

Contents

2 SDL: a Specification and Description Language	
P. Poizat	1
2.1 Overview of SDL	1
2.2 Analysis and Specification of Case 1	2
2.2.1 System Structure	2
2.2.2 Process Graphs	4
2.2.3 Sort Definitions	7
2.2.4 Comments on the First Case-Study	11
2.3 Analysis and Specification of Case 2	11
2.3.1 System Structure	11
2.3.2 Process Graphs	13
2.3.3 Sort Definitions	14
2.4 The Natural Language Description of the Specifications	16
2.4.1 Case 1	16
2.4.2 Case 2	16
2.5 Conclusion	16

2 SDL: a Specification and Description Language based on an Extended Finite State Machine Model with Abstract Data Types

P. Poizat

2.1 Overview of SDL

SDL (Specification and Description Language) is a specification language with a formal semantics that has been developed and standardized [2] by ITU-T¹. It is based on an extended finite state machine model for the description of system behaviour together with abstract data types features. Recent developments such as non-determinism and object-oriented features have led to the definition of SDL-92 [3]. SDL comes with two equivalent notations, GR (Graphical Representation) and PR (Phrase Representation). Here we use the GR representation since it is more readily understood. Please refer to SDL related literature [2,3] for GR and PR equivalence.

SDL is mainly used for the specification of telecommunication protocols and services but may be used more widely on any reactive system. SDL is supported by several tools, like Verilog ObjectGEODE and Telelogic Tau.

General semantic aspects. SDL is a “mixed specification language” in the sense that it has both a dynamic part – for communication aspects – and a static part – coping with data types.

An SDL specification describes a *system* as several communicating extended finite state machines. These machines exchange messages called *signals* that may carry typed data values (and hence provide a simple way for data exchange between them). The machines support the *process* concept. A process is a common description of the behaviour that is shared by its *instances*. Each process instance has a single different process identifier (PID). These PIDs are particularly useful for addressing messages between processes instances. The processes own data values (hence the term “extended” for the machines). Exchange of data between processes is possible by way of signals or variable sharing.

The system may also communicate with the external environment which is assumed to behave like any SDL process instance (for example, it is assumed to own a PID different from all the system components PIDs).

SDL offers basic types (types are called *sorts* in SDL) for use in process behaviours. These include the usual ones like integers or booleans but also time

¹ International Telecommunication Union, has replaced in 1993 the former International Telephone and Telegraph Consultative Committee (CCITT).

(to model timers), duration and PId (to work on process instances). User-defined sorts can be defined by means of *abstract data types*: constants (called *literals*), typed signatures for the operations (called *operators*) available on the sort and equations (called *properties*) for their semantics. SDL-92 defines object-oriented concepts. We will not present them here since we do not use them.

2.2 Analysis and Specification of Case 1

There are various ways to make a specification from scratch (informal requirements) in SDL as in other formal description techniques that have both a dynamic and a static (data type) part [7,8]. Following the usual approach, we will first work on the system structure, then we will make the process graphs and finally define the sorts used in the previous steps.

2.2.1 System Structure

The first task is to find out all the system functionalities (signals triggered by the system environment). The system is about invoicing orders and the unique available functionality is `invoice`. A user (in the environment) is assumed to send the corresponding signal to the system.

This operation raises several questions. We will give them together with the solution we adopted.

Question 1: *What are the invoice operation parameters? Does the user just ask the system to invoice all the orders it can or does the user ask the system to invoice a particular order?*

Answer: *(1) The user asks to invoice a particular order. The invoice signal carries some information on the order that is to be invoiced (its identifier to stay at an abstract level). (2) The user asks to invoice all invoiceable orders (but in which order?). (3) The system runs independently to invoice one (or several orders); in this case, the invoice operation is not triggered by the environment. We choose the first solution.*

We then have to give the `invoice` operation dynamic semantics in more detail.

Question 2: *What shall the system do if the order does not exist?*

Answer: *Return a specific signal (named `error`) to the user.*

Question 3: *What shall the system do if the order cannot be invoiced?*

Answer: *Return an error signal. Another solution would have been to save the invoice signal for later use.*

Question 4: *May the user ask to invoice the same order several times?*

Answer: *Surely not. This should output the error signal.*

We also have to give more details on the operation conditions.

