Bridging the Gap Between Formal and Semi-Formal Methods for Discrete Control : A case study with VDM and Statecharts

Abd-El-Kader Sahraoui (1), Issa Traoré (2)

(1) Laboratoire d'Analyse et d'Architecture des Systèmes du C.N.R.S
7, Avenue du Colonel Roche, 31077 Toulouse, France e-mail: sahraoui@laas.fr
and I.U.T. B, Université Toulouse II 31703 Blagnac
(2) Institute for Informatics, University of Oslo, Norway E-mail: issa@hrp.no

Draft Paper

June 11, 1999

Abstract: In spite of the growing importance of formal methods, they have yet to achieve wide industrial acceptance for several reasons, among which the fact that most of the works about formal specifications have dealt mostly with the notations, not with the elicitation process. This paper is a contribution to the adaptation of formal methods for industry with an emphasis on discrete embedded control systems. A framework is proposed for the integration of semi-formal and formal notations in order to produce a formal specification of systems. We used two models, a discrete event model as statecharts and a model oriented method as VDM for the continuous (discrete time) apects of the system. The approach relies on two steps: the first step consists in using adequately Statemate to guide the analyst's understanding of the system and produce a preliminary document. The second step consists in generating a VDM specification from the preliminary document on the basis of predefined rules. A tool support is proposed to assist (possibly) the second step. The notion of "Control kernel" is introduced, as a means to take into account explicitly the control information in the final VDM specification.

Keywords: formal methods, semi-formal methods,discrete event, discrete time, VDM, Statecharts.

1 Introduction

One of the most challenging issue nowadays in software community concerns the developpement of systems providing a certain level of quality at reasonable cost and time delay. This fact is emphasized by the fact that many important application areas such as aeronautics, nuclear energy, telecommunication or medical applications require a high level of reliability and safety.

For instance, in the aeronautics industry, the current trend is to develop avionics that provide more functionality and high performances. Consequently, they are facing the high cost of making avionics. Nowadays, the whole avionics represents roughly 10% of the cost of an aircraft [Sahraoui *et al.*,1996], [Traoré *et al.*,1998]. This is why it is important to reduce it at each development step, from the avionics system specification to the test of computers.

To do so, it is necessary to increase both verification and validation of a system specification, and the amount of automatically produced software, and consequently to reduce testing. This can only be envisaged by the use of formal languages which provide a semantics to which is attached a mathematical theory allowing only one single possible interpretation [Arago,1997].

In fact, the use of formal methods in software development improves the insight into and understanding of requirements, help clarify the customer's requirements by highliting or avoiding contradictions and ambiguities in the specifications, enables rigorous verification of specifications and their software implementations, and easier the passage from specification and design to implementation [Fraser *et al.*,1994], [Wing,1990].

In spite of these benefits, the utilisation of formal methods in indutry was relatively restricted, due to a certain number of reasons such as the high memory and processing time required, the esoterism and the lack of friendliness of the formalisms, the insufficient broadcasting of knowledge and tools etc [Bowen,1995].

Given these difficulties, a number of strategies have been proposed for incorporating judiciously formal methods into the developpement process [Andrews,1988], [Kemmerer,1990], [Babin *et al.*,1991], [Miriayala,1991]. With the proliferation of such strategies, Fraser *et al.* have identified in [Fraser *et al.*,1994] throughout a classification, their commonalities, their differences and their applicability to different context. This classification relies mainly on two aspects:

1. the *formalization process* which consists either in a *direct process* or in a *transitional process*. In the *transitional* strategies, semi-formal notations are used as intermediate steps in the production of a formal specification, while *direct strategies* are characterized by the absence of any intermediate use of semi-formal notations.

2. the formalization support which concerns the computer support that a strategy use for producing formal specifications.

¿From this principles, they identified four generic strategies: direct unassisted, direct computer-assisted, transitional unassisted and transitional computer-unassisted. It appeared on the first hand that the transitional strategies are best suited for large or ill-structured systems than the direct strategies, thanks to the structuring and elicitation features of semi-formal methods.

On the other hand, the computer-assisted strategies are, naturally, superior to the unassisted ones, which requires an important manpower and could lead to many errors. So this kind of strategies are the best-suited for the penetration of formal methods into the industry.

The purpose of this paper is to propose a computer-assisted strategy taking into account the limitations of the existing strategies.

We propose an approach combining the features of semi-formal methods (friendliness, communicability etc.) and formal methods (rigor, precision etc.) We use statemate notations (especially statecharts and activity-charts) [Harel,1996] and the Vienna Development method(VDM) [Jones,1990], [Fitzgerald,1997], as surrogates for the informal and formal specification languages, respectively.

The proposed approach, consists in giving structural guidance for the construction of a formal specification of the system which should then be used as a starting basis for the rest of the developement process.

The paper is structured as follows. Section II justifies our strategy throughout an overview of existing strategies and highlights the reasons underlying the choice of the used methods. Several case studies were developped with the proposed approach; we introduce in this section an avionics case study proposed by AEROSPATIALE Aircraft, in order to illustrate the key aspects of the approach. Section III outline our approach for integrating statemate and VDM. Section IV deals with the control aspects, the main weakness of the existing strategies; the process of translation from statecharts to VDM is detailed througout the introduction of the notion of *Control Kernel*. Section V reports upon the automation of the translation process. Finally section VI discusses to what extent the work presented in this paper is of significance to software development community.

2 Context of the Work

2.1 State of the Art

Most of the reported examples of strategies integrating formal and semi-formal specification techniques, could be classified in two kinds.

The philosophy behind the first kind of strategy consists in developing first a specification of the system using a structured method, and then deriving a formal specification on the basis of a set of translation rules. The semi-formal method will just serves in this case in giving structural guidance for the construction of a formal specification of the system.

For instance in the case of the strategies proposed in [Conger *et al.*,1990] and in [Larsen *et al.*,1991], first Structured Analysis heuristics are used to develop a top-down hierarchically partitioned data-flow diagram. Then formal specifications are derived according to the overall architecture provided by the DFD set: data-flows and data stores are described in the abstract syntax and a VDM specification is produced for each data transformation process in the DFD set.

The main limitation of these strategies concerns the fact that most of them are function-oriented, and so far tend to obscur the control aspects.

The second kind of strategies consists in exploiting the best features of many different specification techniques to specify a complex system [Traore,1997]. In this context, formal methods could be used for example just for the description of the critical aspects of the system. Therefore the global specification will consists in several partial specifcations expressed in different languages.

For instance in [Ledru,1996], an approach is proposed where formal techniques are inserted into classical specification formalisms (Entity-Relation, State Transition Diagram and DFD): semi-formal notations are translated or annotated with formal notations (DFD with Z). This permits to look at the specification into finer details and then improve precision.

The case study developped in [Bussow,1996] is closely related to this work. It proposes a separation of concerns throughout multiple views and derives proof obligations for systematic relationships between views: data are descibed with the information model of OMT, the control view is described with statecharts and the functional view is described with Z [Spivey,1989], [Spivey,1992].

The main problem with this kind of strategies concerns the integration of the partial specifications written in different notations. These partial specifications should be integrated in such a way that we obtain a consistent global specification which could serve as a basis for further development.

Several works dealing with this issue have been carried out [Hailpern,1986], [Reiss,1987], [Wile,1992], [Zave,1993], [Nuseibeh *et al.*,1994] etc.

