A Formal Event Structuring Approach to Real-Time Design

Peter Gorm Larsen*

IFAD – Institute of Applied Computer Science
Munkebjergvænget 17, DK-5230 Odense M, Denmark

Abstract

In the development of real-time systems it is very important to be able to write down the design decisions. Such design decisions are often written in a natural language. In general this results in a voluminous and ambiguous description of the system. We believe that such informal descriptions with advantage can be formalized. The basic idea is to have a number of formal notations in which the designer stepwise can refine the given specification. For each of the formal notations more and more design decisions are made and noted down for a limited aspect of the problem. The other design decisions will then be postponed to a later stage of development where that aspect is taken into account. In this paper we will show an example of how such formal notations could be defined for the design of real-time systems. The idea behind these notations is based on an event-action model and it will therefore be most appropriate for the development of reactive systems.

1 Introduction

It is our experience that the key problem in the development of real time embedded systems is to structure the system appropriately. This is caused by the fact that a designer's intuitions have a totally unstructured form. Thus, design decisions which conceptually belong at different levels of abstraction will be mixed together with an increase of complexity as a consequence. Our proposal is forcing a designer to make the necessary design decisions in a structured manner. The designer must then add more and more design decisions at each level, and will be forced to postpone design decisions for other aspects of the problem to a later stage of development.

We will illustrate this stepwise development idea by a part of the CEDER[1] method (cf. [Elmstrøm90] and [Løvengreen&90]). This method has been chosen because it is specially dedicated to deal with real-time embedded systems. The method is based on a notion of events[2] which makes it most useful for the development of control oriented real-time systems, i.e., systems where the logic of the reactions is more important than the structure of the calculation which the designed system must do. It is not claimed that this method is general but it is very concrete and relatively easily accessible. It is believe that the concreteness of the method and the brevity of the notations ensure that it is applicable in industry.

The CEDER method has six stages where we in this article are especially interested in the second stage, called the 'event structuring' stage. The first stage identifies the interface to the environment and produces a list of the various concepts involved. Thus at the beginning of the event structuring stage all events going into and coming out from the system has been identified and named. The purpose of the event structuring stage is to achieve a description of the protocol of events that the system must be able to handle. This event structuring stage is then further subdivided in four steps reflecting the stepwise idea. Each of those steps has a formal notation to express the design decisions that must be made at that step.

At the first step of the event structuring stage the designer must find the sequential and alternative dependencies between input events (called in-events). Then at the next step the designer must describe the temporal relations between all events. This concerns iterations and events which possibly occurs. Then at the third step it must be determined how the alternative choices introduced at the second level shall be made. Thus, at step two the designer identified that some iterations was needed and then at this step it is determined how those iterations shall terminate. Finally at the last step it is concluded how the complete system is composed.

The different notations used in these steps are all inspired from the synchronous specification language LOTOS (cf. [ISO8807]). However, since it is the intention to make these notations accepted in the industry as many operators as possible has been removed from standard LOTOS without loosing the expressibility we want for this purpose. The idea is to achieve

*Partially supported by the Danish Ministry of Trade

[1] CEDER is an acronym for 'Construction of Embedded DEdicated Real-time systems.'

[2] Events are atomic transactions between the system and the environment carrying data across the boundary of the system.

a small compact syntax which is easy to learn and convenient to work with, without all the disturbing keywords. From this description a CEDER-LOTOS dialect with all the usual LOTOS keywords can automatically be generated if it is desired e.g. for documentation purposes.

Such a synchronous language has been chosen to allow the designer to focus on the logic of reactions. Thus at this event structuring stage the designer should think about the events as being "ideal" (i.e. with instantaneous reactions). At a later stage in the CEDER method timing constraints from the requirement specification will be taken into account.

