# A Method to Capture Formal Requirements: the *INVOICE* Case Study

Gianna Reggio

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, Italy
Viale Dodecaneso, 35 – Genova 16146, Italy
reggio @ disi.unige.it

## Introduction

In the last years myself in cooperation with other people have developed various specification formalisms for systems which can be named concurrent, parallel, reactive, ..., (e.g., [2, 7]) and applied to various case studies also in projects in cooperation with industry (see e.g., [10, 11, 4]). While the people forwarding the case studies have found the formal specifications produced with our cooperation satisfactory, they were asking "how can we produce such specifications?" and "how can be such specifications integrated into a development process?" That has lead us to try to derive from a specification formalism for systems, satisfactory from a formal point of view, e.g., expressive, powerful, allowing various degrees of abstraction, a method for supporting the development process. Some tasks have been already worked out, as

- informal specifications strictly corresponding to the formal ones, [3];
- a method (i.e., precise guidelines for the specifier) for giving design specifications with an associated graphic notation, [12];
- software tools helping to validate design specifications by prototyping, [1].

Now we are trying to develop a method for giving requirement specifications (thus requirements capture and specification) and a graphic presentation for such specifications. Our formal specification of the *INVOICE* case study is an application of an initial proposal for such method.

Requirement specifications should express the fundamental characteristic of something that we have to develop; and here we consider only the development of products that are changing along the time, and call them *systems*. The systems can be classified as *simple* or *structured*; the latter are those consisting of several parts (components), which are in turn systems, cooperating to give the whole system activity. The description of a system usually includes also entities which cannot be considered systems, since they are static; for example the quantities and the references in the case of *INVOICE*. Such static entities are just data, and we assume that are organized in data structures. In our approach systems and data, simple and structured systems are considered different and are analysed and specified differently.

Our requirement specifications follow an axiomatic (or better property oriented) style, and use a variant of the branching-time temporal logic, [7].

In our treatment of *INVOICE* the emphasis in on "HOW" to get the formal specification. The method provides precise instructions guiding the user to find, after some analysis of the system, in an exhaustive way all sensible properties. The resulting specification will have a very precise format, and that could be positive, since it helps

- to read the specification;
- to easily modify the specification (and that could be used to support the evolution of the system);
- to develop techniques for verification, if we know exactly the subset of formulae that will be used, then the verification task may be simpler.

On the other side such specifications may be longer and less elegant than those produced in a free way.

The guidelines are derived from our experience in specifying; the application to this example and to other simple case studies seems to be positive (see, e.g., in [9] some lift variants); but now the problem is to see whether the proposed method is coping with more realistic cases. At the moment the class of considered systems is still rather large, and we think that the guidelines may be effectively tuned, if we consider more particular classes of systems (e.g., purely reactive).

In this paper we present only the part of our method concerning requirement specifications of simple (non-structured) systems using a basic specification formalism, because that is what we need for *INVOICE*; see [9] for a complete presentation. Furthermore in our approach formal specifications have an associated informal presentation, which is not considered here.

In Sect. 1 we briefly present the formalism used in our approach; initially the reader can also skip it and came back if she/he needs that when reading Sect. 2 and 3, where we present the two specifications and the followed guidelines. Finally in Sect. 4 we try to evaluate our specifications by showing theirs characteristics.

## 1 Formal Basis

In this section we shortly present the formalism used in our approach, just what is need for handling *INVOICE*; for complete presentations see [7, 5].

**Formal models** Data structures are formally modelled by many-sorted first-order signatures (algebras with predicates), see [13].

To model systems we use labelled transition systems, see [8]. A *labelled transition system* (shortly *lts*) is a triple $(ST, L, \rightarrow)$, where $ST$ and $L$ are two sets, the *states* and the *labels* of the system, and $\rightarrow \subseteq ST \times L \times ST$ is the *transition relation*. A triple $(s, l, s') \in \rightarrow$ is said a *transition* and is usually written $s \xrightarrow{l} s'$.

A system S is thus modelled by an lts $(ST, L, \rightarrow)$ and an initial state $s_0 \in ST$; the states reachable from $s_0$ represent the intermediate (interesting) situations (stages) of the life of S and the transitions between them the possibilities of S of passing from a state to another one. It is important to note that here a transition $s \xrightarrow{l} s'$ has the following meaning: S in the state $s$ has the *capability* of passing into the state $s'$ by performing a transition, where the label $l$ represents the interaction with the external (to S) world during such move; thus $l$ contains information on the conditions on the external world for the capability to become effective, and on the transformation of such world induced by the execution of the action; so transitions correspond to *action capabilities*.