Question 5: *Under which conditions is invoicing an order possible?*

Answer: *At this (abstract) level, we assume an `invoiceable` boolean operation in some SDL sort to check if an order may be invoiced. Questions on this operation's semantics will be deferred to the work on sorts in Sect. 2.2.3.*

Question 6: *Do there exist wrong orders (i.e. orders with products not referenced in stock)?*

Answer: *This could have been the case, and an error message could have been sent, somewhere! Since we do not have an operation to create orders in this case, this would make no sense. This question and related ones will be delayed to case 2.*

Now that we have the system functionalities, we split the system into subparts to have a good architecture to work on. This split may be done several times until we have a good level of detail with sequential processes running in parallel.

SDL offers the means to structure a specification. The system is made up of several connected *blocks*. There are *block substructures* and *block diagrams*. They differ only in the fact that block diagrams are at the end of the decomposition process. Whereas block substructures may contain other block substructures or block diagrams, a block diagram may only contain processes.

The connections are modelled by *channels*. Channels may not connect more than two blocks. They may carry the signals defined in their *signal list*. Channels are uni- or bi-directional. A signal list is associated to each direction. Channels between processes are called *signal routes*.

Connection points link block channels with their enclosing superstructure. Like the blocks, the channels may be decomposed into subcomponents (blocks and channels). This may be used for example when modelling unreliable media.

As with processes, blocks (including the system) and channels give a common definition shared by their instances. SDL has a simple name scope rule: any definition is available in the current and sub-blocks. Note that in blocks one may use definitions or references to remote specifications. References are useful as placeholders. They also provide a more readable structuring of systems and blocks by separating abstraction levels.

We model the case study with two subcomponents: a process that manages the stock and a process that manages the set of orders. We will have a single block for the system. This block will contain references to a `STOCK` process and a `SET_ORDERS` process. Note that this is a matter of choice. We might also have used a single component.

In order to model the other types (orders and products), we have to make a choice between active (process) and passive (sort) objects. We here make the choice to use passive objects. Orders could have been modelled using active objects with different PIDs as identifiers.

This decomposition leads us to ask questions about the subcomponent signal exchanges (between each other and the environment):

- What are the channels and signal routes between the system and the environment?
The `invoice` signal comes from the external environment (user) and is received by the `SET_ORDERS` process. If the order is not correct, an `error` signal is sent back.
- What are the channels and signal routes between inner blocks?
The `SET_ORDERS` process when receiving a correct `invoice` signal asks the `STOCK` using an `ask` signal if the order may be invoiced. The `STOCK` may then reply using either an `ok` or a `not_ok` signal.
- What are the new signal parameters?
The whole order information is to be used by the `STOCK`. Full orders will then be used as parameters for the `ask` signal. `ok` and `not_ok` (return signals) have no parameters.

All these questions and the corresponding answers lead to the system architecture given in Figs. 2.1 and 2.2. SDL notations are given in Fig. 2.2.

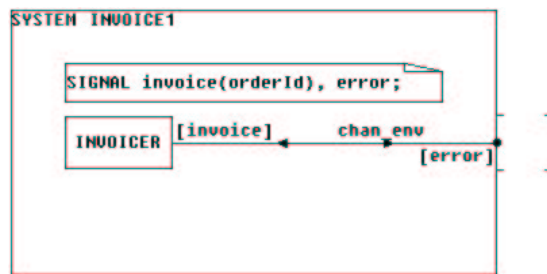


Fig. 2.1. The system architecture for case 1

SDL allows one to give an initial and a maximum number of instances in process references. In the case study, there is a unique `STOCK` instance and a unique `SET_ORDERS` instance.

2.2.2 Process Graphs

In this part, we specify the behaviour of the processes.

Process behaviours are given by means of process graphs. A process instance may be in several *states* where it may receive or send different sets of signals.

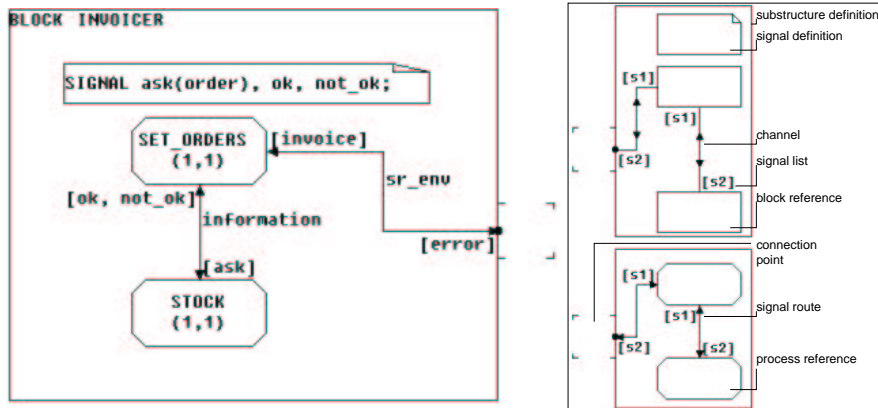


Fig. 2.2. The INVOICER block for case 1 / system structuring notation

Process instances are created (at system initialization or dynamically by another process instance in the same block) in a special state called the *initial state* (the states without names on the process graphs). The semantics for the behaviour of the system is given in terms of its instance process behaviours.