In the approach proposed by Zave in [Zave,1993], [Zave,1996] and [Zave,1997], all specification languages are assigned semantics in the same domain (first-order predicate logic). The semantics of the composition will be the composition of the corresponding specificand sets. One of the main limitation of this approach concerns the complexity of the translation of specification languages into first-order predicate logic.

2.2 Motivations of our Work

In the context described above, it appears that it is necessary to define another kind of strategy taking into account the different aspects of a complex system and proposing a simple format of validation of the global specification.

The best way to achieve this objective consists in using a unified format for the integration of the partial specifications. The global specification obtained, should also take into account all the different aspects induced by the requirements (so the control aspects when relevant).

The work developped in this paper is based on these considerations. The approach proposed consists in the combination of two structured approaches:

- a top-down approach with Satemate [Harel,1996] which highlights the structure of the problem;
- a bottom-up approach, consisting in a systematic translation of the previous semi-formal specification into a global formal specification using VDM [Fitzgerald,1997].

2.3 Selection of the Formalisms

2.3.1 Statecharts/Activity-charts

Statecharts and activity-charts are both notations belonging to the statemate approach [Harel,1996]. Statecharts are dedicated for the modeling of reactive views and activity-charts for the functional view.

Activity-charts can be viewed as a multi-level data-flow diagrams. They describe functions or activities, as well as data-stores, all organized into hierarchy and connected via the information that flows between them. The behavior of a non-basic activity is described by an associated sub-activity, refered to as its *control-activity*. Each control-activity is described by corresponding statecharts in the control view.

Statecharts represent an extensive generalization of state-transition diagrams. They are based on multi-level states, decomposed in and/or fashion, and provide constructs for dealing with concurrency and encapsulation.

Additional nongraphical information related to the views and their flows of information are given in a *Data Dictionary*.

2.3.2 VDM-SL [Fitzgerald,1997]

We retain VDM-SL as the common format because it has proven to be sufficiently expressive for modelling complex components and it has become one of the most widely used formal notations in both academia and industry. VDM-SL is the specification language associated to VDM (Vienna Development method), a model-oriented formal method suitable for the development of transformational systems. VDM-SL provides powerful constructs to deal with complex data structures and to describe the transformations of the system. There is also a module-based mechanism which help in tackling complex systems. But, originally, the language was designed mainly for the description of abstract data types and sequential systems. Thus the language is poor in constructs dealing with control aspects such as time, concurrency etc.

To overcome these limitations, we introduce the notion of *control kernel* as a means to define explicitly the control aspects in a VDM specification.

2.4 Presentation of the Case Study

The case study involves the FANS (Future Air Navigation System), an avionics project of AEROSPATIALE Aircraft [Boyer,1997].

The main purpose of the FANS is the improvement of the AIr traffic management. Among the main Data Link applications of the FANS, there are the CPDLC (Controller/Pilot Data Link Communication), the AFN (Air traffic facilities Notification) and the ADS (Automatic dependant Surveillance). In this case study, we are interested especially in the AFN.

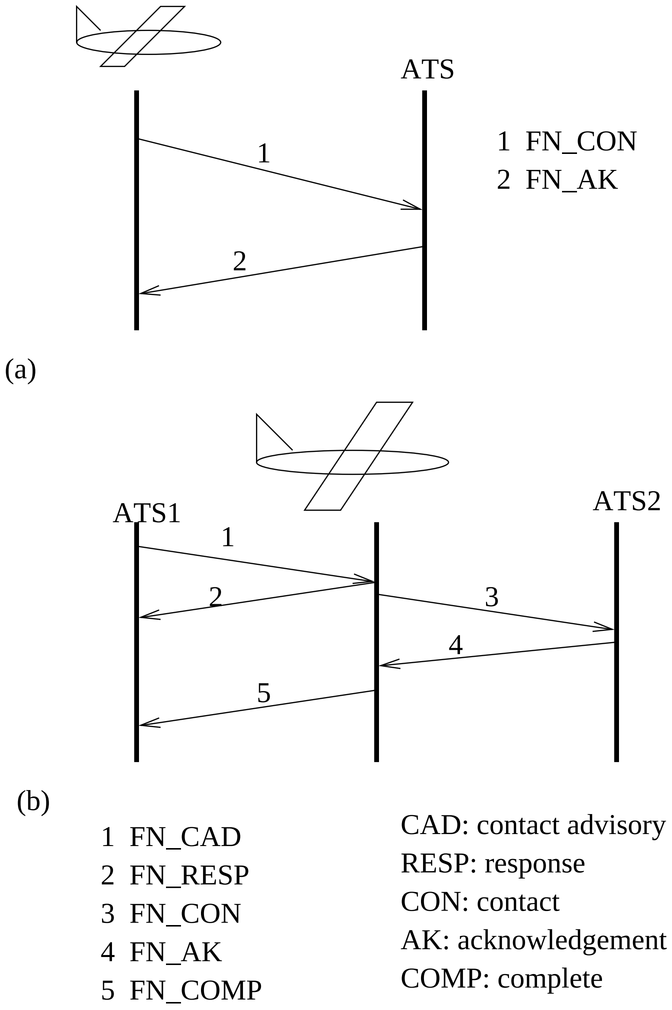

Figure 1: AFN messages

The CPDLC allows aircrafts and ATC center (Air Traffic controller) on ground, to communicate by Data link (instead of vocally). Before the communication is iniated, connexion should be established throughout the AFN.
The AFN allows an ATC center (Air Traffic Controller), to obtain informations about the data link capabilities of an aircraft and to exchange communication addresses. The AFN operates through two main phases:
1. The Log-On: this phase covers the connexion between an Aircraft and a ground ATC center (see figure 1 (a)). It is iniated either on request of the pilot or automatically, by giving the address of the ATC center. A message FN_CON (contact) is then sent to the center, which should reply with another message labelled FN_ACK (acknowledgement).
2. Address Changing: when the aircraft reachs the limits of of the area covered by the current center, this latter will send to the pilot, the message FN_CAD (contact advisory), in order to ask him to contact a next center (see figure 1 (b)).
At the reception of this message, the aircraft should reply with a first message labeled FN_RESP (response) and later, with a second message FN_COMP (complete), on the completion of the contact with the next center.

3 The Approach

3.1 Approach Overview

The approach proposed in this paper starts with the preliminary specification of the system using statemate, followed by the progressive translation of the obtained specification into VDM-SL.
The approach best detailed by fig 2, consists of the following steps:
1. Specification with Activity-charts: the requirements analysis starts with the functionnal specification with Activity-charts. This will consist mainly in the definition of the context diagram, composed by the top-level activity, the external processes and the information flows between the system and the environment.
2. Decomposition: the context diagram is then refined in a number of subactivities , data-stores and a control activity. This process is repeated for every subactivity until an acceptable level of detail