An lts can be represented by a many-sorted first-order structure $A$ on a signature with at least two sorts, *state* and *label*, whose elements correspond to the states and the labels of the system respectively, and with a predicate $\_ \xrightarrow{-} \_ : state \times label \times state$ representing the transition relation. The triple consisting of the carriers associated by $A$ to *state* and *label* and the interpretation of $\rightarrow$ in $A$ is the corresponding lts. If the system uses some data structures, then $A$ will have also sorts different from *state* and *label*, with the relative operations and predicates.

The $\Sigma$-first-order structures corresponding to lts's are called *LT-structures*.

**Axiomatic (property-oriented) specifications** To express the requirements on a system we use the first-order branching-time temporal logic with edge formulae and equality of [7], shortly presented below.

Let $L$ be an LT-structure. We need the following technical definitions. $PATH$ denotes the set of the *paths* on the associated lts, i.e. the set of all sequences of transitions having form either:

$s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \ldots$

$s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \ldots\ s_n,\ \ n \geq 0$ and there do not exist $l$, $s'$ s.t. $s_n \xrightarrow{l} s'$ in $L$

where for all $i \geq 0, s_i \xrightarrow{l_i} s_{i+1}$ in $L$.

$FirstS(\sigma)$ denotes the *first state* of $\sigma$; and $FirstL(\sigma)$ denotes the *first label* of $\sigma$, if exists, i.e. if $\sigma$ is not just a state.

Given $\sigma = s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \ldots$ and $h \geq 0$, if $s_h$ exists, then $\sigma_{|h}$ denotes the path $s_h\ l_h\ s_{h+1}\ l_{h+1}\ s_{h+2}\ \ldots$, otherwise it is undefined.

The set of *formulae*, denoted by $FOR$, and the sets of *path formulae*, denoted by $PF$, on $\Sigma$ and variables $X$ are defined as follows.

**formulae**

- $P(t_1, \ldots, t_n) \in FOR$ — for each predicate $P: s_1 \times \ldots \times s_n$, $t_i$ terms
- $t_1 = t_2 \in FOR$ — $t_1$, $t_2$ terms with the same sort
- `not` $\phi_1$, `if` $\phi_1$ `then` $\phi_2$, `for all` $x$: $\phi_1 \in FOR$ — $\phi_1, \phi_2 \in FOR$, $x \in X$
- $t$ `in any case` $\pi \in FOR$ — $t$ term of sort *state*, $\pi \in PF$

**path formulae**

- `[ x.` $\phi$ `]`$\in PF$ — $x \in X$ of sort *state*, $\phi \in FOR$
- `< x.` $\phi$ `>`$\in PF$ — $x \in X$ of sort *label*, $\phi \in FOR$
- $\pi_1$ `until` $\pi_2 \in PF$ — $\pi_1, \pi_2 \in PF$
- `after` $\pi \in PF$ — $\pi \in PF$
- `not` $\pi$, `if` $\pi$ `then` $\pi'$, `for all` $x$: $\pi \in PF$ — $\pi, \pi' \in PF, x \in X$

Let $L$ be a $\Sigma$-structure and $V$ a variable evaluation of $X$ in $L$; then validity is defined as follows:

- $L, V \models P(t_1, \ldots, t_n)$ iff $\langle t_1^{L,V}, \ldots, t_n^{L,V} \rangle \in P^L$
  ($t^{L,V}$ interpretation of $t$ in $L$ under $V$, $P^L$ interpretation of $P$ in $L$)
- $L, V \models t_1 = t_2$ iff $t_1^{L,V} = t_2^{L,V}$
- $L, V \models t$ `in any case` $\pi$ iff for each $\sigma$ s.t. $FirstS(\sigma) = t^{L,V}$, $L, V, \sigma \models \pi$
- $L, V, \sigma \models$ `[ x.` $\phi$ `]` iff $L, V[FirstS(\sigma)/x] \models \phi$
- $L, V, \sigma \models$ `< x.` $\phi$ `>` iff $FirstL(\sigma)$ is defined and $L, V[FirstL(\sigma)/x] \models \phi$
- $L, V, \sigma \models \pi_1$ `until` $\pi_2$ iff
  there exists $j \geq 0$ s.t. for all $h$, $0 < h < j$, $L, V, \sigma_{|h} \models \pi_1$ and $L, V, \sigma_{|j} \models \pi_2$
- $L, V, \sigma \models$ `after` $\pi$ iff $\sigma_{|1}$ is defined and $L, V, \sigma_{|1} \models \pi$
- `not` $\phi$, `if` $\phi_1$ `then` $\phi_2$, `for all` $x$`:` $\phi$, `not` $\pi$, `if` $\pi_1$ `then` $\pi_2$,
  `for all` $x$`:` $\pi$ as usual

$\phi$ is *valid* in $L$ (written $L \models \phi$) iff $L, V \models \phi$ for all evaluations $V$.