Process instance communication is *asynchronous*. Senders do not block on signal sending. Receivers own an (unlimited) buffer where valid signals (i.e. signals the process may treat in some state) are held until consumption or discard. If a message may be treated in the current state, the process instance initiates a *transition*, consumes the signal, does some optional *activity* and goes into the target state. If the message cannot be treated in the current state, the instance discards it (there is also a way to save the message for later). The buffers, also called *input ports*, behave in a first-in/first-out way (except for saved signals).

Signals are a way to exchange values. A process instance p may send signals (i) implicitly to the unique process instance connected to it via the system structure, (ii) to all processes instances linked to a certain signal route (via signal routes and channels), or (iii) to a specific process instance using process identifiers (remember there is a unique identifier for each process). Destination keywords include: *self* (the process instance itself), *sender* (the process instance that sent the last message), *offspring* (the last process instance created by p) or *parent* (the process instance that created p).

STOCK. The STOCK process receives requests from the SET_ORDERS process to ask if some order is *invoiceable*. As we have already said, we assume that the sort associated with the stock has an operation to reply to this question.

Question 7: *What happens when an order may be invoiced?*

Answer: *We assume there is an invoice operation defined on sort orderStock for this. This keeps abstraction at this level and delays the real answer to this question to the work on the sorts.*

The STOCK process behaviour is then very simple. It receives requests via the `ask` signal and answers them. SDL, being asynchronous, does not force the STOCK process to reply when the `ask` signal is received. There may be already some other signal in its buffer. So the question arises: to which process instance does STOCK reply? Using the `to sender` keywords of SDL, the STOCK, when treating a given `ask` signal, is able to reply to the exact sender of the signal.

The behaviour of the STOCK process is given in Fig. 2.3.

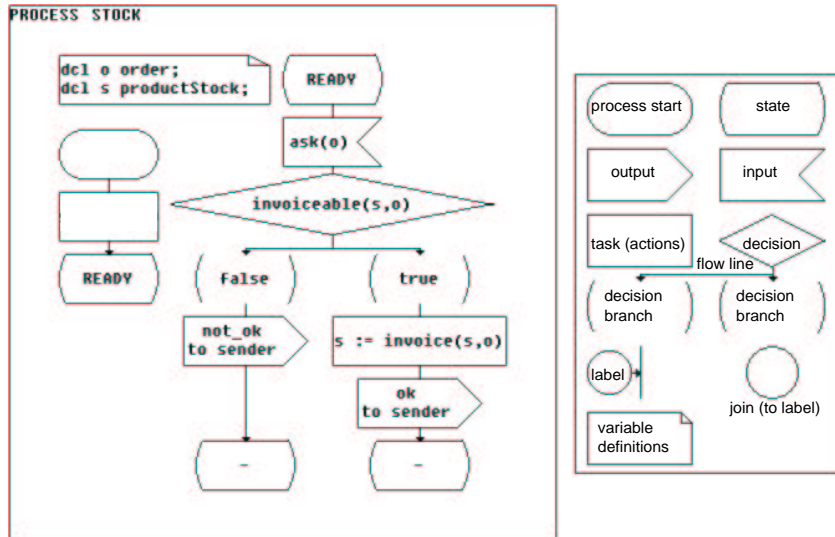


Fig. 2.3. The STOCK process behaviour for case 1 / process notation

SET_ORDERS. The SET_ORDERS process receives invoice requests from the environment. If the order does not exist or has already been invoiced, an `error` signal is output. Otherwise, the STOCK is asked for the order invoiceability. If the order is not invoiceable an error signal is output, otherwise the order is invoiced. Errors are returned (as signals) to the sender of the `invoice` signal. The PID of this sender has to be kept in a variable (`lastsender`).