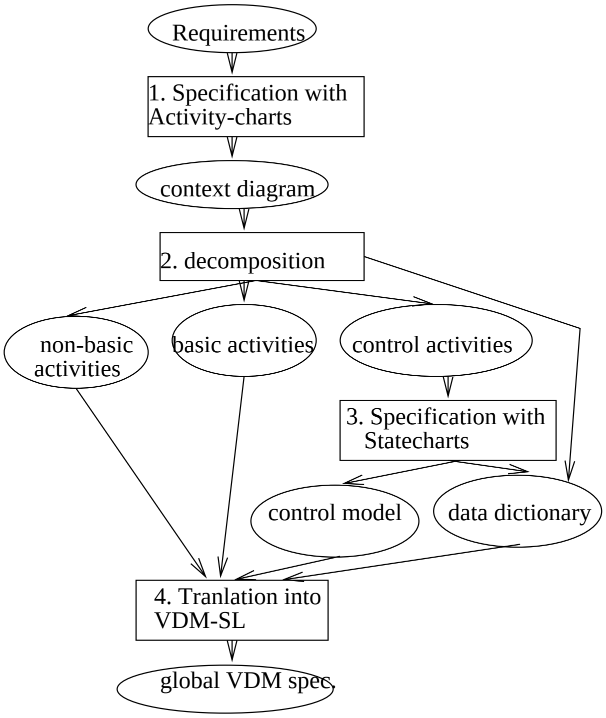

Figure 2: The General Approach

is reached.
Finally, we obtain a hierachy of activities including a set of non-basic activities (activities which require further decompositions), a set of basic activities (activities which don't require other decompositions) and a set of control activities (activities describing the control behavior of their parent activities).
In our approach, the definition of data and basic activities will consists only of the graphical informations provided. It will not be necessary to give textual information through the Data dictionnary as it is normally the case with Statemate.
3. Specification with Statecharts: for each control-activity, we give a corresponding Statecharts description. Textual information about these statecharts are also given in the Data-Dictionnary.
4. Translation into VDM-SL: the statemate specification obtained during the previous step is then translated into a VDM-SL specification. At the end of this step, we obtain global formal specification which is used as the basis of the formal design.

3.1.1 Example

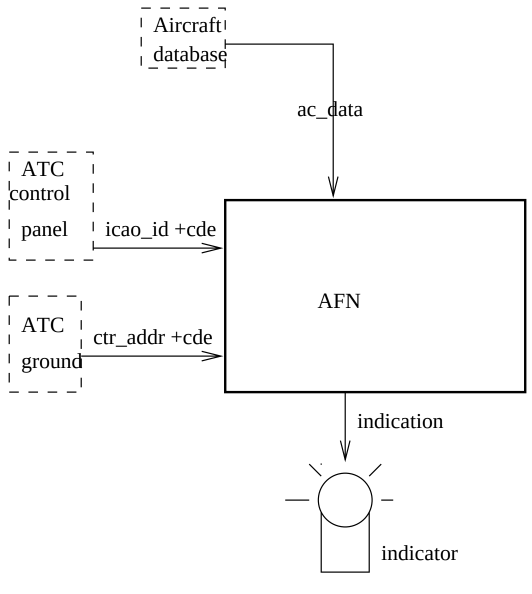

Figure 3: The AFN context diagram

We start the analysis of the case study by giving the context diagram of the AFN system (see figure 3). The environment is composed of several external activities such as the onboard ATC control panel (for typing addresses), the aircraft database (references of onboard AFN applications, aircraft position etc.), a ground ATC center and onboard indicators (presenting the success or failure of message exchange).
We refine the top-level activity *AFN* during successive steps, so that an acceptable level of detail is reached. This will give rise to a hierarchy of activities composed of basic activities, non-basic activities and control activities. Figure 4 gives an overview of this hierarchy.
The first level of refinement leads to two non-basic subactivities *BPROCESS* AND *GPROCESS*, and a control activity *afn_sc*. *BPROCESS* describes the data transformations onboard, while *GPROCESS* corresponds to the ground transformations.
The activity *BPROCESS* is refined in two non-basic activities, *CONTACT* and *BCHECK*, a basic activity, *COMPLETE* and a control activity *bp_sc*.

- The purpose of *CONTACT* is to generate the message *FN_CON* on request of either the pilot or the ground ATC center; in the latter case, the message FN_RESP is also generated.
- The purpose of *BCHECK* is to check the validity of the message received onboard (i.e. FN_AK and FN_CAD) and to give to the pilot an indication concerning the success or failure of the process of message exchange.
- The purpose of *COMPLETE* is to generate the message *FN_COMP*.

The activity *CONTACT* is refined in four basic subactivities: *DSP* (transformation of the OACI code typed in by the pilot into a seven characters center address), *CNORM* (generation of message FN_CON on pilot request or automatically), *CADV* (generation of message FN_CON on a ground ATC center request) and *RESPONSE* (generation of message FN_RESP).
The control activities defined in the preceeding models should be described with statecharts. The statechart corresponding to *con_sc*, the control activity of *CONTACT* is represented by figure 5. The activity starts in state *init*; the generation of event

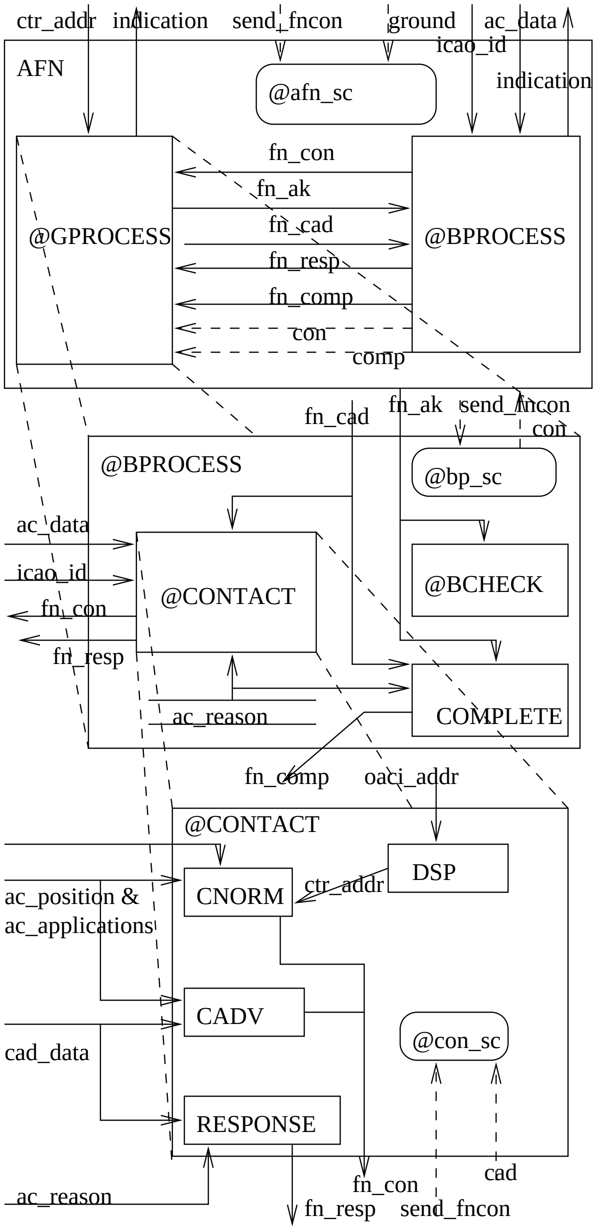

Figure 4: The AFN hierarchy

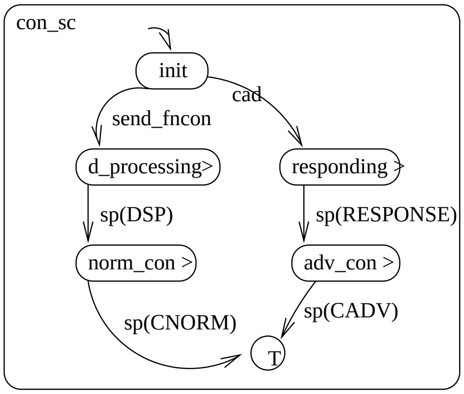

Figure 5: Statecharts describing control activity con_sc

send_fncon starts the execution of activity *DSP* in state *d_processing*, followed by the execution of activity *CNORM* in state *norm_con*, and then the deactivation in state *T*. The generation of event *cad* (contact advisory), when state *init* is active, leads to the execution of activity *RESPONSE* in state *responding*, followed by the execution of activity *CADV* in state *adv_con*.

