Hierarchical Specification of Reactive Systems: a case study

Yves Ledru

Past affiliation :
Unité d'Informatique, Université Catholoque de Louvain, Belgium

Current affiliation :
Université Joseph Fourier (Grenoble 1) - LIG
BP 72, 38402 St-Martin-d'Hères, France
Yves.Ledru@imag.fr

Abstract

This paper discusses the development of the specification of reactive systems. A development strategy is proposed which bases the specification of a reactive system on the description of its environment. This approach has numerous advantages: easier validation of the specification, availability of suitable modeling languages, and simulation of the environment during the testing phase of the product.
The specification of the environment is expressed in the STATECHARTS formalism. It is developed in a top-down manner. This top-down approach calls for the enhancement of the STATECHARTS by a transition refinement construct.
The development of an actual case study, a data transfer problem, demonstrates the methodological approach and shows the usefulness of this extension of the formalism. This experiment also results in further development guidelines.

Y. Ledru. Hierarchical Specification of Reactive Systems : a case study. In Proceedings of the CompEuro'90 Conference, pages 109–116. IEEE Computer Society Press, Tel-Aviv, 1990.

Proceedings of COMPEURO'90"1990 IEEE International Conference on Computer Systems and Software Engineering" Tel-Aviv, Israel, May 8-10, 1990. IEEE C.S. Press

Hierarchical Specification of Reactive Systems: a case study

*Yves Ledru**

Unité d'Informatique
Université Catholique de Louvain
Place Sainte-Barbe, 2
B-1348 Louvain-La-Neuve (Belgium)

Abstract

This paper discusses the development of the specification of reactive systems. A development strategy is proposed which bases the specification of a reactive system on the description of its environment. This approach has numerous advantages: easier validation of the specification, availability of suitable modeling languages, and simulation of the environment during the testing phase of the product.

The specification of the environment is expressed in the STATECHARTS formalism. It is developed in a top-down manner. This top-down approach calls for the enhancement of the STATECHARTS by a transition refinement construct.

The development of an actual case study, a data transfer problem, demonstrates the methodological approach and shows the usefulness of this extension of the formalism. This experiment also results in further development guidelines.

1 Introduction

Many industrial systems are identified as "reactive". These terms apply to systems whose primary role is to maintain some interaction with their environment [8]. Both industrial and academic communities carry on a lot of research in this area of software engineering. These challenging researches cover the design of:

- new specification and implementation languages which take into account the specific nature of reactive applications, i.e. the interactions between reactive entities;
- development methods and formal frameworks which help to establish a link between specification and implementation;
- tools which support this development process.

The goal of this paper is to provide some hints on the development of the specification of reactive systems. Many development approaches (e.g. [11]) aim at a direct specification of the reactive system under development. This description of an abstract object is a non-trivial process and often results in premature design choices. Another approach, where the environment of the system under development plays a central role, is proposed here. The specification of the environment is a modeling process which appears to be easier to lead and then validate.

The Statecharts provide an adequate formalism for this modeling process. This formalism is enhanced here by a natural extension, the OR-decomposition of **transitions**, which is dual to the OR-decomposition of states. This construct is mandatory in a top-down modeling process where states and transitions are described hierarchically.

This paper illustrates the approach by the development of the specification of an actual case study: a data transfer between a computer and an musical synthesizer. Section 2 presents the case study informally. Section 3 details the approach followed during its development and proposes an extension to the STATECHARTS specification language. Section 4 describes the specification process. Section 5 discusses the problems left open in the case study.

2 Presentation of the case study

Figure 1 outlines the data transfer problem. A program must be designed for the computer to achieve the transfer of data stored in the musical synthesizer. A data transfer protocol is imposed by the synthesizer manufacturer and implemented in the synthesizer.

The transfer begins with a *request* packet from the computer to the synthesizer. The synthesizer replies with a *want to send* packet which must be acknowledged by the computer. The synthesizer then sends several data packets and an *end of transmission* packet. All packets must be acknowledged by the computer. If anything goes wrong, a *reject* packet may be issued by each system. This packet terminates the transmission. An *error* packet may also be sent. It must then be acknowledged by a *reject* packet.

This application features thus two systems: the computer and the synthesizer. Both entities are to be considered as reactive, i.e. the problem is only concerned with their external behaviours.