Question 8: *Apart from passing the order from state pending to invoiced, what becomes of the order?*

Answer: (1) *The order may be suppressed or (2) it may be kept in the set of orders. We chose the second solution.*

We saw that the SET_ORDERS process sends `ask` signals to the STOCK in order to check the invoiceability of orders. Thereafter, the SET_ORDERS

process may either work on other invoicing requests after sending this signal or wait until reception of an `ok` or `not_ok` signal. Since the first approach could lead to orders affecting the stock several times before being invoiced, we will choose the first one.

The behaviour of `SET_ORDERS` is given in Fig. 2.4.

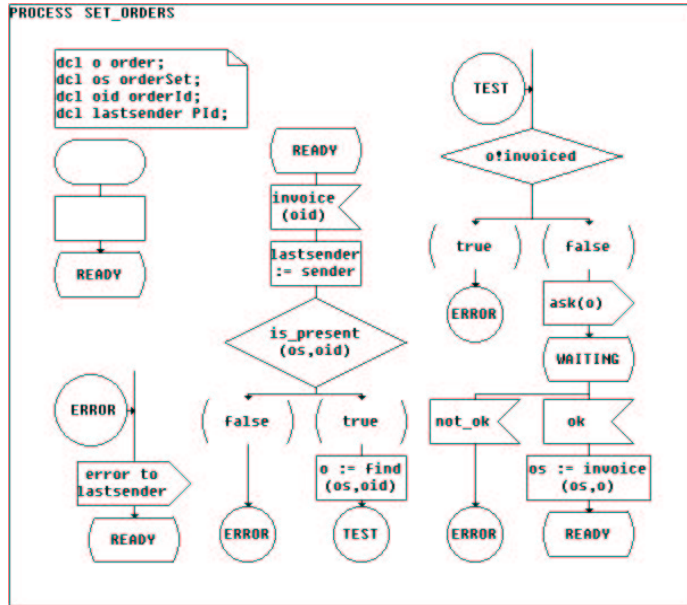


Fig. 2.4. The `SET_ORDERS` process behaviour for case 1

2.2.3 Sort Definitions

In earlier phases we focused on system decomposition or signal exchange conditions. Here we give the corresponding operator properties. For each sort we use a constructive approach. Basic operators are defined and then semantics for all other operators used in process graphs are given in terms of these basic operators.

The sorts used in earlier phases (process graphs) are: `orderId`, `order`, `orderSet`, and `productStock`. Other sorts are SDL predefined sorts (`PId`). Each non-SDL predefined sort has to be defined.

Sorts `orderId` (and `productId`²). These two sorts are used as identifiers. This may be achieved using the `Natural` sort. We use the SDL *syntype* concept that

² The `productId` sort will be used later on.

enables one to define (rename) a sort with a restricted (or here equal) set of values with respect to the type it is based on.

```

syntype orderId = Natural          syntype productId = Natural
endsyntype;                        endsyntype;

```

Sort order. Orders have an identifier. We assume identifiers are unique. Orders also contain references to certain quantities of products. Orders may be invoiced or (alternatively) pending. Therefore, we will use a *structure* type (close to records in programming languages) for orders. Elements in a structure may be accessed using a “!” notation (e.g. `o!id` yields the identifier of the order `o`).

Question 9: *How many references are there to an order?*

Answer: *(1) Only one reference or (2) a set of references (with the constraint that the products in these references are all different). Both solutions cope with the informal specification stating that “on an order we have one and only one reference to an ordered product of a certain quantity”. For simplicity, we choose to model orders with only one reference to a given product. Solution number two would have led us to take into account complex things such as “partially invoiceable” orders.*

Sort productRef. This sort models product references.

Question 10: *What is a productRef?*

Answer: *A productRef is made up of a product identifier and a quantity.*

Question 11: *What is a quantity? May it be negative?*

Answer: *A quantity may not be negative.*

In SDL, we may model such numbers using naturals.

```

newtype order struct                newtype productRef struct
  id orderId;                       id productId;
  ref productRef;                   qty Natural;
  invoiced Boolean;                 endnewtype;
endnewtype;

```

Sorts orderSet and productStock. These two sorts are sets. SDL enables one to define sets by means of the *powerset* generator. The usual operations on sets (`incl` to add an element, `del` to remove an element, `in` to test if an element is in a set, and `empty` for the empty set) are available (amongst others).

```

newtype basicOrderSet              newtype basicProductStock
  powerset(order)                  powerset(productRef)
endnewtype;                        endnewtype;

```

These basic sorts must be extended to define operators used in the process graphs that are not defined in basic sets. This can be done using the SDL *inheritance* concept. All that is defined in the parent sort is inherited, and more can be defined using the `adding` keyword. Partial inheritance (or renaming) can be specified on literals and/or operators.