The formulae of our logic include the usual ones of first-order logic with equality, and formulae built with the transition predicate (arrow).

The formula $t$ `in any case` $\pi$ can be read as "for every path $\sigma$ starting in the state denoted by $t$, the path formula $\pi$ holds on $\sigma$".

The formula `[ x.` $\phi$ `]` holds on the path $\sigma$ whenever $\phi$ holds at the first state of $\sigma$; while the formula `< x.` $\phi$ `>` holds on the path $\sigma$ if $\sigma$ is not just a single state and $\phi$ holds at the first label of $\sigma$.

In the above definitions we have used a minimal set of combinators; but it is possible to define other, derived, combinators as:

`eventually` $\pi =_{\text{def}}$ `true until` $\pi$
(eventually the property represented by $\pi$ will hold)

`forever` $\pi =_{\text{def}}$ `not eventually not` $\pi$
(the property represented by $\pi$ will hold forever)

$t$ `in at least a case` $\pi =_{\text{def}}$ `not` $t$ `in any case not` $\pi$ (at least in one case, i.e. the property represented by $\pi$ holds in at least one path).

A *specification* is a pair: a signature $\Sigma$ with at least two sorts for states and labels and a transition predicate (arrow), plus a set $AX$ of formulae in $FOR$; its semantics is the class of all $\Sigma$-LT-structures $L$ s.t. for all $\phi \in AX$ $L \models \phi$.

## 2 Requirement Specification of *INVOICE* (Case 1)

We present how the specification of the *INVOICE* requirements has been produced following our method.

*The instructions and the questions posed by the method will be written in this way;* while [the answers for the *INVOICE* case are written in this way.]

Each subsection corresponds to a part of the document presenting the requirements.

### 2.1 Natural Description

*Require to the client a document in whatever format (natural language text, diagrams, pictures, ...) describing the system that he wants; this is called the "natural description".*

[The natural description is the just the text proposed by the Workshop Organizers.]

This document is the starting point of the development process. In the following, ambiguities, inconsistencies, incompleteness found in the natural description must be reported in another document, the "Shadow Spots"; if when giving the requirements such points have been settled in some way (e.g. by making a particular choice) that should be recorded in the Shadow Spots too, together with motivations. Sometime the problematic points are too many or too relevant and so the natural description is not apt to be the basis for defining the requirements; in such cases we need to interact with the client to get a new improved natural description.

[The Shadow Spots of *INVOICE* case 1 are reported in Sect. 2.4. ]

### 2.2 Border Determination

*Determine the* universe*, i.e. the smallest closed system (no interactions with its external world) including all entities mentioned in the natural description.*

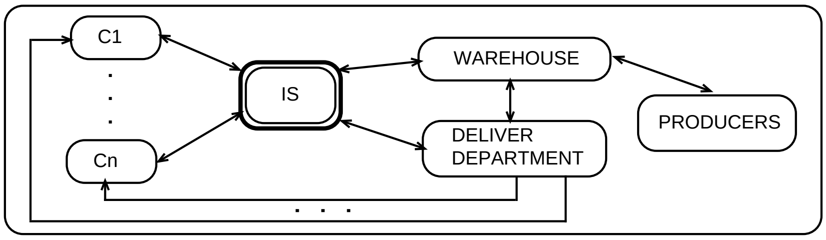
[ The first line of the natural description "The subject is to invoice orders" is already ambiguous; indeed it is not clear who/what invoices the orders. We have assumed that the information system (shortly IS) of some company automatically takes care of invoicing the orders. Another possibility is that there is a clerk that sends commands to IS requiring to invoice some orders; in this case IS either will signal that there is not enough product, or that the order is not pending, or will change the state of the order.

We assume that the orders are static (a data structure).

IS interacts with

- the *clients* (they send orders, receive invoices, pay and receive the products),
- the *warehouse* (it informs IS when products are added to the stock),
- the *deliver department* (it is informed by IS on the invoiced orders and then delivers the ordered products);

thus the clients, the warehouse and the deliver department are in the universe. Some producers (e.g. factories, trading companies) send the products to the warehouse, and so also they are in the universe; other entities interacting with the producers could not be relevant to this case.