* Leibniz Project (This research has been funded in part by the Belgian SPPS under grant RFO/AI/15).
This research was undertaken within the ESPRIT Project 510 Tool'Use.

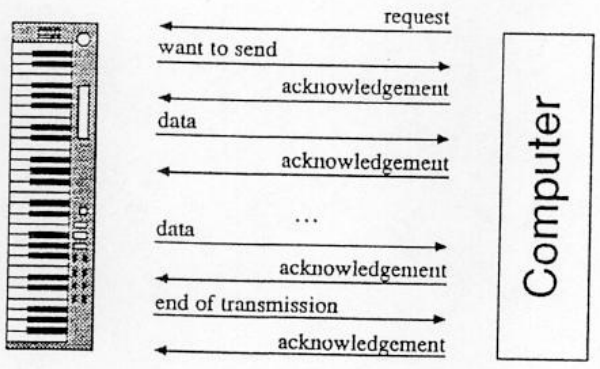

Figure 1: Outline of the data transfer

3 The development strategy

3.1 Specification of the environment

The development of a reactive system is made up of several phases, one of which being the precise specification of the problem.

In [14], A. Pnueli distinguishes between the notions of *requirement specification* and *system specification*. The requirement specification describes *what* is to be done while the system specification explains *how* it is performed, i.e. it is a high-level description of the implementation.

In any reactive problem, three viewpoints may be considered:

- S : the viewpoint of the reactive system under development;
- E : the viewpoint of the environment of S, i.e. the reactive entity which will interact with S;
- $S+E$: the global viewpoint which considers both reactive entities (S and E) from an external point of view.

In the data transfer problem, S corresponds to the computer, E groups the synthesizer and the wires which link it to the computer.

For each viewpoint, a specification may be set up.

1. The behaviour of S must still be designed; its specification is thus a requirement specification.
2. In most cases, E already exists at the specification stage; its behaviour is defined by its existing implementation. Its specification is thus a system specification and results from a modeling process.
3. The global behaviour of $S+E$ is an abstract object which sets several constraints on the behaviour of the reactive entities it rules. Once again, it should be described in terms of a requirement specification. In the data transfer case study, $S+E$ corresponds to the transmission protocol which rules the data transfer.

The specification phase must end up with a specification of the intended behaviour of S. By essence, a reactive system maintains strong interactions with its environment. There is thus a high degree of redundancy between these three specifications. This redundancy makes it possible to design S from only one of these three specifications.

The specification of the behaviour of S (1) corresponds to the abstract description of a family of implementations which fulfill the requirements. It corresponds to a typical requirement specification and care must be taken not to restrict the space of possible solutions by premature design choices which would turn it into a system specification. In [11], L. Lamport proposes a method for the development of the requirement specification of S. In this method, E is only seen through its interface with S.

Specifying of the behaviour of E (2), i.e. the direct environment of S, is a modeling process where the behaviour of an actual object is being described. As such, this specification does not really specify the behaviour of S but states how E will react to events issued by S. If we combine this specification with a goal to achieve (G), we get a requirement specification of S:

In the context of E, G must be achieved.

To build the specification of the actual environment of the system under development appears to be easier than the construction of the specification of the system itself. Indeed, in the best case, this specification may already be available (e.g. provided with the environment). In most cases, it only describes an existing behaviour and may be validated against it, using simulation techniques. If the environment does not yet exist, the availability of some specification of its behaviour is a prerequisite to the development of the specification of S. Therefore, it was decided to take this specification of E as the key element of the specification of the system under development. In the data transfer case study, the specification of the computer behaviour is thus:

> *In the context of the system specification of the synthesizer, design one of the possible computer behaviours which will interact with it and lead it to send an "end of transmission".*

Indeed the receipt of *end of transmission* is only issued after a successful execution of the data transfer. The development of the requirement specification is thus reduced to the one of a system specification.

Another specification approach is to build the system from the protocol specification (3). This approach is justified if the protocol is clearly defined and if its specification is available. Otherwise, the specification must be set up and validated from an informal description of the protocol, which is more difficult and error prone than the validation of E.

3.2 The specification of unexpected behaviours

The requirement specification of a reactive system must take into account two fundamental behaviours of its environment. The first behaviour is the one of a cooperative environment. This should be viewed as the normal case, where everything goes right. When the environment behaves normally, the reactive system (S) must achieve its goal (G), e.g. the computer must perform the data transfer in this case study. The second possible behaviour is the one where something goes wrong! In this context, the reactive system (S) might not be able to achieve its goal but it must ensure a minimal service which in many cases enforces a safety property. In the data transfer case study, this corresponds to the fact that the synthesizer may send *err*, *rjct*, or *garbled* messages due to internal choices or due to the possible corruption of messages sent to or received from the synthesizer.