The operators needed for sort `orderSet` are the following. An operator `invoice: orderSet, order → orderSet` (it takes an `orderSet`, an `order`, and it returns an `orderSet`) marks the order in the order set as invoiced. An operator `is_present: orderSet, orderId → Boolean` will be used to check if an order of a certain `orderId` is in the set. An operator `find: orderSet, orderId → order` is also needed to find the order corresponding to a certain identifier.

```

newtype orderSet
inherits basicOrderSet
  literals all;
  operators all;
adding
  operators
    invoice : orderSet, order -> orderSet;
    find : orderSet, orderId -> order;
    is_present : orderSet, orderId -> Boolean;
axioms
  for all os in orderSet (
    for all o1,o2 in order (
      for all oid in orderId (
        invoice(empty, o2) == empty;
        (o1!id = o2!id)==> invoice(incl(o1,os),o2) == incl(invoice(o1),os);
        (o1!id /= o2!id)==> invoice(incl(o1,os),o2) == incl(o1,invoice(os,o2));

        is_present(empty, oid) == false;
        (o1!id = oid)==> is_present(incl(o1,os),oid) == true;
        (o1!id /= oid)==> is_present(incl(o1,os),oid) == is_present(os,oid);

        (o1!id = oid)==> find(incl(o1,os),oid) == o1;
        (o1!id /= oid)==> find(incl(o1,os),oid) == find(os,oid);
      );););
endnewtype;

```

The operators needed for sort `productStock` are the following. Operators `invoice: productStock, order → productStock` and `invoiceable: productStock, order → Boolean` are needed.

Question 12: *When is an order invoiceable?*

Answer: *When the product it references is present in the stock and in a sufficient quantity.*

Question 13: *What is the effect of invoicing an order on the stock?*

Answer: *There are several solutions: (1) no effect, (2) the stock may be reduced by a corresponding amount, or (3) the corresponding amount of the required product may be marked as being reserved (for example until some customer pays for it). We choose the second solution*

```

newtype productStock
inherits basicProductStock
  literals all;
  operators all;
adding
  operators
    invoice : productStock, order -> productStock;
    invoiceable : productStock, order -> Boolean;
axioms
  for all s in productStock (
  for all o in order (
  for all p in productRef (
    invoice(empty,o) == empty;
    (o!ref!id = p!id)==> invoice(incl(p,s),o) ==
      incl(subtractQty(p,o!ref!qty),s);
    (o!ref!id /= p!id)==> invoice(incl(p,s),o) ==
      incl(p,invoice(s,o));

    invoiceable(empty,o) == false;
    (o!ref!id = p!id)==> invoiceable(incl(p,s),o) ==
      (o!ref!qty <= p!qty);
    (o!ref!id /= p!id)==> invoiceable(incl(p,s),o) ==
      invoiceable(s,o);
  )););
endnewtype;

```

Modifications in productRef. An operator `subtractQty: productRef, Natural → productRef` is needed due to prior sort definitions. Its definition makes use of `*!` operators. These operators are to be used in structure type definitions. The operator *Make!(...)* builds a structure from its fields, an operator like *IDExtract!(record)* is used to extract the value of some field *ID* in the record, and an operator like *IDModify!(record,value)* is used to replace the value of some field *ID* in the record.

```

newtype productRef struct
  ...
adding
  operators
    subtractQty : productRef, Natural -> productRef
axioms
  for all pr in productRef (
  for all qtySub in Natural (
    (qtyExtract!(pr) >= qtySub)==> subtractQty(pr,qtySub) ==

```

```

    qtyModify!(pr,qtyExtract!(pr) - qtySub);
  ););
endnewtype;

```

Modifications in order. An operator `invoice: order → order` is needed due to prior sort definitions.

```

newtype order struct
  ...
adding
operators
  invoice : order -> order
axioms
  for all o in order (
    invoice(o) == invoicedModify!(o,true);
  );
endnewtype;

```

2.2.4 Comments on the First Case-Study

The first case-study specifies that the stock and set of orders are always up to date. On the other hand, it is said that no other operation than invoicing should be defined. This causes a problem since SDL is dynamic and does not specify static properties (invariants) of the system. In order to have a fully working system, we should have modelled operations for initialization and adding of orders/products in stock.

2.3 Analysis and Specification of Case 2

Case 2 is an extension of case 1. This is reflected in the SDL specification.

