in English. Customers appeared quite able to properly understand these forms.

On the designers side, the conversion should take place towards the design language that will be used during design and implementation. In the context of the METEOR project, the COLD language, developed at the Philips Research Laboratories, Eindhoven, currently plays this role. It is, however, only a first iteration of a family of design languages with increasing expressive power. The conversion between ERAE and COLD is addressed in ref. 32.

4. Conclusions

The goal of the METEOR project is to provide formal software development methods usable in an industrial environment. A first way of ensuring this usability has been the performance of real-size case studies using ERAE. Such case studies have been successfully performed by an Irish software house which participates to the METEOR project. More recently, an
Formal requirements engineering with ERAE

experiment was performed within Philips, by the newly created Centre for Software Technology), in cooperation with Consumer Electronics. The task consists in expressing in ERAE a requirements specification previously performed using the method Structured Analysis / Real-Time, and not felt completely satisfactory. These experiments showed the usability of ERAE in an industrial environment, and provided highly valuable feedback, especially on the side of the management of large amounts of data.

Another essential aspect of the industrialization of a method such as ERAE is the availability of an automated support environment. Here again, the Centre for Software Technology plays its key role of technology transfer between research and product divisions, and will develop the needed environment and documentation. In the meantime, ERAE is taught as an example of an advanced RE method in the educational department of Philips Holland.

On the side of research, the METEOR project will last until the fall of 89, and the work associated to ERAE will continue beyond that point. Topics needing further investigation include the structuring mechanisms, the formalization of the process model and its heuristics, the expert-system guidance of this process, prototyping and conversion facilities, and the reuse of existing requirements.

REFERENCES

19) M.L. Brodie and E. Silva, in Information System Design Methodologies: A Compara-
E. Dubois, J. Hagelstein and A. Rifaut


Authors

Eric Dubois: M.S. (computer science) University of Namur, Belgium, 1981; Docteur-Ingénieur (computer science) Institut National Polytechnique de Lorraine, France, 1984; Philips Research Laboratory, Brussels, 1984-. Since 1982, he has worked on requirements engineering including development of formal requirements languages, studies of methods, investigation on supporting environments. He is presently a participant of the ESPRIT project METEOR.

Jacques Hagelstein: M.S. (electronics), University of Liège, Belgium, 1978; Ph.D. (computer science) University of Liège, Belgium, 1983; Collaborator Professor at the University of Namur, Belgium, 1983–1986; Philips Research Laboratory Brussels, 1982-. His thesis work was on the formal semantics and specification techniques of functional programming languages. Since joining Philips, he worked on formal specification languages, and is now leading the team working on formal requirements engineering, which developed the language ERAE.

André Rifaut: M.S. (applied mathematics), University of Louvain-La-Neuve, Belgium, 1981; Glaverbel, Brussels, Belgium, 1981–1984; Philips Research Laboratory Brussels, 1984-. Since joining Philips, he is working on formal requirements engineering, with a special interest in theoretical questions.