The specification of a reactive system is thus twofold:

1. if the environment conforms to a given "normal" behaviour, the system must fulfill a maximal specification and perform useful activities;
2. in any case, it must satisfy a minimal specification.

During the design phase, efforts may be done in order to lower the requirements on the environment behaviour in (1) and to extend the coverage of the minimal specification in (2). In fault-tolerant applications, intermediate situations may be defined between (1) and (2) to allow the "graceful degradation" of the service performed by the reactive system.

In the data transfer case study, the second specification states that:

> *If the behaviour of the environment does not conform to the "normal" behaviour, the system should send a "rjct" message to its environment.*

Indeed, the *rjct* message is one way to close the data transfer and reset the synthesizer into its *Idle* state.

3.3 Extension of the STATECHARTS

3.3.1 Specification languages for reactive systems

Many specification and implementation languages have been designed for reactive systems.

- Some result from a logical or declarative approach: [14] reviews temporal logics and proposes a classification in terms of their scope within the life-cycle.
- Other relate to algebraic frameworks (process calculi like CCS [13] or ACP [1], the protocol specification language LOTOS [15]).
- A more structural approach may also be followed (ESTEREL [2], Estelle [5], finite state automata, Petri nets, LUSTRE [3], STATECHARTS, ...).

The scope of these languages within the software life-cycle is rarely made explicit by the authors. This is essentially due to the fact that most languages may be used at several abstraction levels.

The STATECHARTS formalism [6] has been introduced by D. Harel and A. Pnueli [8]. It is aimed at the description of complex reactive systems. Its development is due to both industrial and academic efforts [9] [10]. Being based on finite state automata which make it an imperative specification language, the STATECHARTS formalism is particularly *well-suited for system specification* and for the modeling of existing systems. Another asset of the STATECHARTS is their graphical appearance which has led them to industrial acceptance, particularly when combined with a suitable support environment. Finally, many academic works aim to define a formal semantics for the language [10] [9] [4]. The existence of such a formal semantics is a necessary condition to allow a thorough integration of the language in a development process. All these reasons have led to adopt the STATECHARTS as specification language for the data transfer problem.

3.3.2 Decomposition of transitions

The STATECHARTS formalism is based on finite state automata augmented by two constructs: the OR- and AND-decompositions of states. A dual concept to the OR-decomposition of states is the OR-decomposition of transitions. Figure 2 illustrates this new concept. All arrow paths which leave state *A* end in state *D*. These may be grouped into a global transition (*arrow*) which links these states.

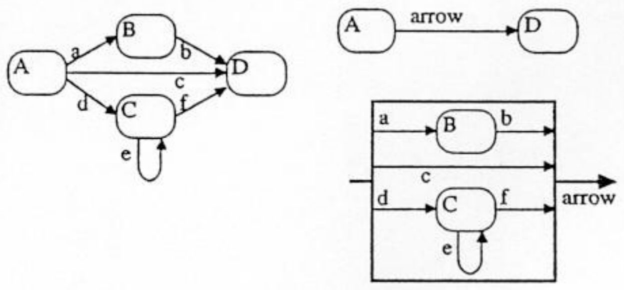

Figure 2: OR-decomposition of transitions

The diagram describing a transition refinement corresponds to sequences of sub-transitions. The graphical notation associated to this construct is made up of a rectangle followed by an arrow with the name of the abstract transition. The rectangle stores arrow paths starting at its left hand side and ending at its right hand side. From a graphical point of view, just as decomposed states look like other states, nothing distinguishes a refined transition (e.g. *arrow*) from an elementary transition (e.g. *a*). This changes the semantics of STATECHARTS transitions which are no more elementary events but may now denote substantial and non-instantaneous behaviours.

This natural extension to the STATECHARTS formalism reveals to be a useful construct. The additional structuring facility it provides is not only well-suited to describe macroscopical transitions, but it is also very useful to only detail once a transition which appears at several places on the same diagram. In this particular case, unnamed states are used to prevent naming conflicts (see figure 6). The case study described in section 4 also shows that this construct is mandatory in a top-down development process.