2.3.1 System Structure

There is still the `invoice` functionality. New ones are `addProduct` to add a certain quantity of a given product in stock, `createOrder` to create new orders and `cancelOrder` to cancel an order.

Question 14: *What are their parameters?*

Answer: *addProduct takes a product identifier and a certain quantity as parameter. This pair is of sort `productRef` as seen earlier. createOrder takes an order as parameter. cancelOrder takes an order identifier (of sort `orderId`) as parameter.*

Question 15: *Are there any conditions on addProduct?*

Answer: *We choose to impose non-negative quantities. Another choice would have been to treat negative quantities and to either return an error when there is not a sufficient product quantity in stock, or to keep the request for later (saving the signal).*

Question 16: *What shall be done when `addProduct` is used to add a quantity of a product that does not exist in stock (yet)?*

Answer: *The product is created in stock with an initial amount equal to the quantity given in `addProduct`.*

Question 17: *Are there any conditions on `createOrder`?*

Answer: *This is not said in the informal specification. We assume the product reference should exist in stock, and if this is not the case return an error. Another choice would have been to save the `createOrder` signal.*

Question 18: *What shall be done if the order identifier already exists?*

Answer: *Return an error signal.*

Question 19: *What is/should be the initial status of orders?*

Answer: *Orders may be created in (1) any status or (2) in a pending state. We choose the second solution. If the order given as parameter for `createOrder` is invoiced, return an error signal.*

Question 20: *Are there any conditions on `cancelOrder`?*

Answer: *The order with the given identifier has to exist. If this is not the case, an error signal is returned.*

Question 21: *Can invoiced orders be cancelled?*

Answer: *No. Return an error signal if a `cancelOrder` is received for an invoiced order.*

As far as the system structure is concerned, the architecture of case 1 is still relevant for case 2. Again (see case 1) the structuring leads us to reply to the questions:

- What are the channel and signal routes?
The channel and signal routes for case 1 are kept. New signals corresponding to the new operations are added where needed. A new channel and a new signal route are created between the environment and the STOCK concerning the `addProduct` operation. A new signal is created for `SET_ORDERS` to ask the STOCK if the product referenced in a newly created order is valid or not. This signal will be called `askValid`. The STOCK will use `valid` and `not_valid` signals to answer.
- What are the new signal parameters?
Information needed for testing the validity of a product consists of a product identifier. Return signals have no parameters.

The architecture of case 2 is given in Figs. 2.5 and 2.6.

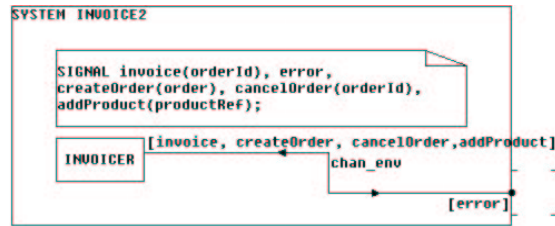


Fig. 2.5. The system architecture for case 2

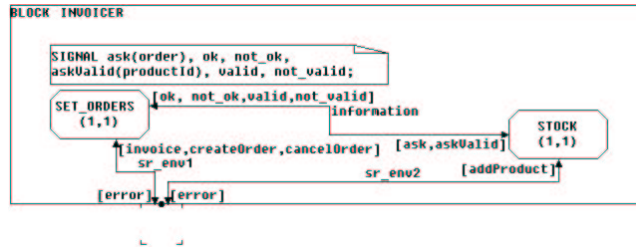


Fig. 2.6. The INVOICER block for case 2

2.3.2 Process Graphs

STOCK. We have to take into account the new communication scheme corresponding to the SET_ORDERS process asking the STOCK process if a given product identifier is valid or not. This will be modelled using a transition, a decision to test the validity, and corresponding output signals. As far as the addProduct signal is concerned, we just call an addProduct operation that has to be defined on the sort productRefStock: as usual, the exact semantics of operations are to be defined latter, in the sorts part of the specification, the dynamic conditions being in the process graph.

Figure 2.7 gives only what should be added to the definition given in Fig. 2.3.

SET_ORDERS.

Question 22: *What happens when cancelling an order?*

Answer: *There are two solutions: (1) mark it as being cancelled, or (2) remove it from the set of orders. The first solution would require modifications to the existing process graph (when invoicing, we should verify that the order is not cancelled). So we choose the second solution.*

As with STOCK, Fig. 2.8 gives only what should be added to the definition given in Fig. 2.4.