]

*(1) Find the components of the universe belonging to the system we have to develop (system of interest); the remaining ones belong to the "application domain".*

*(2) Find the domain components s.t. in the natural description there is some information about them, relevant w.r.t. the system of interest.*

*In the following steps you must specify the system whose components are those determined at points (1) and (2).*

To clarify what we mean by "relevant", here there some examples of possible relevant information for *INVOICE*: "some particular clients can cause troubles if their orders are not serviced within one month", or "the deliver department handles overseas orders on Tuesday".

In our framework, the specification of a system does not include properties on its external environment, and so the relevant parts of the application domain must be included in the specified system.

[The system of interest is IS, we have no relevant information on any domain component, thus the system we are going to specify is just IS. ]

### 2.3 Specification

In our framework a system is modelled by an lts represented by a first-order structure, where two sorts correspond to the states and the labels of the lts and an arrow predicate to the transition relation, other sorts describe the used data. Thus the specification of a system is split in 4 parts: used data structures, states, interactions (labels), and activity (transition relation). Here we consider only simple systems, since IS is so.

**Basic Data Structures** *Determine which are the data structures used in the system. Specify them.*

In our framework a data structure is modelled by a first-order structure; a requirement specification is just a signature $\Sigma$ plus a set of first-order formulae with equality built on it, and determines a class of $\Sigma$-structures: all those in which the formulae hold. We assume that each data structure has at least a sort, named as the structure, but using low-case letters. The guidelines to specify a data structure are as follows.

*Has it any sub-data structure? If so, specify them.*

*Determine the constructors to represent the elements of the main sort, and the operations and predicates we need to operate on them.*

*Find the essential properties on them, and express such properties by using first-order formulae.*

There are further detailed instructions for the last point, not reported here, guiding to find all properties, asking, e.g., for each pair of predicates if there is any relationships between their truth, see [9].

[ The basic data structures are: REFERENCE, QUANTITY and ORDER.

```
pr  _ <= _: quantity quantity (total order)
op  _ - _: quantity quantity
------------------------------------------
if q' <= q then q - q' <= q
```

QUANTITY

```

```

REFERENCE

```
[QUANTITY]   [REFERENCE]

op Ref: order -> reference
op Quantity: order -> quantity
**  do two elements of type "order" correspond to the same order
pr  Same_Order: order order
------------------------------------------
Same_Order(o,o)
```

]

ORDER

The box is the icon for data structures; the double box denotes requirement specifications (the metaphor tries to suggest a pile of boxes, since a requirement specification determines many, usually infinite, data structures). The constructors, operations and predicates are defined in the first part of the box, they give the interface of the structure; the properties are in the second part. The boxes with inside a name represent sub-data structures; the referred data structures should have been defined before.

These specifications are really simple, since we just need to compare and subtract quantities. The predicate `SameOrder` has been introduced to properly express some properties; indeed we need to check whether two elements of sort `order` refer to the same order, to avoid, e.g., that the same order is both pending and invoiced. In any sensible implementation we should have a key to identify orders, e.g. client name and data.

**Interactions** *Determine the interchanges with the external environment that the system must have.*

In our framework the interactions (labels of the lts modelling the system) are represented by a data structure with a sort named `lab-name`, where `NAME` is the name of the system.

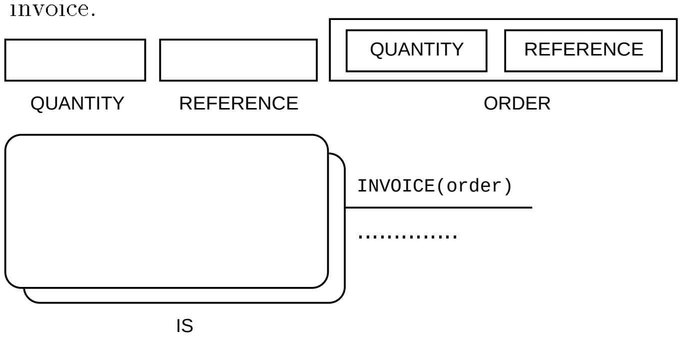
*Determine the various kinds of the interactions (i.e. classes of them having common characteristics).*

*Each kind of interactions should be informally specified by a sentence having form "to (adverbs) verb (complements)"; e.g. "to correctly send a message", "to break down", "to receive an order from a user". Each complement in one of the above sentences should be described by the elements of some basic data structures.*

*Is it possible that different interactions of the same or of different kinds may be performed simultaneously?*

*If the answer is no, for each kind we introduce a constructor to represent the interactions of such kind; otherwise for each kind we introduce a predicate checking whether a label includes an interaction of such kind.*

[The interactions of IS are just of one kind: "to invoice an order"; they are interactions with the deliver department and with a client, which will receive the invoice.