Although based on the same decomposition principle as state refinements, transition refinements differ from these in that:

- state refinements have a single default entry state while transition refinements may feature several entry transitions (e.g. *a*, *c* and *d* for *arrow*);
- state refinements do not explicitly define exit states; in transition refinements, exit transitions are precisely specified;
- the activities of a refined state may be interrupted by a transition occurring at the upper state; in the case of re-

fined transition, this may only happen if the refined transition is itself included in a refined state which is interrupted.

3.4 Summary of the approach

The principles of the development approach may thus be stated as follows:

- The specification of the reactive system is provided in terms of the behaviour of its environment combined with a goal to achieve.
- The specification of a reactive system is twofold: a normal case describes the goal to achieve if the environment behaves normally, and a safety behaviour is specified for all other behaviours of the environment.
- The hierarchical structure of the STATECHARTS diagrams leads to combine their use with a top-down approach.

4 Top-down specification

The guidelines of section 3.4 are now applied to describe the system specification of the synthesizer.

4.1 Top-level specification

The top-level view of the behaviour of the synthesizer distinguishes two states (figure 3):

- a state where the synthesizer is idle, i.e. does not interact with its environment (*Idle*);
- a state where the synthesizer performs the data transfer (*Xfer*).

Idle is the initial state (denoted by a special arrow). In this first state, the system ignores any incoming message except the request messages (*rqst*). Once it has received such a message, it enters the *Xfer* state and performs the transmission. It will only leave this state when it has received an acknowledgement and the variable *EOT* is set to true. It may also leave this state if it receives a message which corresponds to a transmission flaw (*defect*). The convention taken here expresses input events as labels on the transitions and output events as *send* commands (see figure 4).

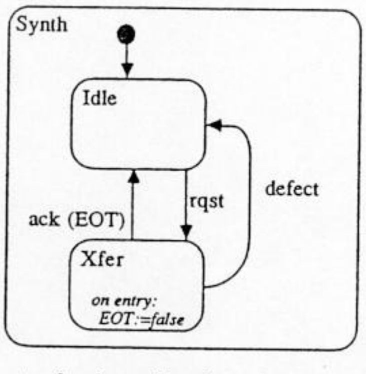

Figure 3: The top-level specification of the environment

The precedence rules of the STATECHARTS make it mandatory to use an auxiliary boolean variable in combination with the *ack* transition. Otherwise, the system should leave the *Xfer* state at the reception of the first *ack*. The *EOT* variable provides an abstracted piece of information on the content of the *Xfer* state. It is initially set to false on entry in the *Xfer* state and denotes that the system should eventually enter an internal state which will turn the variable to true and enable to leave the *Xfer* state on reception of an *ack* message.

Inter-level transitions

The introduction of the auxiliary variable *EOT* prevents the use of inter-level transitions, i.e. transitions which cross the border of a rectangle representing a state. These transitions link states located at different decomposition levels.

These inter-level transitions not only lower the clarity of STATECHARTS specifications, they are also incompatible with a sound top-down development process. Indeed, they force the simultaneous development of several decomposition levels.

The problem of inter-level transitions appears as one of the major rationale for the design of the Argos language [12]. This case study presents an alternate solution based solely on the STATECHARTS primitives.

4.2 State refinement

The STATECHARTS allow the hierarchical decomposition of a state into sub-states. This feature is used here to refine *Xfer* (figure 4).

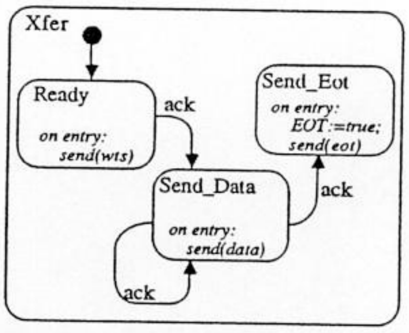

Figure 4: The decomposition of *Xfer*

The transfer activity proceeds in three phases:

- initialization;
- actual transfer of data;
- termination.

At any moment of the transfer, defects may arise (loss of data, reception of garbled messages, ...). These have already been abstracted by the *defect* transition at the previous decomposition level and are no more relevant at this level.

Xfer is thus decomposed into 3 sub-states corresponding to the three phases of the transmission:

- *Ready* is the default state where the transfer is started;

- *Send_Data* corresponds to the transmission of the data;
- *Send_Eot* ends the transmission.