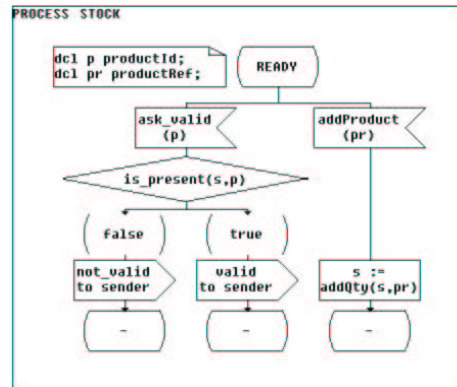


Fig. 2.7. The STOCK process (added) behaviour for case 2

2.3.3 Sort Definitions

The new process graphs introduce new operators. We herein give the parts that are to be added to the sorts defined in Sect. 2.2.3. Another solution would have been to use the SDL inheritance concept to define the new sorts, and to modify the process graphs to take this into account.

Sorts orderSet and productStock. As far as sort `productStock` is concerned, new operators `is_present: productStock, productId → Boolean` and `addQty: productStock, productRef → productStock` are needed. `is_present` is like the operator with the same name defined in `orderSet` (we may have defined a new set type constructor by specializing `Powerset`).

```

newtype productStock
inherits basicProductStock
  literals all;
  operators all;
adding
  operators
    ...
    is_present : productStock, productId -> Boolean;
    addQty : productStock, productRef -> productStock;
axioms
  for all s in productStock (
  for all p1,p2 in productRef (
  for all id in productId (
    ...
    is_present(empty, id) == false;
    (p1!id = id)==> is_present(incl(p1,s),id) == true;
    (p1!id /= id)==> is_present(incl(p1,s),id) == is_present(s,id);

    addQty(empty,p2) == incl(p2,empty);
  
```

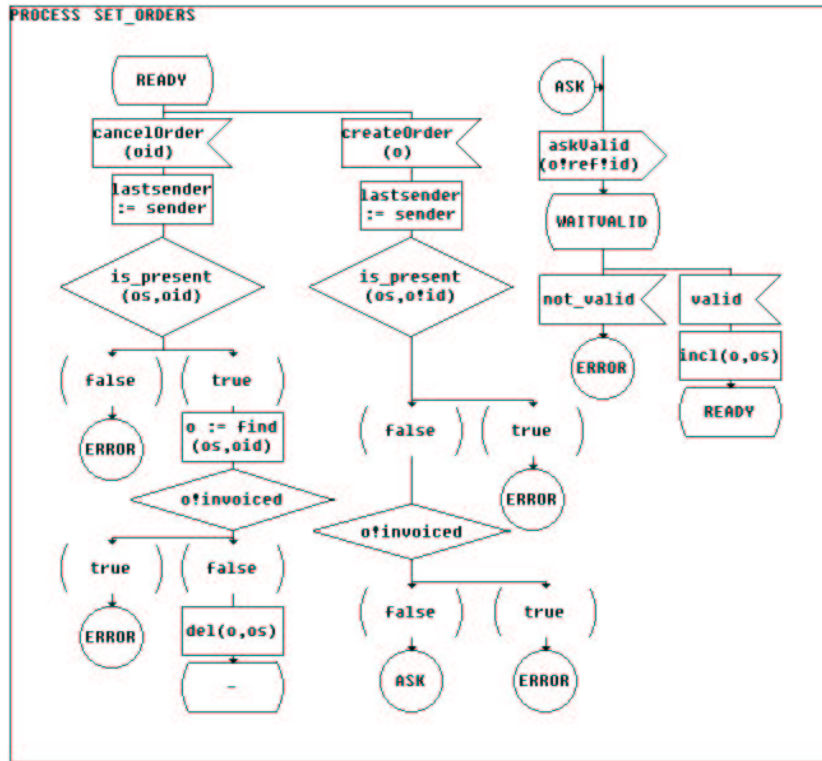


Fig. 2.8. The SET_ORDERS process (added) behaviour for case 2

```
(p!id = p2!id) ==> addQty(incl(p,s),p2) == incl(addQty(p,p2!qty),s);
(p!id /= p2!id) ==> addQty(incl(p,s),p2) == incl(p,addQty(s,p2));
...
);););
endnewtype;
```

Sort productRef. Due to prior sort definitions, an operator `addQty: productRef, Nat → productRef` is needed.

```
newtype productRef
operators
...
addQty : productRef, Natural -> productRef;
axioms
for all pr in productRef (
for all qtyAdd in Natural (
addQty(pr,qtyAdd) == qtyModify!(pr,qtyExtract!(pr) + qtyAdd);
));
endnewtype;
```

2.4 The Natural Language Description of the Specifications

2.4.1 Case 1

In case 1, the only functionality is *invoice*.