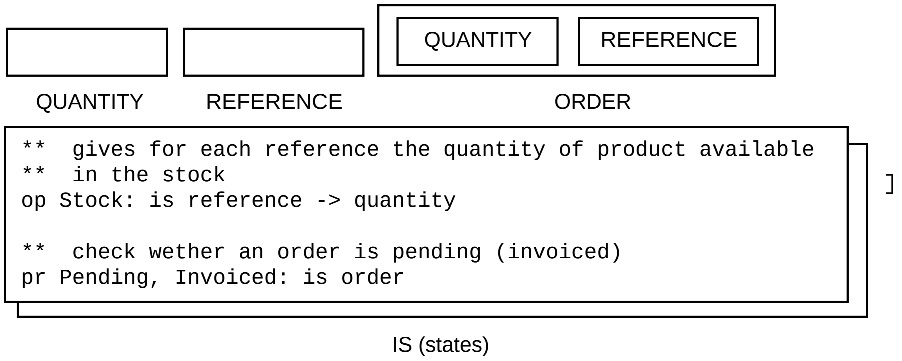

]

Graphically interactions are represented by lines leaving the icon of the system (a box with round angles). The dots remember that `INVOICE(order)` are not the unique interactions of the system, during its development more interactions may be added. Above the drawing we report schematically the basic data structure part, to have at hand which are the available data.

**States** *Determine the relevant intermediate situations that the system may reach along its life (the states); more precisely since at this level it is really hard to be able to precisely define them, determine which are the predicates and the operations extracting from them the relevant information.*

In our framework the "states" are just the states of the lts modelling the system and are represented by a first-order structure, with a sort named `name`, where `NAME` is the name of the system.

[In any state of IS we must know which are the pending and the invoiced orders; in our opinion the simplest way to extract this information is by two predicates asking whether an order is pending/invoiced (alternatively we may have two operations returning sets of orders). We need also to know in any state which quantities of products are in the stock; this is done by an operation returning the available quantity for each product reference.




Since states are a data structure, we use the box as their icon. Also in this part we report schematically the basic data structure part.

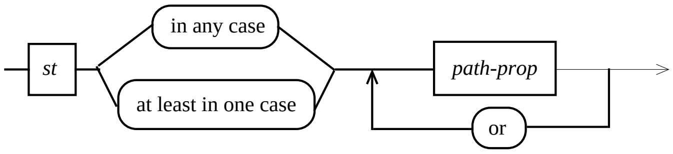
**Activity** In our framework the activity of a system is the transition relation of the lts modelling it, and it is represented by a ternary predicate
`_ -- _ -->_: syst lab-syst syst`; the properties on the activity are then expressed by formulae of the temporal logic introduced in Sect. 1.

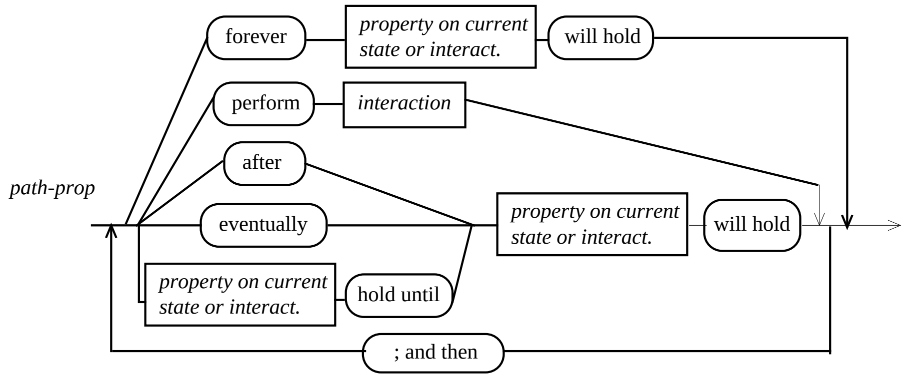
The guidelines concerning the activity are more complex, and we present them step by step. They ask to look for informal properties, guiding to produce the informal specification, but such properties are organized in a way to be easily transformed into formulae of the used logic.

It may happen that the system has a property that does not fit the proposed schemas, or that afterwards being informally stated cannot be transformed into a formula. There are two possibilities:

– the property concerns aspects of the system not expressible in our framework, e.g. probabilistic information or strict real-time. At the moment an extension of our approach has been developed to cover particular real time properties (see the case studies [10, 11]); if that it is not enough, then you can either leave such property as informal annotation, or choose another method.
– The property is about the happening along the time of some interactions; our experience shows that frequently it could be rearranged till to have the required format by adding more operations/predicates on the states. This is not over-specification; we are just pointing out that some information must be recorded in some way in the states.