The structure of the upper-level state (*Xfer*) has put a constraint on its sub-states: the existence of an internal state where the EOT variable will be verified and will enable the upper-level to react to *ack* messages. The variable is set to *false* at the entry point of the *Xfer* state and verified when the system is in the last sub-state (*Send_Eot*).

When the system enters *Send_Data*, it issues *data* packets. Since the number of data messages is not specified, non-determinism is used to express that *ack* messages may lead either to *Send_Data* or *Send_Eot*. As such, the specification does not state that an *end of transmission* (*eot*) is eventually issued. An additional fairness constraint on the selection of the *ack* transitions should thus complement the specification. This constraint may be avoided by adding an implicit fairness assumption on the choice of any transition of the diagrams.

4.3 Transition refinement

The current status of the specification does not need further state refinements. Nevertheless, the specification is not sufficiently detailed. For example, the *defect* transition corresponds to several behaviours depending on the kind of transmission error encountered. Obviously, the precise description of this transition leads to the definition of new states and transitions. If we stick to the state decomposition constructs of the STATECHARTS, the introduction of this supplementary information results in the modification of the diagrams. Some intermediate states and transitions will be added, other ones will be deleted. Such modifications do not fit in a top-down development approach.

Another solution is to describe the transitions in another formalism. This seems to be the solution supported by the STATEMATE* [7] environment.

The development strategy followed here leaves the diagrams unchanged. Instead of multiplying the formalisms, the transition decomposition construct is used to integrate the supplementary information as separate diagrams in the same description framework.

Figure 5 uses this new construct to describe the *defect* transition. There are several kinds of "defects":

- the synthesizer receives a *rjct* packet which closes the transmission;
- the synthesizer receives an error (*err*) message and issues a *rjct* packet which closes the transmission;
- the synthesizer receives a garbled message, i.e. an unexpected or corrupted message; this case is not well defined in the informal documentation of the synthesizer and will be discussed later; at this level of the specification, we will assume that the synthesizer detects this message and reacts to it with a *rjct* packet.

The *defect* transition is thus refined into two parallel constructs. The first one is a direct transition corresponding to the reception of a reject (*rjct*) message. The second one involves an

* STATEMATE is a registrated trademark of i-Logix, Inc.

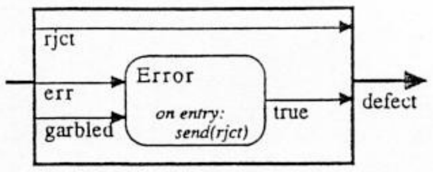

Figure 5: The decomposition of *defect*

intermediate state (*Error*) where a *rjct* message is sent to the computer after the reception of an error message (*err*) or a *garbled* message. The *Error* state is instantaneously left after the emission of the *rjct* message, due to the *true* transition which is always satisfied.

The top-down decomposition proceeds with the decomposition of the elementary messages. Figure 6 gives the decomposition of *rqst*. *ack*, *err*, and *rjct* are described in a similar way. *rqst* is made up of a header (*hdr*), followed by a byte which denotes the type of the message (*rqst_code*), and a *tail*. The intermediate states are not named in this subdiagram.

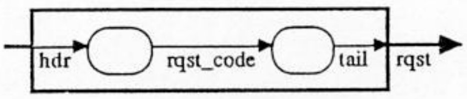

Figure 6: The decomposition of *rqst*

Finally, figure 7 details *hdr* as a sequence of 2 bytes. *tail* may also be refined in a similar diagram.

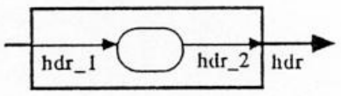

Figure 7: The decomposition of *hdr*

4.4 Description of the messages

As shown in figure 5, some messages may be both sent and received by the synthesizer (e.g. *rjct*). The existence of different techniques for the expression of input activities (transitions) and output activities (*send* commands) may lead to inconsistencies between the descriptions. Indeed, the format of messages such as *rjct* is the same in both input and output activities. Therefore, both inputs and output messages are described here in the framework of transition refinements.