3.2 The Translation Process

The translation into VDM is performed, bottom-up, in a structured manner. The translation relies on the definition, for each non-basic activity, of a corresponding VDM module. Each module consists of the following kinds of elements:

- a set of data types corresponding to the translation of the data associated to the relevant activity;
- a set of VDM functions and operations corresponding to the basic subactivities;
- a set of VDM data types, functions and operations corresponding to the translation of the control subactivity.

The translation of control activities relies on the definition of what we refer to, in this work, as the *control kernel*. The *control kernel* is the set of the different elements which synchronise the different processes of the system. Deriving from the translation of the statecharts associated to control activities, it is composed of a predefined VDM module named *module CK* (for control kernel) and of a set of elements (data types, operations, state and values definitions) specific to each control activity, so to the module correponding to its parent activity.

Instead of giving textual definitions (in the data Dictionnary) for the data involved by a non-basic activity, a direct translation is given as VDM data types definitions.

The basic functionnalities involved by a non-basic activity, are also directly defined as VDM functions or operations. By basic functionnalities, we mean the basic subactivities as well as the actions attached to the transitions and to the static reactions associated to the statecharts corresponding to the control subactivity.

The VDM module associated to a non-basic activity describes the different elements resulting from its refinement, but in order to be able to handle easily its representation in VDM, we introduce another operation, identified as *TRANSFER* operation, as the definition of its global behavior.

The translation process is performed algorithmically through the following steps:

1. Definition of *module CK* (which is predefined)

2. Tranlation of non-basic activities which have only basic subactivities, in the following way:
   (i) translation of the data involved, directly into VDM data types;
   (ii) definition of the basic functionnalities involved, directly in VDM notation;
   (iii) translation of the control subactivity by introducing the specific elements of the control kernel.

3. Translation of non-basic activities which non-basic subactivities have already beeen translated, in the following way:
   (i) importation (by the relevant VDM module) of TRANSFER operations corresponding to the non-basic subactivities;
   (ii) step 2.

4. Iteration of step 3 until the top-level activity is translated.

3.2.1 Example

If we consider the activity-charts hierarchy given in figure 4, the translation step corresponding to *GPROCESS*, starts with the definition of the modules corresponding to the non-basic activities *CONTACT* and *BCHECK*, in either order, because all their subactivities are basic activities.

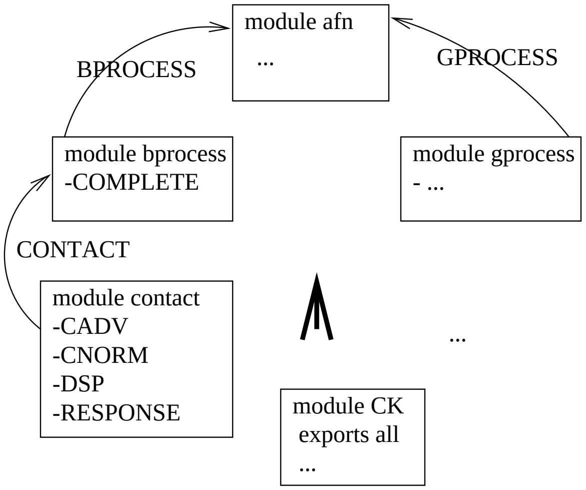

Figure 6: The translation process

For instance, for the translation of *CONTACT*, a VDM module named *module contact* is defined. This module will contain the definitions of the data types corresponding to the data-items of activity *CONTACT*. The basic subactivities of *CONTACT* (i.e *CADV, CNORM, DSP* and *RESPONSE*) are directly defined as VDM operations or functions in *module contact* (see figure 6). The control activity *con_sc* is also translated throughout the control kernel.

A similar way is followed for the activities derived from the refinement of *GPROCESS*.

The next step will concern the activities *GPROCESS* and *BPROCESS*, in either order, since all their non-basic subactivities (i.e. *CONTACT* etc.) have already been translated, during the previous step.

For activity *BPROCESS*, a VDM *module bprocess* will be defined in the same way as in the previous step, except the fact that its non-basic subactivities (i.e. *CONTACT* and *BCHECK*), should be represented in this module, by the *importation* of their corresponding *TRANSFER* operation.

Thus subactivity *CONTACT* is represented in *module bprocess*, by the *importation* from *module*

contact of the operation *TRANSFER* renamed CONTACT (see figure 6).
Finally,the last steps will concern the top-level activity *AFN*, for which, a module labelled *afn* is defined. This module will *imports* the *TRANSFER* operations corresponding to activities *BPROCESS* and *GPROCESS* from the rtelevant modules.
In the end, the VDM global specification will consist of the modules corresponding to the non-basic activities and *module CK*, which is predefined and belongs to the control kernel.

3.3 Data Translation

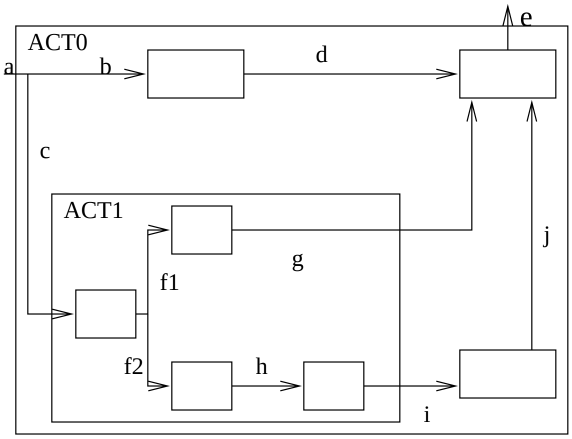

Figure 7: Internal data/Interface data

In this approach, in order to be able to take advantage of the rich set of mechanisms offered by VDM for data modelling, we propose a direct definition in VDM notation.
Given a non-basic activity, we consider two kinds of data:

1. **Internal data** which are produced and consumed locally (e.g. by the subactivities). For instance in the case of activity ACT1 (in fig. 7), *f1,f2* and *h* are the internal data.
2. **Interface data** which are either produced locally and consumed externally, or produced locally and consumed externally. In fig. 7, *c,g* and *i* are examples of external data (for ACT1).

Data translation will be performed in this setting under the following rule:
Rule 1
There are two cases for data definition:

1. *If the relevant activity is not the top-level activity, the data types corresponding to the internal data are defined in its corresponding VDM module while the data types corresponding to the interface data are imported from the parent activity.*

2. *For the top-level activity, the data types corresponding both to internal and interface data are defined in the corresponding module.*

3.3.1 Example

The data involved in activity *CONTACT* are the following (see figure 4): *oaci_addr, ctr_addr, ac_data, ac_position, ac_application, ac_reason, fn_con, fn_resp, fn_cad.* There is only one internal data: *ctr_addr* which corresponds to a seven characters ATC center address.
So, *module contact* should define the data type *n_address* corresponding to *ctr_addr* and imports the data types corresponding to the other data. We give in the following an overview of this module:

```
module contact
  imports   from CK all
            from afn types        address;
                                  ac_data;
                                  position;
                                  application_con;
                                  msg_con;
                                  msg_resp;
                                  msg_cad
            from bprocess types   reason
  exports   ...
...
definitions
types
n_address = seq of char
inv na == len na = 7;


  ...
end contact
```

4 The Control Kernel

The control behavior of a non-basic activity is represented by its control activity which in its turn, is described by a statechart. So the translation of the control data will consist in the translation of the related statechart.