In the following "*future-prop*(*st*)" denotes a property on the possible future evolutions of *st*, a state of the system, whose possible informal forms are given by the following syntactical diagrams.






An example of *future-prop* is (forgot the English grammar)
"it starts; and then the heath is less than 3 until stops; and then the heath is 0" which corresponds to the formula of our logic

```
<i.i = START> and after
([x. Heath(x)<3 ] until (<i.i = STOP> and after [x.Heath(x)=0])).
```

*For each predicate pr on the states look for properties of the following form:*

*(1) relationships with other predicates and operations on the states*

*(2) if $pr(st, a_1, \ldots, a_n)$ holds/does not hold, then future-prop(st)*

*(3) if future-prop(st), then $pr(st, a_1, \ldots, a_n)$ holds/does not hold*

*(4) if $pr(st, a_1, \ldots, a_n)$ holds/does not hold, st goes to $st'$ by performing interaction i and $pr(st', a_1, \ldots, a_n)$ does not hold/holds, then property on i*

[ There are two predicates on the states of IS, `Pending` and `Invoiced`, and the resulting properties are as follows. Notice that there are no (*2*) properties for `Pending`, since from the natural description it seems possible that a pending order stays pending forever, also if the ordered quantity product is in the stock.

```
Pending

(1) if Pending(o,is) then not exists o': o =/= o' and
    Same_Order(o,o') and (Pending(o',is) or Invoiced(o',is))

(3) if is in at least a case <i. i = INVOICE(o)>  then Pending(is,o)

(4) if Pending(o,is) and is -- i --> is' and not Pending(o,is') then
    i = INVOICED(o)
```
]
```
Invoiced

(1) if Invoiced(o,is) then not exists o':  o =/= o' and
    Same_Order(o,o') and (Invoiced(o',is) or Pending(o',is))

(2) if Invoiced(o,is) then is in any case forever [x. Invoiced(o,x)]

(4) if not Invoiced(o,is) and is -- i --> is' and Invoiced(o,is') then
    i = INVOICED(o)
```

*For each operation op on the states look for properties of the form:*

*(1) relationships with other predicates and operations on the states*

*(2) if property on op(st,$a_1,\ldots,a_n$), then future-prop(st)*

*(3) if future-prop(st), then property on op(st,$a_1,\ldots,a_n$)*

*(4) if st goes to st′ by performing interaction i and op(st,$a_1,\ldots,a_n$) is different from op(st′,$a_1,\ldots,a_n$), then property on i*

[There is only one operation on the states of IS, `Stock`, and the found properties are as follows.

```
Stock
(3) if  is in at least a case <i.i=INVOICE(o)> then
    Quantity(o) <= Stock(is,Ref(o))

(4) if is -- i --> is’ and Stock(is’,r) < Stock(is,r) then
    exists o:
    i = INVOICE(o) and Stock(is,Ref(o))-Stock(is’,Ref(o))=Quantity(o)
```
]

*For each interaction kind represented by constructor IK*

**pre-post condition**

*(1) if st goes into st′ by performing interaction IK($a_1,\ldots,a_n$), then property on st and on st′*

**reaction in the future**

*(2) if st goes into st′ by performing interaction IK($a_1,\ldots,a_n$), then future-prop(st′)*

**incompatibility**

*(3) if st goes into st′ by performing interaction IK($a_1,\ldots,a_n$), then it cannot go into st″ by performing interaction i s.t. property on i and st″*

**vitality**

*(4) if property on st, then there exists st′ s.t. st goes into st′ by performing interaction IK($a_1,\ldots,a_n$) and property on st′, $a_1,\ldots,a_n$*

*(5) if property on st, then in any case/at least in a case st′ eventually will perform IK($a_1,\ldots,a_n$) s.t. property on $a_1,\ldots,a_n$*

[There is only one interaction kind `INVOICE` and the resulting properties are as follows. Notice the lack of vitality properties; nothing is required about finally invoicing the pending orders.

```
INVOICE
(1) if is -- INVOICE(o) --> is’ then
    - Pending(o,is) and Invoiced(o,is’)
    - Quantity(o) <= Stock(Ref(o),is)
    - Stock(Ref(o),is’) = Stock(Ref(o),is) - Quantity(o)
    - for all o’: if o’ =/= o then
       Pending(o’,is) iff Pending(o’,is’) and
       Invoiced(o’,is) iff Invoiced(o’,is’)
    - for all r: if r =/= Ref(o) then
         Stock(is,r) = Stock(is’,r)
```
]

Here and in the following we write `if cond then - cond1 ... - condk` instead of `if cond then cond1 and ... and condk`

### 2.4 Shadow Spots

[ The first line of the natural description "The subject is to invoice orders" is already ambiguous; indeed it is not clear who/what invoices the orders. We have assumed that the information system (shortly IS) of some company automatically takes care of invoicing the orders. Another possibility is that there is a clerk that sends commands to IS requiring to invoice some orders; in this case IS either will signal that there is not enough product, or that the order is not pending, or will change the state of the order.