When transition refinements are reduced to sequences of subtransitions, a notation shortcut, based on regular expressions, is used which provides a textual representation of the refinement. For example, figure 6 is now replaced by:

$$rqst \triangleq hdr; rqst\_code; tail$$

Figure 8 completes the specification with the description of the elementary messages. In this figure, a *garbled* message is defined as the complement of a set of accepted messages.

$$
\begin{array}{|ll|}
\hline
\mathit{rqst\_code}, \mathit{wts\_code}, \mathit{ack\_code}, \mathit{eot\_code}, & \\
\mathit{err\_code}, \mathit{rjct\_code}, \mathit{dat\_code}, & \\
\mathit{hdr\_1}, \mathit{hdr\_2}, \mathit{tail\_1}, \mathit{tail\_2}, \mathit{tail\_3} & : \ \mathit{constants} \\
\mathit{byte\_1}, \ldots, \mathit{byte\_256} & : \ 0 \ldots 127 \\
\hline
\end{array}
$$

$$
\begin{aligned}
\mathit{rqst} &\triangleq \mathit{hdr}; \mathit{rqst\_code}; \mathit{tail} \\
\mathit{wts} &\triangleq \mathit{hdr}; \mathit{wts\_code}; \mathit{tail} \\
\mathit{ack} &\triangleq \mathit{hdr}; \mathit{ack\_code}; \mathit{tail} \\
\mathit{eot} &\triangleq \mathit{hdr}; \mathit{eot\_code}; \mathit{tail} \\
\mathit{err} &\triangleq \mathit{hdr}; \mathit{err\_code}; \mathit{tail} \\
\mathit{rjct} &\triangleq \mathit{hdr}; \mathit{rjct\_code}; \mathit{tail} \\
\mathit{hdr} &\triangleq \mathit{hdr\_1}; \mathit{hdr\_2} \\
\mathit{tail} &\triangleq \mathit{tail\_1}; \mathit{tail\_2}; \mathit{tail\_3} \\
\mathit{data} &\triangleq \mathit{hdr}; \mathit{dat\_code}; \mathit{tail\_1}; \mathit{tail\_2}; \mathit{data\_bytes}; \mathit{tail\_3} \\
\mathit{data\_bytes} &\triangleq \mathit{byte\_1}; \mathit{byte\_2}; \ldots; \mathit{byte\_256}; \mathit{check\_sum} \\
\mathit{check\_sum} &\triangleq 128 - ((\textstyle\sum_{i=1}^{256} \mathit{byte\_i}) \mathit{mod} 128) \\
\mathit{garbled} &\triangleq \mathit{complement\_of}(\{\mathit{ack}, \mathit{rjct}, \mathit{err}\})
\end{aligned}
$$

Figure 8: Description of the elementary messages

4.5 Towards a formal requirement specification

This section has presented the system specification of the environment of the computer. Section 3.1 has explained how this specification must be associated to a goal in order to specify the behaviour of the computer. The goal was also stated: to lead the synthesizer to send an *end of transmission*.

When this specification is used at later stages of the development of the reactive system, efforts will be spent to relate the implementation of the system to this specification. It is thus useful to state this requirement specification more formally. This should then look like:

$$\mathcal{B}_{Synth} \wedge rqst \wedge at(Idle) \Rightarrow \Diamond(ack \wedge EOT)$$

which means that once a *rqst* has been sent in the *Idle* state, it will eventually be followed by an *ack* with the *EOT* variable true. $\mathcal{B}_{Synth}$ is a notation which expresses that the synthesizer behaviour conforms to the STATECHARTS specification of *Synth*, i.e. the top-level state of the specification. Unfortunately, the formal framework which links this temporal logic formula to the STATECHARTS is still to define.

The formal expression of the fairness constraint on the choice of *ack* transitions (section 4.2) may be formalised as:

$$at(Send\_Data) \wedge (\Box\Diamond ack) \Rightarrow \Diamond at(Send\_Eot)$$

Finally, the minimal requirement on the behaviour of the computer is:

$$\neg\mathcal{B}_{Synth} \wedge rqst \wedge at(Idle) \Rightarrow \Diamond rjct$$

which states that a *rjct* must be eventually issued if the synthesizer behaves improperly.

In fact, this specification is not implementable! It is dependent on the fulfillment of a fairness constraint by the environment. Indeed, if the environment is in the *Send_Data* state, it is impossible to decide from its past behaviour whether or not it will eventually reach the *Send_Eot* state and conform to the "normal" behaviour.