The rationale behind the translation consists in defining in VDM, generic data types and operations corresponding to the basic semantic features of Statechart notation, such as *state, event, transition etc.* The translation of a specific statecharts will then consist in the adaptation of these generic elements.

We give, in this study, the name of *control kernel* to the sum of these generic elements.

The elements of the control kernel are shared among a predefined module identified as *module CK* and the different modules corresponding to the translation of the non-basic activities.

There are four kinds of elements: data types, state variables, functions/operations and values definitions.

4.1 Data Types

We map the basic features of statecharts with VDM data types; this give rise to the following definitions:

4.1.1 Data types deriving from basic features

We define a token type for events, conditions and transformations (this include all kinds of transformations):

event = token; condition = token; action = token

In order to be able to access to the value of conditions, we define a map type relating condition to their truth value:

condition_value = map condition to bool

4.1.2 Static reactions

The reactions attached to a state in the data Dictionary are called *static reactions.* Associating the reaction *trigger/action* with state *s* in the data Dictionary means that as long as the system is in state *s*, the action is performed whenever the trigger occurs.

We define a composite type labeled *reaction*, as follows:

reaction :: act: action
trigger: [event]

where *act* represents the associated action and *trigger*, the triggering event.

4.1.3 States

we translate the notion of *state* as a composite data type called *state_type*:

state_type :: direct_substate: set of state_type
instate: condition
P_activity_T: set of action
P_activity_W: set of action
P_static_R: set of reaction
inv s == s not in set s.direct_substate.

with

- *direct_substate*: the set of direct substates of the considered state;
- *instate*: a condition corresponding to the statechart primitive *in(state)*, indicating the activation status of the state;
- P_activity_T: the set of activity defined "throughout" the state;
- P_activity_W: the set of activity defined "within" the state;
- P_static_R: the set of static reactions associated to the state.

4.1.4 Configuration

The configuration is defined in Statemate as the set of maximal states where the system resides simultaneously. We define a corresponding VDM type as follows:

configuration = set of state_type
inv C == (root in set C) and
(forall s in set C &
(if Andstate(s)
then s.direct_substate subset C
elseif Orstate(s)
then card (C inters.direct_substate) = 1
else s.direct_substate = {}))

root is the identifier of the root state; the previous definition means that a configuration is a set *C* of states obeying to the following rules:

- *C* contains the root state;
- If *C* contains an OR-state *A*, then it must contain exactly one of A substates;
- If C contains an AND-state A, then it must contain all the substates of A.

4.1.5 System status and executions

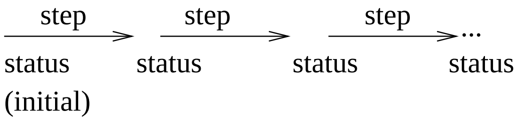

Figure 8: System execution

The behavior of a system described with statemate is a set of possible executions, representing system responses to a sequence of stimuli from the environment.
An execution consists in a sequence of detailed snapshots or *status* of the system. The first element in the sequence is the initial status; the other elements are deduced from their predecesor by executing a step (see fig. 8).
A sytem status is composed, in statemate, of the following elements:

- the system configuration
- the actual data values
- the truth value of the conditions
- the set of the events generated internally and the set of scheduled actions and events
- the system history informations

We define the corresponding data type in VDM, as follows:

```
    status ::  d1: data_type1
               ...: ...
               dn: data_typen
               INT: set of event
               C_value: condition_value
               conf: configuration
```

where fields *d1* to *dn* corresponds to the data involved, *INT* is the set of event generated internally, *C_value* describes the correpsondence between the relevant conditions and their truth values, *conf* is the current configuration of the system.
We define another type corresponding to system executions, as a sequence of status:
run = seq of status
Example: We give in the following the definition of type *status* corresponding to *module contact*; the above skeleton is adapted by including the fields corresponding to the different data involved.

```
module contact
...
definitions
types
n_address = seq of char
inv na == len na = 7;
    status ::  d1:afn' address
               d2: n_address
               d3: afn'acdata
               d4: afn'position
               d5: seq of afn'application_con
               d6: bprocess'reason
               d7: afn'msg_con
               d8: afn'msg_resp
               d9: afn'msg_cad
               INT: set of CK'event
               C_value: CK'condition_value
               conf: configuration ;
run = seq of status;
configuration = ...
...
end contact
```

4.1.6 Transitions

Transitions are represented in Statemate by the following syntax: *e[c]/A*, where *e* is the triggering event, *c* is the triggering condition and *A* is the implied action.
We define a composite data type as the representation framework of transitions in VDM, as follows:

```
    transition ::  trig_ev: [event]
                   trig_cond: [condition]
                   source: set of state_type
                   target: set of state_type
                   Paction: set of action
    inv t ==       t.source <> {} and
                   t.target <> {}
```

with *trig_ev* the triggering event, *trig_cond* the triggering condition, *source* the set of depart states, *target* the set of target states and *Paction* the set of actions associated to the transition.

4.1.7 Abstract representation of a statecharts corresponding to a control activity

The main objective of the control kernel is to represent, in VDM notation, the control information which is carried by control activities. We have introduced above, the different elements of a statechart individually. We propose to integrate these different elements in a global data structure which could be considered as an abstract data type representation of a statechart describing a control activity. Abstract data types have been formally defined by Guttag and Horning in [Guttag,1978]. We define the following composite type:

```
sc_structure ::  Pstat: set of state_type
                 Pevent: set of event
                 Pcondition: set of condition
                 Paction: set of action
                 Preaction: set of reaction
                 Pactivity: set of action
                 Ptransition: set of transition
inv mk_sc_structure(Ps,Pe,Pc,Pa,Pr,Pav,Pt) ==
(forall sr in set Pr & (sr.trigger in set Pe
and sr.act in set Pa)) and (forall s in set Ps
& ((s.direct_substate subset Ps)
and (s.instate in set Pc) and
( (dunion {s.P_en_action,s.P_ex_action,
s.P_activity_T,s.P_activity_W}) subset Pa)
and ({s.P_static_R} subset Pr))
and (forall t in set Pt &
((t.trig_ev in set Pe) and (t.trig_cond in set Pc)
and ((dunion {t.source,t.target}) subset Ps)
and (t.Paction subset Pa))))
```

With *Pstat* the set of states, *Pevent* the set of events, *Pcondition* the set of conditions, *Paction* the set of actions, *Preaction* the set of static reactions, *Pactivity* the set of actions and *Ptransition* the set of transitions.

4.2 State Definition

For each module correponding to the translation of a non-basic activity, we define a global state composed of three fields:

History: in order to capture the history of the system, we define a generic state variable labelled *ST* representing the execution of the corresponding non-basic activity; *ST* is of type *run*.

Occured events: we define the state variable *occur* as the set of all the events generated (internally and externally) at the beginning of a step.

External events: we represent the set of external events by a state variable labelled *EXT*.