The most relevant shadow spot is the lack of a policy for handling the pending orders (we may have correct implementations serving the orders in a truly unfair way); furthermore there are no "vitality" requirements, saying for example that a pending order must eventually be invoiced when the ordered quantity of products is in the stock.

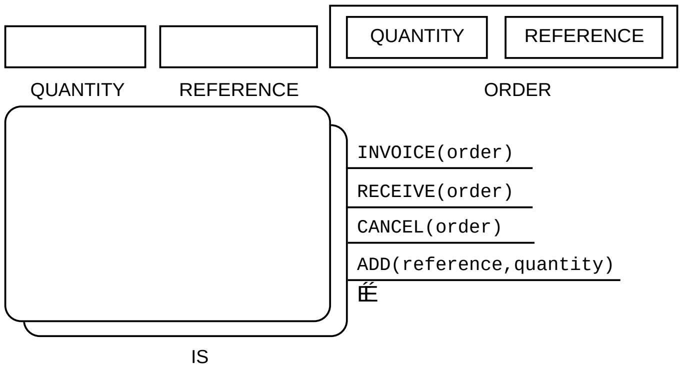
We have assumed that invoicing an order cannot be performed simultaneously with other activity (including invoicing another order); because nothing was said in the natural description and because for systems of this kind "timed related" aspects (as the possibility to have two orders invoiced simultaneously) are not relevant.

It seems sensible to have some keys to identify orders, to easily identify them.]

## 3 Requirement Specification of *INVOICE* (Case 2)

If we consider the case 2 of *INVOICE*, then IS has more ways to interact with outside (it can receive orders, have an order cancelled and be informed when products are added to the stock). The specification of case 2 is simply obtained by modifying that of case 1; the disciplined way to produce the specifications results in structured specifications making easy to modify them in the case of changes in the natural description. Recall that each property could be equipped with info making precise which was the question introducing it.

Starting with the document produced in Sect. 2 and the new natural description given by the Workshop Organizers, we have to add a plus operation to QUANTITY, and to change the interaction and the activity parts of IS.



In the activity part we have to change the properties of type (4) for `Pending` and `Stock`, because there are new ways to modify them,

```
(4) if Pending(o,is) and is -- i --> is' and not Pending(o,is') then
    i = INVOICE(o) or i = CANCEL(o)

(4) if not Pending(o,is) and is -- i --> is' and Pending(o,is') then
    i = RECEIVE(o)

(4) if is -- i --> is' and Stock(is,r) < Stock(is',r) then
    exists q: i = ADD(q,r) and Stock(is',r) = Stock(is,r) + q
```

and to add the properties relative to the new interactions:

```
RECEIVE
(1) if is -- RECEIVE(o) --> is' then
    - not Pending(o,is) and not Invoiced(o,is)
    - Pending(o,is')
    - for all r: Stock(r,is) = Stock(r,is')
    - for all o': if o' =/= o then
          Pending(o',is) iff Pending(o',is') and
          Invoiced(o',is) iff Invoiced(o',is')
```

```
CANCEL
(1) if is -- CANCEL(o) --> is' then
    - Pending(o,is) and not Pending(o,is')
    - not Invoiced(o,is')
    - for all o': if o' =/= o then
          Pending(o',is) iff Pending(o',is') and
          Invoiced(o',is) iff Invoiced(o',is')
    - for all r: Stock(r,is) = Stock(r,is')
```

```
ADD
(1) if is -- ADD(o) --> is' then
    - Stock(r,is') = Stock(r,is) + q
    - for all r: if r=/= r' then Stock(r',is) = Stock(r',is')
    - for all o:
          Pending(o,is) iff Pending(o,is') and
          Invoiced(o,is) iff Invoiced(o,is')
```

## 4 Conclusions

We have tried to present a "method" to formally capture system requirements, or to be more precise the part of the method needed to handle *INVOICE*; thus here we have not considered structured systems. For us method means to give very precise "guidelines" to fully guide the user to produce the specification.