Orders are made up of an order identifier, a status (“pending” or “invoiced”) and a unique product reference. A *product reference* is made up of a product identifier and a non-negative quantity. A *stock* is a set of product references. The orders are in a *set of orders*.

To invoice means : change the status of the order from “pending” to “invoiced” and reduce the quantity of product in stock by an amount corresponding to the amount in the product reference of the order being invoiced. Invoiced orders are kept in the set of orders.

An order is said to be *invoiceable* if it is in “pending” status and if the product it references exists (in a quantity superior or equal to the one in the order) in stock.

The *invoice* functionality takes a unique order identifier as parameter. Its effect is to invoice the order if it is invoiceable. If it is not, a specific signal (*error*) is returned. Orders are invoiced in sequence.

2.4.2 Case 2

Case 2 is an extension of case 1. New functionalities are *addProduct*, *createOrder* and *cancelOrder*.

addProduct takes a product identifier and a certain non-negative quantity as parameters. If the product does not exist in stock, it is created (with the given quantity as initial amount), elsewhere, the quantity of the product in stock is increased by the given quantity.

createOrder takes an order as parameter. The product in the order must exist in stock or a specific signal (*error*) is returned. This signal is also returned if the order identifier exists in the set of orders or if the order is created in invoiced status.

cancelOrder takes an order identifier as parameter. If the order does not exist in the set of orders or if it has already been invoiced then a specific signal (*error*) is returned, elsewhere the order is removed from the set of orders.

2.5 Conclusion

As shown by the invoicing system, informal specifications, even of small case studies, are inherently incomplete and not precise. Being formal, SDL enables us to express the system requirements in a more precise way (raising questions) and to validate the specification using a wide range of tools (theorem provers for the data part, simulation, model-checking).

Like other “mixed” formal specification languages or methods [4,5], SDL enables one to describe both the dynamic and the static parts of systems. This is really a great advantage as the semantics of operations and the order and conditions under which they may be applied are equally important. Clearly separating the specification into structuring, process graphs and data types makes mixed specification easy.

SDL comes with both a textual and a graphical representation. This provides the specifier with a wider specification toolbox. The SDL graphical concepts are intuitive. They enable the specifier to work at different abstraction levels using the structuring mechanisms. They also make the extension of existing specifications easier (see the passage from case 1 to case 2 for example). The system structuring may be extended adding blocks and channels. The process graphs may be extended adding new transitions. Finally, data types may be extended using the SDL inheritance concept.

SDL asynchronous buffered communication semantics are closer to real-world communication mechanisms - closer to an implementation in terms of (asynchronously) communicating objects - than some other specification languages such as LOTOS. Moreover, requirements involving time may be expressed in SDL using timers.

If these SDL strengths were used in case 2, case 1 showed its main weakness. Unlike model-based specification methods such as Z [6] or B [1], SDL is inherently dynamic and requirements like “the set of orders and the stock are always given in an up-to-date state” (a state invariant) cannot be expressed.

Acknowledgements

I would like to thank Prof. K. J. Turner for his careful reading and comments about an earlier version of this chapter.

References

1. J.-R. Abrial. (1996) *The B Book - Assigning Programs to Meanings*. Cambridge University Press
2. CCITT. (1992) *Recommendation Z.100: Specification and Description Language SDL*, blue book, volume x.1 edition
3. J. Ellsberger, D. Hogrefe, and A. Sarma. (1997) *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall
4. ISO/IEC. (1989) ESTELLE: A Formal Description Technique based on an Extended State Transition Model. ISO/IEC 9074, International Organization for Standardization
5. ISO/IEC. (1989) LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization
6. D. Lightfoot. (1991) *Formal Specification using Z*. Macmillan

7. P. Poizat, C. Choppy, and J.-C. Royer. (1999) Concurrency and Data Types: A Specification Method. An Example with LOTOS. In J. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th International Workshop on Algebraic Development Techniques WADT'98*, volume 1589 of *Lecture Notes in Computer Science*, pages 276–291, Lisbon, Portugal. Springer-Verlag
8. K. J. Turner, editor. (1993) *Using Formal Description Techniques, An introduction to Estelle, Lotos and SDL*. Wiley