So we can give the following template for state definition within the modules:

```
state ... of
ST: run
occur: set of event
EXT: set of event
end
init s == s = ...
```

4.3 Predefined Functions and Operations

Besides the data types definitions, we define a number of functions and operations for the description of statecharts.
There are three main operations labeled *EXEC_action*, *EXEC_step*, *TRANSFER* and several auxiliary functions/operations used in the definitions of the main operations.

4.3.1 Operation EXEC_action

We encapsulate the set of basic functionnalities of a non-basic activity (e.g. basic subactivities and actions) in a VDM operation, in order to be able to handle them as a whole. We label this operation as *EXEC_action* and give the following template, which should be adapted according to the non-basic activity:

```
    EXEC_action : set of action*status ==> status
    EXEC_action(A,sta) ==
    (dcl i: status := sta ;
    if A <> {}
    then ( for all a in set A do
    ( cases a:
    mk_token(“OP1”) - > ... := OP1(...)
    ... - > ...
    mk_token(“OPn”) - > ... := OPn(...)
    others - > skip
    end ) )
    else skip;
    return i )
```

The OPi representing the VDM operations corresponding to the basic functionnalities.

Example: we give in the following, the definition of *EXEC_action* corresponding to *module contact*. The basic transformations involved are *DSP, CNORM, CADV, RESPONSE* (see figure 4).

```
module contact
...
definitions
...
operations
    CNORM: n_address*afn'acdata*afn'position
    *seq of afn'application_con ==> afn'msg_con
    CNORM(ad,ac_d,pos,app) ==
    (dcl i: msg_con;
    – message d'entête
    i.head.mfi := < B0 >;
    i.head.ctr_addr := ad;
    i.head.imi := < AFN >;
    i.head.mti := < FMH >;
    i.head.ac := ac_d;
    – corps du message
    i.tail.mti := < FPO >;
    i.tail.curr_pos := pos;
    i.tail.act_fl := 1;
    i.tail.con_app := app;
    return i )
    pre app(len app).ap_name = mk_token(”AIF”);
...
    EXEC_action : set of CK'action*status
    ==> status
    EXEC_action(A,sta) ==
    (dcl i: status := sta ;
    if A <> {}
    then ( for all a in set A do
    ( cases a:
    mk_token(“dsp”) - > i.d2 := DSP(i.d1)
    mk_token(“cnorm”) - >
    i.d7 := CNORM(i.d2,i.d3,i.d4,i.d5)
    mk_token(“cadv”) - >
    i.d7 := CADV(i.d9,i.d4,i.d5)
    mk_token(“resp”) - >
    i.d8 := RESPONSE(i.d9,i.d6)
    others - > skip
    end ) )
    else skip;
    return i )
  ...
end contact
```

4.3.2 Operation EXEC_step

The execution of steps could be considered as the basic units describing the behavior of a system in Statemate. So we introduce the VDM operation labelled *EXEC_step* as the description of step execution. The following definition is given:

```
    EXEC_step : set of event ==> status
    EXEC_step(oc) ==
    (dcl i: status, j:status,s: status,
    S: set of state_type, H:set of transition;
    s := ST(lenST);
    H := {t | t in set SC.Ptransition
    & enable(t,s,oc) };
    s.INT := {};
    if ( H <> {} or oc <> {} )
    then ( i := TRANSIT(s,oc,H);
    S := i.conf inter s.conf;
    j := STATIC_reaction(i,oc,S);
    ST := ST[j])
    else skip;
    return ST(len ST) );
```

Step execution is function of the current status and of the set of events (internal and external) available at the beginning of the step.

There are two phases: firstly the transition from a configuration to another, described by the auxiliary operation labelled *TRANSIT* and secondly the execution of static reactions, described by the auxiliary operation *STATIC_reaction*.

The transition step is based on the set H of the enabled transitions chosen among the global set of the transitions associated to the control activity which is represented by the value SC (see section 4.4).
The enabling status of a transition is described by another auxiliary operation denoted *enable*.
The definitions and details of these operations could be found in [Traore,1998].

4.3.3 Operation TRANSFER

Operation EXEC_step describes the behavior of a non-basic activity only during a step; in order to have a representation of its global behavior, we introduce an operation labelled TRANSFER which is built from the definition of EXEC_step and the structure of the corresponding control activity. The structure of a control activity depends on the termination type of its parent activity.
There are three kinds of activities according to the termination type: activities that have self-termination, activities that have controlled-termination, activities that have both and which are considered as self-terminating.
The control activity associated to a self-termination activity has a *T-connector*, which is equivalent to a basic state that we label T; transition towards this connector, stops the statecharts and the activity is deactivated.

The controlled-termination activities are stopped externally, most of the time, by the generation of an event that we label *stop*. We give a different template as the definition of *TRANSFER* for each kind of activity. Consider (see figure 9) a non-basic activity A with *e1,...,ep* the input data, of respective types *E1,...,Ep* and *s1,...,sn* the output data, of respective types *S1,...,Sn*. As, we saw in the definition of type *status*, each data has a corresponding *status* field. We consider that these fields are *d1,...,dn+p* with *d1* to *dp* corresponding respectively to *e1,...,ep* and *dp+1* to *dn+p* correspoding respectively to *s1,...,sn*.

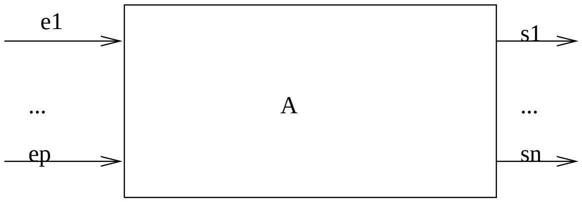

Figure 9: Building of operation TRANSFER

We give in the following, the template corresponding to self-terminating activities, which is a combination of the other two cases.

```
TRANSFER: E1*...*Ep ==> S1*...*Sn
TRANSFER(e1,...,ep) ==
(dcl i: status, j1: S1,...,jn: Sn;
ST(len ST).d1 := e1;
...
ST(len ST).dp := ep;
while ( (stop not in set EXT) or
(T not in set ST(len ST).conf ) do
( occur := EXT union ST(len ST).INT;
i := EXEC_step(occur) ;
occur := {}) ;
j1 := i.dp + 1
...
jn := i.dn + p;
return mk_(j1,...,jn) )
```

Input data are processed in an execution loop, througout successive steps. The loop is exited only if the final state T is reached or if the event *stop* is generated. The templates corresponding to the other kinds of control activity, could be found in [Traore,1998].