From section 2 to section 5 we will describe these different steps one by one. Each time we will explain the syntax[3] and the semantics[4] of the used notation. Furthermore we will try to illustrate which kind of design decisions that will be made at that step by means of an ongoing small example. Because of the limited space in this paper this example has been chosen especially to illustrate how limited aspects of the problem are designed at each stage. The given example will only end up with one process and therefore there will not be any communication between internal processes and the complete system will be trivial. However, we believe that this example is able to illustrate how the decisions can be made stepwise in a structured manner. Finally in section 6 we will give a few concluding remarks about the perspectives of such a formal event structuring notation.

2 Event Expressions

At this step of the development the designer will only try to discover sequential and alternative dependencies between in-events. Only in-events are considered because we are only interested in being able to structure input-driven systems. The method does not attempt to design systems which gives output without having had any input. At this step the designer will reach a number of independent maximal event expressions. An event expression is considered maximal if it is not contained in any of the other event expressions. These maximal event expressions are the first candidates to end up as user processes in the designed system.

[3] The syntax is given in EBNF, where terminal symbols are denoted by symbols in a typewriter style font and quoted, while nonterminal symbols are given in italics with lower case letters. Abbreviations are used and explained each time.

[4] The semantics is compositional and it is given by means of transition graphs as they are used in e.g. [Milner89].

2.1 Syntax

ees = *id* ':=' *ee* '.' [*ees*]
ee = *ee* '|' *ee* |
ee ';' *ee* |
'(' *ee* ')' |
inev |
id
inev = *name*
id = *name*

Here the following abbreviations has been used to fit the two column style:

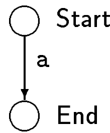
ees = *event-expressions*
ee = *event-expression*
inev = *input event*
id = *identifier*

2.2 Semantics of operators

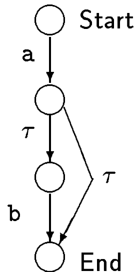
The terminal symbols ':=' and '.' are used respectively to assign an expression to a given name and to separate such assignments. The parentheses '(' and ')' may be used to group sub-expressions together in order to exceed the normal operator precedence rules. All these terminal symbols are common to the different notations and they will therefore not be considered as operators that should be given semantics in any of the following sections.

Op	Meaning
;	If the action taken on an occurrence of the event 'b' (in some cases) implies that the event 'a' must have occurred earlier then there is said to be a sequential dependency from 'a' to 'b', and it is written like 'a ; b'. However, 'a' and 'b' do not need to follow immediate after each other.
|	If the system (in some cases) should take the same action on the occurrence of 'a' as of 'b' then there is said to be an alternative dependency between 'a' and 'b' and this is written like 'a | b'. However, this operator does not indicate how the choice between 'a' and 'b' is taken. In addition this operator does not indicate that 'a' and 'b' are real alternatives (i.e., they may be a part of two larger expressions which are alternatives).

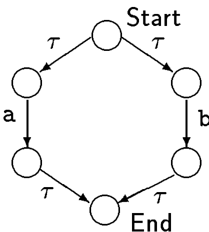
This may be illustrated by the following graphs where the arrows labelled with tau indicates that other (internal) transitions may take place: The graph for a simple event 'a' looks like:



The graph for an expression like '`a ; b`' looks like:



The graph for an expression like '`a | b`' looks like:



Finally it should be mentioned that the operator '`;`' is given higher precedence than '`|`' and thus an expression like '`a ; b | c`' is interpreted as '`(a ; b) | c`' and not as '`a ; (b | c)`'.

2.3 Example

The requirement specification for the example can shortly be summarized like this:

> The system is started by an in-event "**`start`**". It receives the operation mode of the system by the in-event "`mode`". Then the system repeatedly reads an input value and calculate one or more output values: The system must be capable of operating in two modes: a) one in which it calculates summary information (sent by the out-event "**`output`**") based on the value from the in-event "**`input`**". b) in the other mode it calculates three values (sent by the same out-event and based on the same input value). The values sent in "**`output`**" is received outside the system by some equipment. If any one of these values exceeds specific limits determined by the equipment the in-event "**`error`**" will be send to our system, and interrupt the normal mode of operation. Now the system can only continue when an operator sends the in-event "**`restart`**". This event can also be given at any time when the system has been started by the operator if a change of the operating mode is desired.