In our opinion the small experiment on *INVOICE* has worked satisfactorily; indeed

- everything known about the system has been formalized (i.e., what given by the Workshop Organizers), but nothing more;

– many lacking relevant information about the system have been found (look at the Shadow Spots part), among them who is invoicing, a policy for deciding in which order to invoicing the orders, the need of a key to identify orders.

The most relevant features of our specifications of *INVOICE* are:

– no over-specification at all, exactly only what said by the Workshop Organizers has been formalized (also for the slightly artificial case 1);
– rich modularity; that has helped to pass from case 1 to 2;
– high level of abstraction; e.g., consider the interaction `INVOICE`, it could correspond to call another process (in the case of a distributed implementation where each component of the universe is connect by Internet), or just to print two pieces of paper: the invoice for the client and a memo for deliver department, in a more traditional realization.

The last point is positive in our opinion, since abstraction could help to master complexity; but that could be negative for someone. Our specification is too far from the system to implement at the end, and so it may look unfamiliar to the developers, and there is the need of several steps before the coding.

The proposed method with its guidelines to get the fundamental properties of the system has worked for the very simple *INVOICE* case, and also on something of more complex, see the various lift examples in [9]. However to asses its value it has to be experimented on industrial case studies; and, still better, experimented by some industrial developers, to see if the proposed concepts and techniques are not too far from what they are used to.

We think that by considering more specialized classes of systems, e.g. purely reactive systems, we can tune up the guidelines getting a more effective method.

Trying to compare our approach to the requirement specification with others, we can say:

– it is formal, we give formal specifications also if they are presented using a graphical notation;
– its specifications have strictly corresponding informal specifications (not reported here for lack of room) where formulae and declarations are replaced by English sentences;
– it aims to support the development of “systems” considering the reactive, concurrent, parallel aspects, and so it is not similar to Z or B, where such aspects have to be considered by ad hoc extensions;
– similarly to object-oriented methods it offers some conceptual entities for modelling the entities in the universe, but it does not consider only one kind of entity. Indeed we assume to have systems distinguished by data structure, that systems are further distinguished in simple and structured (those having subcomponents cooperating among them) and that the components are still distinguished in active and passive or inert. We think that to use only one conceptual entity to model the real world entities may lead to misunderstandings and to unnecessary complications.

– it asks many questions to the person writing the requirement specification; however in many cases that require some further interactions with the client, to avoid to detect too late inconsistencies between hers/his ideas and the developed system. Clearly some of the questions raised by myself in the role of developer could be trivial for real developers, that knows much more about the considered application domain.

## References


1. E. Astesiano, F. Morando, and G. Reggio. The SMoLCS Toolset. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. of TAPSOFT '95*, number 915 in L.N.C.S. Springer Verlag, Berlin, 1995.
2. E. Astesiano and G. Reggio. Specifying Reactive Systems by Abstract Events. In *Proc. of Seventh International Workshop on Software Specification and Design (IWSSD-7)*. IEEE Computer Society, Los Alamitos, CA, 1993.
3. E. Astesiano and G. Reggio. Formally-Driven Friendly Specifications of Concurrent Systems: A Two-Rail Approach. Technical Report DISI–TR–94–20, DISI – Università di Genova, Italy, 1994. Presented at ICSE'17-Workshop on Formal Methods, Seattle April 1995.
4. E. Astesiano and G. Reggio. A Dynamic Specification of the RPC-Memory Problem. In M. Broy, S. Merz, and K. Spies, editors, *Formal System Specification: The RPC-Memory Specification Case Study*, number 1169 in L.N.C.S. Springer Verlag, Berlin, 1996.
5. E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. Technical Report DISI–TR–96–20, DISI – Università di Genova, Italy, 1996.
6. E. Coscia and G. Reggio. Deontic Concepts in the Algebraic Specification of Dynamic Systems: The Permission Case. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specification*, number 1130 in L.N.C.S. Springer Verlag, Berlin, 1996.
7. G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2), 1997.
8. R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.
9. G. Reggio. A Guide to the Use of the SMoLCS Method. Technical report, DISI – Università di Genova, Italy, 1998.
10. G. Reggio and E. Crivelli. Specification of a Hydroelectric Power Station. Technical Report DISI–TR–94–17, DISI – Università di Genova, Italy, 1994.
11. G. Reggio and V. Filippi. Specification of a High-Voltage Substation. Technical Report DISI-TR-95-09, DISI – Università di Genova, Italy, 1995.
12. G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In J. Fitzgerald and C.B. Jones, editors, *Proc. FME 97 - Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in L.N.C.S. Springer Verlag, Berlin, 1997.
13. M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B. Elsevier, 1990.