4.4 Values Definitions

We define a parameter labelled SC representing the abstract representation of the statechart asociated to a control activity; this parameter is of type *sc_structure*.
The definition of the specific part of this statechart will consist in giving the value definition of SC in the relevant VDM module. Consequently, we should also give values definitions for all the elements (states, transitions, reactions etc.) used in the value of SC. We give in the following, an overview of the value section of *module contact*; the corresponding statechart, i.e. *con_sc*, is given in figure 5.

```
module contact
...
values
init : CK'state_type =
mk_CK'state_type({}, in_init, {}, {}, {});
d_processing : CK'state_type =
```

$mk\_CK'state\_type(\{\}, in\_d\_proc, \{DSP\}, \{\}, \{\});$
$\ldots$
$t1 : CK'transition = mk\_CK'transition(send\_fncon, nil, \{init\}, \{d\_processing\}, \{\});$
$t2 : CK'transition = mk\_CK'transition(cad, nil, \{init\}, \{responding\}, \{\});$
$\ldots$
$SC : CK'sc\_structure = mk\_CK'sc\_structure(\{con\_sc, init, d\_processing, norm\_con, responding, adv\_con, T\}, \{send\_fncon, cad, sp\_DSP, sp\_RESPONSE, sp\_CADV, sp\_CNORM\}, \{in\_con, in\_init, in\_d\_proc, in\_norm, in\_resp, in\_adv, in\_T\}, \{\}, \{\}, \{DSP, CNORM, RESPONSE, CADV\}, \{t1, t2, t2, t3, t4, t5, t6\})$
$\ldots$
end contact

The definition of *SC* is composed of the set of states, the set of events, the set of conditions etc. We give also the definitions of the different elements invoked in the definition of *SC*. For instance, we define state *init* with the set of its direct substates which is {} (i.e. empty), its associated condition *in_init*, the set of the activities defined *throughout* which is empty etc. Other example is transition *t1* which is defined by its triggering event *send_fncon*, by its triggering condition which is *nil*, by its source states {init} , by its target states {d_processing} and by the set of its associated actions which is empty.

4.5 Optional Elements

In order to simplify our translation model, we define some features provided by Statemate as optional elements, since these elements are not systematically relevant.
As optional elements, we consider the following:

- timing features which are represented by statemate primitives for timeout event or scheduled action;
- conflicting transitions;
- history connectors

We define data types and operations corresponding to these features; more informations could be found in [Traore,1998].

4.6 Pratical Use

The different definitions that were given above belong to the *control kernel*. The elements of the *control kernel* are distributed among a VDM predefined module labelled *CK* and the different modules corresponding to the non-basic activities. We give in figures 11, 12 and the fig:template2 the templates of *module CK* and of a standard module corresponding to a non-basic activity.

5 Automated Translation

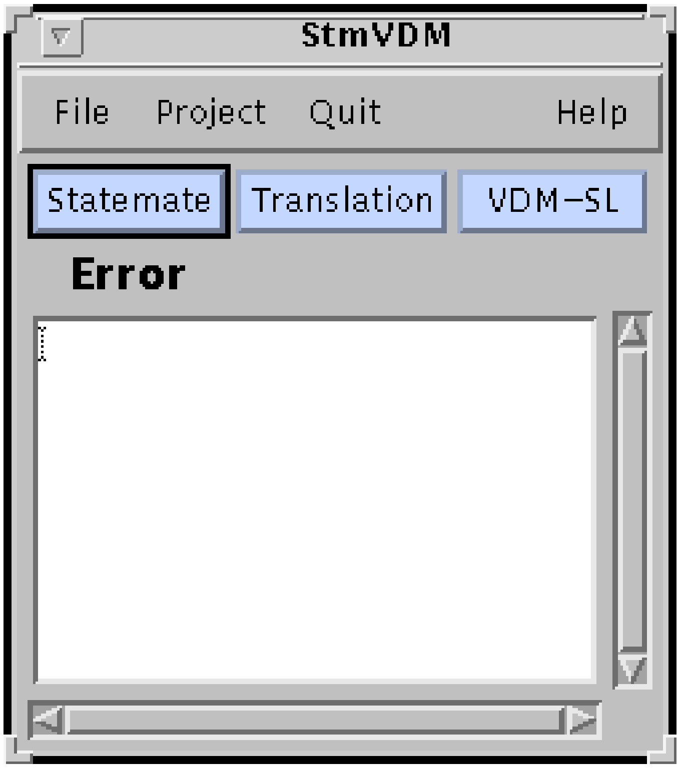

Figure 10: Stmvdm: graphical user interface

We develop a tool supporting the translation process; the main window of *StmVDM* the corresponding environment is represented by figure

10. This tool is interfaced with the STATEMATE environment of i-Logix [Statemate,1987] and with the VDM-SL environment of IFAD [IFAD,1996]. These environments could be started indepently by clicking on the corresponding buttons.

The translation process is started by clicking on the button *Translation*: the user should create a project by giving the path (workarea) and name of the STATEMATE project containing the initial specification. The translation carried out interactively, will lead to a file containing the global specification in VDM.

6 Concluding Remarks

Several case studies were successfully developped with the proposed approach (see [Traore,1998] for the case study of an access control system). This shows the approach is practical for realistic systems development and is capable of dealing with applications in a convenient fashion.

One of the main advantage of this approach is that it helps reduce the complexity of formal specification and provide a comprehensible structure. The existence of an effective tool support extends the applicability of the approach to large scale projects. The different definitions proposed in the approach were checked by simulation with the VDM-SL toolbox of IFAD. For highly critical system, it is sometimes required to check the system by genertating proof obligations. In [Traore,1997], we have already defined the different kinds of proofs obligations involved in the process of translating a statecharts specification to a VDM one. Future research will concern the adaptation and the extension of these proof obligations to the approach.

Another advantage of the approach concerns the fact that important features of Statemate such as concurrency, synchronisation, visibility and time are preserved by the translation process. A number of approaches have been proposed for handling concurrency in VDM [Jones,1983b], [Stolen,1991], none of these have yet achieved the level of use needed to make it part to ISO standard VDM-SL. In systems where part of the state might be changed by several processes, classical post-conditions aren't enough to fully express an operation's functionality; this applies also to other formal methods as well. We refer the reader to [Traore,1998] for more about how concurrency or synchronisation are described in the resulting VDM specification.

The concept of *control kernel* is not an extension of VDM but rather an adaptation, since instead of adding new constructs, we proceed by using judiciously existing constructs. Future works will concern, also, the simplification of the structure of the *control kernel*; it may be possible to define all the elements of the *control kernel* in a unique VDM parameterized module, instead of distributing them as it is actually the case.

References

[Andrews,1988] Andrews D.,Gibbins P., *An introduction to formal methods of software development*, The open university Press,Milton Keynes,UK,1988,units 1-4.

[Arago,1997] ARAGO 20, *Application des Techniques Formelles au Logiciel*, OFTA Paris, 1997.

[Babin *et al.*,1991] G. Babin, F. Lustman, P. Shoval, *Specification and design of transaction in information systems:a formal approach*, IEEE Trans. sw. eng. 17,8(Aug. 1991),P 814-829.

[Bowen,1995] J.P. Bowen, M.G.Hinchey,*Seven more myths of formal methods*, IEEE sw,July 1995,P34-41.

[Boyer,1997] E. Boyer, *Modélisation Fonctionnelle et Validation des Exigences des Utilisateurs application à la Fonction FANS*, ENSICA, 1 Pl. Emile Blouin 31056 Toulouse, France.

[Bussow,1996] R. Bussow, M. Weber, *A Steam-Boiler Control Specification with Statecharts and Z*, Formal Methods for Industrial applications, P. 109-128, Springer-Verlag, Berlin, 1996.