First a number of sequential and alternative dependencies are identified:

```
start ; mode
start ; input
start ; error
start ; restart
mode ; input
mode ; error
mode ; restart
input ; error
input | error
input | restart
error | restart
```

These dependencies are then gradually grouped together to form one final event expression.

```
calc := start ;
        mode ;
        (
          input
        |
          error
        |
          restart
        ) .
```

Here it is important to notice that only decisions concerning sequential and alternative dependencies are taken at this step of development. Thus, just a limited aspect of the given problem has been considered so far. The out-events has not yet been taken into account, no iterations has yet been incorporated and the interruption is not yet visible.

3 Event Sequences

At this step of the development the designer will concentrate on additional temporal relations between the events. By means of loops and events which may or may not occur the designer will reach a number of event sequences from the maximal independent event expressions. However, at this step the designer does not take any decisions about how the different alternative choices shall be made by the designed system. By choices we here mean termination criterions for loops, and decisions about under which circumstances

an event placed within the maybe operator should occur.

3.1 Syntax

ess	=	*id* ':=' *es* '.' [*ees*]
es	=	*es* '|' *es* |
		es ';' *es* |
		ae
ae	=	*ev* |
		id |
		'(' *es* ')' |
		'[' *es* ']' |
		ae '#'
ev	=	*inev* | *outev*
inev	=	*name*
outev	=	*name*
id	=	*name*

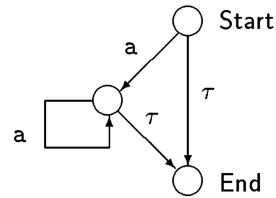
Here the following abbreviations has been used to fit the two column style:

ess	=	*event-sequences*
es	=	*event-sequence*
ae	=	*atomic-expression*
ev	=	*event*
inev	=	*input event*
outev	=	*output event*
id	=	*identifier*

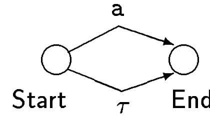
3.2 Semantics of operators

Op	Meaning
#	An expression like a# means that 'a' is repeated a number of times. There is no implicit decision made about how the repetition is terminated.
[.]	An expression like [a] means maybe 'a'. There is no implicit decision made about under which circumstances it should occur.
;	An expression like a ; b means that there is a strong sequential dependency, (i.e., 'a' is immediately followed by 'b').
|	An expression like a | b means that there is an alternative dependency between 'a' and 'b'. However, this operator does not indicate how the choice between 'a' and 'b' shall be taken.

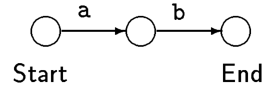
This may be illustrated by the following graphs. The graph for a loop expression like 'a#' looks like:



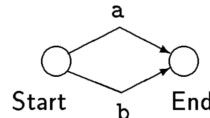
The graph for the "maybe" operator in an expression like '[a]' can be shown like this:



The graph for an expression like 'a ; b' looks like:



Finally does the graph for an expression like 'a | b' looks like:



Finally it should be mentioned that the operator ';' is given higher precedence than '|' in the same way as for event expressions and that the unary operator '#' is given higher precedence than both of these binary operators.

3.3 Example (cont.)

At this step the temporal relations between the events are taken into account and the out-events are inserted. The number-sign '#' is used to indicate that a part of the sequence must be repeated a number of times. In this case two such loops are identified. At this step the loops are just identified, it is not decided how they should be terminated. That design decision is postponed to the next step. The result of this step looks like:

```
calc := start ;
        (
          mode ;
          (
            input ;
            out
          |
            error
          ) #
        |
          restart
        ) # .

out :=   output
       |
         output ;
         output ;
         output .
```

Notice the named sub-sequence "`out`". This part corresponds to the two modes in which the system is capable to operate in. Again it is important to notice that the designer only have identified the structure of the two modes. It is not yet visible how the choice must be taken between the two operation modes. The idea is that the designer can concentrate on getting the system structure right and postpone decisions about how the various choices must be made. Thus, at this step it is not decided how the still non-deterministic choice in "`out`" shall be taken.