Therefore, the specification should be modified in such a way that conformance of the behaviour of the actual environment to the "normal" behaviour may always be decided within a finite time. In this case study, this may be achieved by the definition of an upper bound on the number of *ack* messages received in the *Send_Data* state.

This results in a methodological guideline:

- the specification of a "normal" behaviour should avoid the use of "eventualities" to allow a clear and easily stated distinction between normal and unexpected behaviours.

5 Open problems

The system specification presented in section 4 specifies most of the behaviour of the synthesizer and consequently the behaviour of the system under development. Nevertheless, additional information should be associated to the specification:

- the behaviour of the synthesizer when confronted to unexpected inputs (5.1);
- the real-time characteristics of the synthesizer behaviour (e.g. response time after reception of a message) (5.2);
- the target system description, i.e. the characteristics of the hardware of the computer on which the program must be implemented (5.3).

5.1 Reaction to unexpected inputs

The system specification of the synthesizer does not necessarily define the complete synthesizer behaviour when confronted to unexpected inputs. For example, how does it react to an *ack* message in the *Idle* state? In other words, should the specification exhaustively cover the behaviour of the environment?

A first answer to that question is related to the availability of complete documentation on the synthesizer behaviour. Without such information, exhaustive tests of the actual synthesizer behaviour are needed to complete the specification. Such a costly activity must be motivated. In fact, the answer to the question is that a specification should not be overloaded with useless information. In the context of the requirement specification of the computer program, the specification described in section 4 provides enough information to build and prove the computer program.

A related problem is linked to the formal semantics of the transitions. In section 4.1, the semantics of figure 3 was interpreted as: in the *Idle* state, the synthesizer ignores any incoming input except *rqst*. If we stick to this interpretation, then the semantics of the specification is completely determined. Another interpretation, which is probably better-suited to the notion of specification, is to allow to leave some aspects undefined in the specification.

This discussion is particularly significant for the transition refinement construct: to ignore an incoming message, it is necessary to recognise it as "unexpected". If an incoming message begins with an acceptable sub-message and then differs from the expected messages (e.g. *rqst* and *ack* both start with *hdr*), shall the synthesizer ignore the incoming sub-message and be left in an intermediate sub-state of the refined transition or backtrack to its initial state? From this point of view, the second interpretation appears as more appropriate: the system is left in a state such that its subsequent behaviour is not specified.

5.2 Real-time characteristics

Another unspecified aspect of the environment is its real-time characteristics. In this case study, real-time constraints exists on the rate of output messages. For example, the 263 bytes of *data* messages are emitted as one block and an adequate buffer must be prepared at the receiver side. Unfortunately, the current state of the art in the specification of real-time characteristics does not allow yet a thorough formal coverage (specification and proofs) of this aspect of the development of reactive systems. This does not prevent the developer from describing informally the real-time characteristics and the related problems, e.g. the STATECHARTS allow to express some timing information.

Moreover, the specification of the minimal (safe) behaviour must take into account these potential problems. Here, the design of the reactive system must ensure that eventually (after some time-out) the *rjct* message will be issued. This requirement reduces the set of acceptable behaviours of the environment by putting some time constraints on the interactions. Indeed, typical misbehaviours include now:

- cases where the environment reacts too slowly (after the time-out) to its inputs;
- cases where the environment reacts so quickly that its response is missed by the computer.

There are thus minimal and maximal requirements on the timing of the transitions in the environment behaviour.

5.3 Target system description

A complete requirement specification must also describe the relevant characteristics of the target system, i.e. the ones that must be taken into account to prove the correctness of the solution. These may include:

- the semantics of the programming language in which the reactive program will be implemented;
- the specification of the external functions which drive the input/output devices of the computer and will be called by the reactive program to interact with its environment;
- the hardware characteristics of the computer (e.g. memory space, real-time figures,...);
- ...

In fact this information describes the direct environment of the reactive program which will rule the behaviour of the target computer. This information is thus absolutely necessary to establish the link between the program and the environment described in section 4.

5.4 Additional guidelines

These open problems result thus into additional guidelines:

- the specification of the environment must only describe its relevant features to avoid overloading; ambiguities may be left in this specification, provided enough information is available to develop the system;
- due to the poor level of formality in the treatment of real-time aspects, the specification should avoid as much as possible the use of timing constraints;
- the requirement specification should include the description of the target system.