[Conger *et al.*,1990] S.A. Conger, M.D. Fraser, R.A. Gagliano, K. Kunar, E.R. Mc Lean, G.S. Owen,V.K. Vaishnavi, *A structured stepwise refinement method for VDM*, Proceedings of the annual Conference

```
module XX
 imports form CK all,
-- importation of data types corresponding to
-- parent activity YY

   from YY  types T1;...;Tp ,
-- importation of operations corresponding .
-- to non-basic subactivities SubAct1...
from ZZ1 operations TRANSFER: ... ==> ...
                                        ...
from ZZr operations TRANSFER: ... ==> ...
 from ...
exports
-- exportation of types corresponding to
-- interface data of subactivities
    types Ti;...;Tj;....
-- exportation of the opération corresponding
-- to the current non-basic activity XX
    operations TRANSFER: ... ==> ...
 definitions
  types
 -- definition of internal data types
  Tp+1...;
     ...
  Tp+n ...;

-- definitions of type status and run
   status :: ...;
   run = seq of status
   configuration = ...;
 values
 -- value definition of the control statecharts

  SC: sc_structure = mk_sc_structure(....);

  -- definition of the initial status
    sti: status = mk_status(...)

 state XX of
     ST: run
     occur: set of CK‘event

     EXT: set of CK‘event
      init s == s = mk_XX([sti],{...},{...})
```

Figure 11: Template of a module corresponding to a non-basic activity

```
 functions
  Andstate: state_type -> bool
  Orstate: state_type -> bool
 ...
 operations
  -- definitions of operations corresponding
 -- to basic subactivities

      OP1: ... ==> ...
        ...
      OPq: ... ==> ...
-- definitions of predefined operations
EXECUTE_step: set of event ==> status

STATIC_reation: status*set of event
                      *set of state_type ==> status
TRANSIT: status*set of event*set of transition
                ==> status
enable: transition*status*set of event ==> bool
INstate: state_type ==> bool
-- definitions of optional operations
           ...

 --  definitions of semi-predefined operations
EXEC_action: set of CK‘action*status
                         ==> status
 TRANSFER: ... ==> ...

 end XX
```

Figure 12: Template of a module corresponding to a non-basic activity

```
module CK

exports all


definitions
 types
   sc_structure::  ...;
   state_type:: ...;
   configuration = ...;
   transition :: ...;
   event = token;
   condition = token;
   condition_value = map [condition] to bool;
   action = token;
   time = nat;
   mtimeout = map event*time to event;
   mhistory = map transition to Htransition;
   Htransition :: ...;
   Htype = <H> | <H*>

 functions
   Substate: state_type -> state_type
  Lower: set of state_type -> state_type
  low: state_type*state_type -> bool
  Conflict: transition*transition -> bool
  Priority: transition*transition -> [bool]
 operations
Nonconflictset:set of transition ==> set of transition

end CK
```

Figure 13: Template of module CK

on ADA technology ANCOST,Inc,Culver City,Calif.,1990,P 311-320.

[Fitzgerald,1997] J. Fitzgerald, P.G. Larsen, *Modelling Systems: Practical Tools and Techniques in Software Development*, cambridge University Press, 1997.

[Fraser *et al.*,1994] M.D. Fraser, K. kumar, K. Vaishnavi, *Strategies for Incorporating Formal Specifications in Software Development*, Communications of the ACM,Vol 37, No 10, Oct.1994.

[Guttag,1978] J.V. Guttag and J.J. Horning, *The Algebraic Specification of Abstract Data Types*, ACTA Informatica, 10, 1978.

[Hailpern,1986] B. Hailpern, *Multiparadigm languages and environments (guest editor's introduction to a special issue)*, IEEE Softw. 3, 1, Jan. 1986.

[Harel,1996] D. Harel, M. Politi, *Modeling Reactive Systems with Statecharts: the Statemate Approach*, i-LOGIX-INC, 1996.

[IFAD,1996] The VDM Tool Group, *The IFAD VDM-SL Language*, IFAD-VDM-1, Dec. 1996.

[Jones,1983b] C.B. Jones, *Tentative steps towards a development method for interfering programs*, Transactions on programming Languages and Systems, 5(4):596-619, Oct. 1983.

[Jones,1990] C.B. Jones,*Systematic software development using VDM*, 2d ed.,Prentice-Hall,Englewood Cliffs,NJ,1990.

[Kemmerer,1990] R.A. Kemmerer, *Integrating formal methods into the development process*, IEEE Trans. sw eng. 15 (Oct. 1990),P 37-50.

[Larsen *et al.*,1991] P.G. Larsen, J.V. Katwijk, N. Plat, K. Pronk, H. Toetenel, *SVDM: An Integrated Combinaison of SA and VDM*,Methods Integration Conference, Leeds, Sept. 1991.

[Ledru,1996] Y. Ledru, *Complementing Semi-Formal Specifications with Z*, proceedings of KBSE'96,p. 52-61, 1996.

[Miriayala,1991] K.Miriayala, M.T. Harandi, *Automatic derivation of formal software specification from informal description*, IEEE Trans. sw eng. 17,10 (oct.1991),P 1126-1142.

[Nuseibeh *et al.*,1994] B. Nusseibeh, J. Kramer, A. Finkelstein,*A Framework for Expressing The Relationships between Multiple views in Requirement Specification*, IEEE Trans. Soft.Eng, Vol 20, No.10, 1994, pages 760-773.

[Reiss,1987] S.P. Reiss, *Working in the Garden Environment for Conceptual Programming*, IEEE Software 4, November 1987, pp.16-27.

[Sahraoui *et al.*,1996] A.E.K. Sahraoui, M. Romdhani, A. Jeoffroy, A.A. Jerraya, *Co-Specification for Co-Design in the Development of Avionics Systems*, Control Eng. Practice, Vol. 4, N0. 6, pp. 871-876, 1996.

[Spivey,1989] J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall International, 1989.

[Spivey,1992] J.M. Spivey, *Understanding Z: A Specification language and its formal semantics*, cambridge University Press, 1992.

[Statemate,1987] STATEMATE, *STATEMATE:the languages of Statemate*, technical report, i-Logic, Inc, 22 Third Avenue, Burlington Mass.01803, USA, 1987.

[Stolen,1991] K. Stolen, *An Attempt to Reason about Shared-State Concurrency in the Style of VDM*, VDM'91: Formal Software Development Methods, pages 324-342, Springer-verlag, Oct. 1991.

[Traore,1997] , I. Traore, A.E.K. Sahraoui, *A multiformalism Specification Framework with Statecharts and VDM*, 22nd IFAC/IFIP Workshop on Real-time Programming, Lyon, France, Sept. 15-17, 1997.

[Traoré *et al.*,1998] I. Traore, A. Jeffroy, M. Romdhani, A.E.K. Sahraoui, *An Experience with a Multiformalism Specification of an Avionics System*, to be published in INCOSE 98, July 25-31 1998, Vancouver, Canada.

[Traore,1998] , I. Traore, *Application Characterization and Multiformalism in Software Engineering*, PhD thesis, LAAS-CNRS,7 av. Colonel Roche 31077 Toulouse France, May 1998.

[Wile,1992] D.S. Wile, *Integrating Syntaxes and their associated Semantics*, USC/Information Institute, Technical Report RR-92-297. Univ. Southern California, 1992.

[Wing,1990] J.M. Wing, *A specifier introduction to formal methods*, IEEE Computer,vol 23,no 9,Sept 90,P8-24.

[Zave,1993] P. Zave, M. Jackson, *Conjunction as Composition*, ACM Trans. on sw Eng.,vol 2,no 4,Oct 93,P 380-411.

[Zave,1996] P. Zave and M. Jackson, *Where Do Operations Come From? a Multiparadigm Specification Technique*, IEEE Trans. on SOFT. ENG., vol. 22, No. 7, July 1996.

[Zave,1997] P. Zave and M. Jackson, *Four dark Corners of Requirements Engineering*, ACM Transactions on Soft. Eng. and Meth., Vol. 6, No. 1, Jan 1997.