4 Expanded Event Sequences

At this step of development the designer will for all choices decide whether they shall be made externally[5] or internally[6]. To stress that these decisions has been taken the syntax for iteration and alternative (which both introduces choices) from the previous step is changed. In addition the "maybe" operator is not present at this level. Notice that all of the operators at this step have both syntax and semantics corresponding to standard LOTOS except for the use of the two kinds of iteration (denoted by '`*`' and '`*>`') instead of (tail) recursion.

[5] By externally we mean that the first-coming event is chosen.

[6] By internally we mean that an internal calculation determines which event that is chosen.

4.1 Syntax

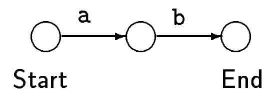
eess	=	*id* '`:=`' *ees* '`.`' [*eess*]
ees	=	[*ees* '`[>`'] *exc*
exc	=	[*exc* '`[]`'] *ge*
ge	=	[*guard* '`->`'] *pe*
pe	=	[*pe* '`;`'] *aol*
aol	=	[*aol* '`*>`'] *ae*
ae	=	'`STOP`' |
		id |
		'`(`' *ees* '`)`' |
		ae '`*`' |
		ev |
		'`I`' ['{' *opid* '}'] |
		'`C!`' '{' *comid* ['`.`' *opid*] '}' |
		'`C?`' '{' *comid* ['`.`' *opid*] '}'
guard	=	'`G`' ['{' ['`NOT`'] *opid* '}']
ev	=	*inev* |
		outev ['{' *opid* '}']
inev	=	*name*
outev	=	*name*
id	=	*name*
comid	=	*name*
opid	=	*name*

Here the following abbreviations has been used to fit the two column style:

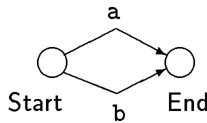
eess	=	*expanded-event-sequences*
ees	=	*expanded-event-sequence*
exc	=	*exception*
ge	=	*guarded-expression*
pe	=	*prefix-expression*
aol	=	*atomic-or-loop*
ae	=	*atomic-expression*
ev	=	*event*
inev	=	*input event*
outev	=	*output event*
id	=	*identifier*
comid	=	*communication identifier*

4.2 Semantics of operators

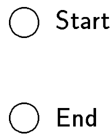
The semantics of the '`;`' is preserved from section 3.2 so the graph for it still looks like:



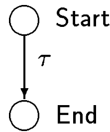
Similarly is the semantics for the alternative operator preserved so the graph for it looks like:



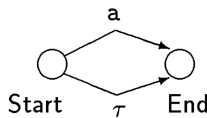
The standard event 'STOP' represents the completely inactive sequence. It cannot proceed with any event at all. Therefore 'STOP [] A' will correspond to 'A', while 'STOP ; B' will correspond to 'STOP'. This semantics can be illustrated by a graph where there are no connection between the start-node and the end-node:



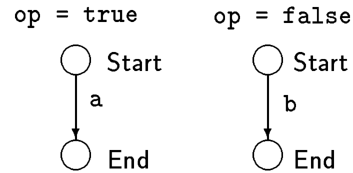
The standard event 'I' describes a spontaneous event by which the connected operation is evaluated. 'I' will typically be used to make a calculation possibly changing the state of that process as a kind of side-effect. Thus, if for example 'I' is used as an alternative to a 'real' in-event the choice between them are nondeterministic. This can be illustrated by two graphs. The graph for 'I' alone looks like:



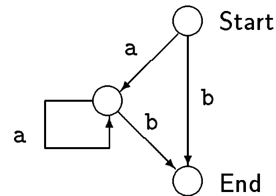
while the graph for an expression like 'a | I' looks like:



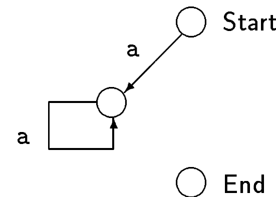
Guards are used to ensure that certain conditions are fulfilled, by means of truth-valued predicates which have no side-effects. Thus, it is used to indicate an internal choice where the result of the operation depends upon the state of the process. In this case there will be a 'G' on both sides of the alternative operator '[]' (possible 'G{OP}' and 'G{NOT OP}'). Then a side where 'OP' (or 'NOT OP') is true will be chosen and the false one will correspond to the standard event 'STOP' (the inactive sequence). Thus an expression using guards will have different graphs depending upon the value of the guards. In the case of the expression 'G{op} -> a [] G{NOT op} -> b' the graph looks like:



The loop exception operator '*>' corresponds to a normal loop which can be interrupted at the beginning of each turn in the loop. The operator itself contains no information about whether the corresponding choice is internally or externally. Like all other operators this one also has a compositional semantics, which can be illustrated by the following graph. Consider the expression a *> b:



The infinite loop operator '*' can only be interrupted by means of the exception operator '[>' (see below). The semantics of it can be illustrated by this graph, where there is no connection from the start node to the end node:



The semantics of the exception operator '[>' can be described by rewriting the expression in which it occurs. This can be illustrated by means of a small example: Let 'A := a ; b ; c' and let the process 'P' be described as: 'A [> d'. This expression can then be rewritten by allowing 'd' as an alternative at any place in 'A'. This looks like:

```
P := d
    []
     (a ; (d
         []
          (b ; (d
              []
               c))))
```

Thus, this exception operator may interrupt the 'normal' course of events at any point. The graphically representation of this operator is not itself compositional. Therefore we cannot give any general graph representing it. Informally the graphically representation can be explained by saying that there will be arrows going from all internal nodes from the left hand side expression to the end point of the graph where this arrow would be labelled with the right-hand-side. However we will rely on the above given rewriting rule to explain the semantics of it.

Finally a few words should be given about the use of inter-process communication even though that it is not illustrated by the example. Communication between processes inside the designed system is handled by a process which takes care of shared states independently of the user processes.

4.3 Example (cont.)

At this step internal calculation will become visible and the various choices will be determined. In order to stress that a choice has been taken, the syntax for alternatives is changed to '[]'. Similarly the syntax for iteration has been changed from '#' to '*'. The result of this step looks like:

```
calc := start;
          (I{Init};
            ( mode;
              (
                (
                  input;
                  out
                )*
              [>
                error_and_halt
              )
            )
              [> restart
          )*

out := G{SummaryMode} ->
           output{Summary}
       []
       G{NOT SummaryMode} ->
           output{FirstItem};
           output{SecondItem};
           output{ThirdItem}

error_and_halt :=
       error; STOP
```

Here it can be seen that apparently a lot of changes has been made. However after a closer investigation it is possible to recognize the structure.

Internally events corresponding to the evaluation of an operation are denoted by 'I{...}' where the dots are replaced by the name of the corresponding operation. Thus, this time the designer has decided that the system shall be initialized by means of the operation "Init".

Guards are denoted by 'G{...}' or 'G{NOT ...}' where the dots are replaced with a truth-valued predicate depending upon the internal state of the process. At this step the designer has decided that the choice in the sub-sequence "out" shall be taken by means of a predicate "SummaryMode" which evaluates on behalf of the value transferred by the "mode" event. Do also notice that the different instances of the out-event "output" all have an extra name attached to them. These names indicate the internal operations which must be evaluated to generate the desired output value.

Finally the designer has decided that the loops can be interrupted at any time. This corresponds to a decision saying that it is valid at any time in the outermost loop to receive the in-event "restart". Similarly can the innermost loop accept the in-event "error" even between the different out-events "output" which seems to be reasonable enough. Furthermore the designer has decided that it makes no sense to continue after the arrival of "error" before the arrival of the event "restart". The standard component 'STOP' is taken from LOTOS and corresponds to an inactive process. Thus, the designed system can only proceed by the in-event "restart".