6 Conclusions

This paper has shown how a requirement specification may be derived from the system specification of its environment. This approach has numerous advantages:

- in the area of reactive systems, many languages (Petri nets, STATECHARTS, ESTEREL, ...) are better suited to system specification than to requirement specification; their application to the description of the system under development often results in premature design choices;
- the environment is often an existing object; descriptions of the object may already be available and supplementary information may be extracted from tests;
- if the specification is executable, it may be compared to the actual behaviour of the object and may be used to simulate the environment of the program at a testing stage;
- the requirement specification is preserved if the environment is re-implemented afterwards but still conforms to its system specification.

The case study also experimented the adequacy of the STATECHARTS to support the top-down development of a system specification. It has shown that transition refinements provide a natural extension to the formalism which is necessary in a top-down development approach. Other extensions to the STATECHARTS have only been sketched and may be investigated further:

- the use of temporal logic formulae;
- the use of regular expressions to describe the transition refinements; only of subset of these was used here as a notation shortcut;
- the AND-decomposition of transitions.

The introduction of these extensions in the formalism must also be motivated by methodological arguments.

This case study was conducted with the help of very simple tools (text editor and graphical editor). When large scale applications are under development, dedicated tools, such as STATEMATE [7], are needed to manage the large amount of diagrams involved. The decomposition of transitions, which is very similar to the OR-decomposition of states, should be easy to integrate in such environments.

A crucial aspect in the evolution of software engineering techniques for reactive systems is the definition of development methods. The methodological guidelines stated in section 3.4 have been demonstrated on the data transfer problem. Sections 4.5 and 5.4 have provided further guidelines. New ones should also be derived from:

- feed-back from the next activities of the development (design, implementation, integration, ...);
- the use of tools, especially in the validation of the specification using simulation and animation techniques;
- the development of large-scale industrial specifications.

In any case, we believe that efforts spent to record, analyse, understand, and model these development activities are the driving force which will eventually give rise to sound development methods.

Acknowledgements

Thanks to Pierre Collette, Daniel Dzierzgowski, Aubin Fanard, Marc Lobelle, Pierre-Yves Schobbens, and Michel Sintzoff for their contributions to the final version of this paper. Other thanks are due to the anonymous referees of the COMPEURO conference for their comments.

References

[1] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *TCS*, 37(1), 1985.

[2] G. Berry and I. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency (Lecture Notes in Computer Science 197)*, pages 389–449, Springer Verlag, 1985.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th POPL*, pages 178–188, ACM, 1987.

[4] P. Collette and B. Villar. *Intégration de formalismes déclaratifs et impératifs pour la spécification de systèmes réactifs.* Technical Report MEM 89 01, Université Catholique de Louvain, Unité d'Informatique, 1989.

[5] M. Diaz, J.-P. Ansart, J.-P. Courtiat, P. Azema, and V. Chari. *The formal description technique Estelle - Results of the ESPRIT/SEDOS project.* North Holland, 1989.

[6] D. Harel. STATECHARTS: a visual formalism for complex systems. *Science of Computer Programming*, 8(3), 1987.

[7] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. In *Proceedings of 10th International Conference on Software Engineering (ICSE 10)*, pages 396–406, IEEE Computer Society Press, 1988.

[8] D. Harel and A. Pnueli. *On the development of reactive systems.* Technical Report CS85-02, The Weizman Institute of Science, Rehovot, Israel, 1985.

[9] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. In *Symposium on Logic in Computer Science*, pages 54–64, IEEE Computer Society Press, 1987.

[10] C. Huizing, R. Gerth, and W. P. de Roever. Modeling Statecharts Behaviour in a Fully Abstract Way. In *Proceedings of CAAP'88, 13th Colloquium on Trees in Algebra and Programming (LNCS 299)*, pages 271–294, Springer Verlag, 1988.

[11] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, 1989.

[12] F. Maraninchi. Argonaute: Graphical Description, Semantics and Verification of Reactive Systems by Using a Process Algebra. In *Workshop on Automatic Verification of Finite State Systems (Grenoble, june 1989)*, Springer Verlag, to appear in 1990.

[13] R. Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer Verlag, 1980.

[14] A. Pnueli. Specification and development of reactive systems. In H.-J. Kugler, editor, *IFIP 86*, pages 845–858, North-Holland, 1986.

[15] P.H.J. van Eijk, C.A. Vissers, and M. Diaz. *The formal description technique LOTOS - Results of the ESPRIT/SEDOS project.* North Holland, 1989.