5 System Expression

At this step of development the designer conclude how the complete system is composed of some of the expanded event sequences. Thereby a total description of the identified system is obtained.

5.1 Syntax

se	=	*sysid* ':=' *parex* '.'
parex	=	[*parex* '||'] *id*
sysid	=	*name*
id	=	*name*

Here the following abbreviations has been used to fit the two column style:

se	=	*system-expression*
parex	=	*parallel expression*
sysid	=	*system identifier*
id	=	*identifier*

5.2 Semantics of operators

Op	Meaning
`||`	A system expression like A `||` B means parallel composition of A and B. If A and B have common events A and B will synchronize on those events.

6 Concluding Remarks

Within the CEDER-Tools[7] parsers has been made for all these four notations. In addition a simulator for the system expression and the expanded event sequences is provided. Such support tools provides a framework which enables the acceptance of an event structuring approach like this. Furthermore the CEDER-Tools does also provide an automatic generator to a CEDER-LOTOS dialect. This can be used as a kind of documentation of the designed system where keywords and comments are added to increase the readability. Transformation to a subset of standard LOTOS is also under consideration as a possible extension in the future.

The differences between the CEDER-LOTOS dialect and standard LOTOS has been made to make it easier to get this approach diffused to practitioners. Recursion has been replaced with iteration because of more common usage in industrial applications. In addition recursion is in general badly understood by practitioners. The data part in standard LOTOS is described in an algebraic specification language called ACT-ONE (cf. [Ehrig&85]). Since algebraic specifications also are hard to understand for practitioners this part has been replaced with a subset of a model-oriented specification language called BSI/VDM SL[8] (cf. [Larsen&89] and [BSIVDM90]). Finally standard LOTOS has a notion of action prefix which in our opinion decreases the readability (at least for our purpose)[9]. Therefore we have not adopted this restriction. Our syntax also has restrictions compared to standard LOTOS. We have for example deliberately restricted the syntax so that dynamic creation of processes is not allowed (this is also abandoned by the UK Ministry of Defence in their draft standard about safety critical software [MoD0055]).

[7] A prototype of the CEDER-Tools is demonstrated at this workshop.

[8] BSI/VDM SL is currently being standardized.

[9] If the action prefix was kept an expression like '`(a | b) ; c`' would have to be written as '`a ; c | b ; c`.

Future work needs to be done in order to investigate how this scheme can be used in an iterative design process. Some kind of help must be provided to the designer to keep all four notations consistent with each other.

As a conclusion we can say that we believe that this stepwise approach with advantage can be applied in industry. Furthermore the presence of automatic tool support for this methodology will ensure that it can be applied efficiently by practitioners. The underlying principle behind the CEDER method in general is to have a firm theoretical basis. Such a principle is normally a barrier for industrial usage but because of the special attention paid to the practitioners we believe that they will be able to comprehend and use this method.

References

[BSIVDM90] *VDM Specification Language – Proto-Standard.* Technical Report, British Standards Institution, February 1990. BSI IST/5/50.

[Ehrig&85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1.* Springer-Verlag, 1985.

[Elmstrøm90] Rene Elmstrøm. Design of Programs for Embedded Systems using CEDER. In *2nd Nordic Workshop on Program Correctness*, October 1990. 8 pages. An earlier version of this article was published at the Euromicro Real-Time Workshop 1990.

[ISO8807] E. Brinksma. *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* International Organization for Standardization, 1988.

[Larsen&89] P.G. Larsen, M.M. Arentoft, B.Q. Monahan and S. Bear. Towards a Formal Semantics of The BSI/VDM Specification Language. In *Information processing 89*, North-Holland, August 1989.

[Løvengreen&90] H.H. Løvengreen, A.P. Ravn and H. Rischel. Design of embedded, real-time systems: Developing a method for practical software engineering. In *COMP EURO 90*, May 1990.

[Milner89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[MoD0055] *Defence Standard for military safety-critical software – 00-55.* The UK Ministry of Defence, 1989. Draft.