Design of Embedded Real-time Systems: Developing a Method for Practical Software Engineering

Løvengreen, Hans Henrik; Ravn, Anders P.; Rischel, Hans

Published in:
IEEE International Conference on Computer Systems and Software Engineering

Link to article, DOI:
10.1109/CMPEUR.1990.113649

Publication date:
1990

Document Version
Publisher's PDF, also known as Version of record

Design of embedded, real-time systems: Developing a method for practical software engineering*

Hans Henrik Løvengreen, Anders P. Ravn and Hans Rischel

Department of Computer Science,
Technical University of Denmark,
DK-2800 Lyngby, Denmark

Abstract

The methodological issues and practical problems in development and industrial use of a theory-based design method for embedded, real-time systems are discussed. The method has been used for several years in a number of smaller industries, which develop both electronics and software for a professional market. The design is expressed in a notation for communicating sequential processes; while data types and operations are expressed in a notation built on mathematical set theory. Our main contribution has been to delineate an order in which to use the notations, a technique for deriving states and operations, and systematic checks of a design with respect to system requirements.

1 Introduction

During the last decade theories of communicating processes have matured [6,13], and in the same period mathematically based notations for describing data and operations on data [2,10,19] have been used, though not widely accepted in software development. As researchers at a technical university we believe that use of these and presumably other theories is an important factor in making software creation a professional engineering discipline.

Our beliefs were put to the test, when we in 1984 were asked by a company manufacturing high-quality electronic instruments whether we had any good ideas about "design tools" for embedded software systems. Their development teams were worried because 70 % of their development costs went to software development, an area without professional standards. Given this challenge, we produced a one week course for engineers. The course was developed for an independent institution — the Danish Institute of Technology and we benefited from their experience with the field. The course was tuned over a period of one year by giving it to development engineers, and then turned over to the staff of the Institute of Technology. The material [18] is used in our own teaching. Although it is not a standard in any company we know of, it is used by a growing number of development teams. In early 1988 it was proposed to build a tool to support the method, and 8 companies agreed to make an initial payment without any guarantee for delivery. This tool is now developed by a consortium, and is to be completed this year.

*This work has been partially supported by the Danish National Agency of Technology. Some of the results were published in [16]

The main part of this paper concerns *methodology,* i.e. the issues in making a method, and our solutions to the problems of integrating functional and temporal constraints in a design, and how a design can be checked against requirements. These are contained in a section which outlines our methodological decisions and our theory base, and a section with a simplified case study to illustrate the techniques of the method. A final section discusses methodology, techniques and theory in a closer comparison with Jackson System Development (JSD) [8], which was an initial inspiration for us.

2 Methodology

During a design activity a number of *decisions* are made and *written down.* The design is the produced document(s). The *design method* we want should provide:

- A defined sequence of *stages.* In each stage the designer decides on a solution to a limited aspect of the problem or evaluate a solution.
- *Notations* to write down decisions.
- *Checks*, i.e. means of checking functionality and performance of a design against requirements.

2.1 Stages

What to consider important aspects at a certain point of the design activity is not a question of notation or theory. Our delimitation of stages is based on *pragmatics* — good practice — within the application area. Fortunately we had a well-defined application area and could build on traditions, parts of which have been formulated in [5] and more specifically for the particular application area in [3]. The outcome was the following sequence of stages:

1. **Interface Definition** defining the events which form the interface to the system.
2. **Event Structuring** defining the behaviour of the system in terms of a partial, temporal ordering of events.
3. **Program Structuring** defining the relationship of output values to input values.
4. **Functionality Check** check of interface to entities.
5. **Timing Analysis** estimating performance.

CH2867-0/90/0000/0385$01.00 © 1990 IEEE

6. **Functional Decomposition** arranging functions in modules.

The selected sequence of stages represent three major decisions:

1. To assume that the *requirements* are known by the designer. At the time we found no satisfactory theory on which to base an analysis of the requirements in order to produce a rigorous *specification.* The solution we offer is to proceed with the design and then check it against the requirements at later stages.

2. To suppress knowledge of functional relationships between event values until the (temporal) *behaviour* has been defined. This reflects that in the application domain, synchronization of events is a more intricate problem than design of the sequential operations on state variables.

3. To postpone performance considerations till the functionality is believed to be correct. Furthermore only to add input-output buffers or parallel algorithms if analysis indicates that it is required. By this line of action a considerable amount of unnecessary, complex mechanisms in the resulting design is avoided.

2.2 Notation

Notation is needed to write down the design. Our main criteria for the *syntax* of the notation is *brevity.* This made us choose linear, textual notations as the designers main working tool. Diagrams to summarize a design can be derived from the text. The rationale for brevity is that a design document has to be rewritten many times and is read many more times. In order to make the method useful for real problems our goal was that the method should produce documents which are an order of magnitude smaller than the resulting high level programs. From completed projects we can estimate that we have achieved a factor of 6 – which we find acceptable.

Another concern is that a notation should have a *clear meaning* in order to understand the design and its consequences, and be able to manipulate the design when checking it. We have tried to insist that the *semantics* of our notations should be rooted in accepted computer science theory. Thus we rely on the work of [11,4]. In analysis of time performance of concurrent processes, we did not find a satisfactory solution in the available theory. Here we had to rely on informal definitions. One point worth noticing is that the method as presented to the user does not include formal semantics for the notations. It is a responsibility of the producers of notations and not a concern of the users.

2.3 Checking functionality and performance

A good design team is very intent on getting things right; but in a larger design misconceptions of informal requirements do occur. In the absence of a formal specification, we decided that checks on a design should be based on systematic manipulation of the formulae of the design document. The idea being that a software engineer, like any other engineer, checks a design by computing certain consequences of the design.

3 Application of the method

In the following we describe the method in more detail to illustrate techniques for resolving design decisions within each stage. The description is based on a small, simplified case. The requirements for the case system are:

> An embedded system used in an aviation control center (ACC) should have the following functions:
>
> A radar is connected to the computer system. The radar produces coordinates for observed objects (*OO*-coordinates), which are collected by the system and classified as positions for new or already-known flying objects (*FO*'s). The *FO*'s together with identifying codes (*FOid*'s) are shown on a graphic display. The operator may enter informations for *FO*'s and the information is then shown on the display alongside the *FO*'s. The system should handle up to 500 *FO*'s at the same time, refreshing the display two times a second.

3.1 Interface Definition

The interface of the computer system to its *environment,* in which it is embedded, is characterized by the smallest units of atomic interaction, called *events.* Events are described by named *event (classes),* and are classified as *input* or *output* according to the flow of data. The *domain* (i.e. contents or abstract structure) of the *data* of each event is defined by a *formula.* Events define the observable interface to named *entities* in the environment. There is no concept of internal events. Internal events are avoided because they would reflect a premature functional decomposition. Premature, because the need for certain functions is only known when input-output relations are examined in the Program Structuring stage. For the ACC-system we arrive at the list shown on figure 1.

Domains are defined by formulas using basic predefined domains (BOOLEAN, TOKEN, NUMBER, VOID) and *rules* for *building composite domains* (cartesian product, sequences, sets, etc.). Notations and concepts are borrowed from VDM [2], [10].

3.2 Event Structuring

The temporal behaviour of the system is described by the partial ordering of events. Ordered events or (event alternatives) constitute *event sequences.* Each event sequence is described by an *event expression,* using a regular-expression-like notation (sequencing ';', choice '|', repetition '*'). The *maximum* event sequences are written down and given names.

For the ACC-system, we may arrive at the event sequences

```
collection  =  OO*
displaying  =  ( Halfsec ; FOout * ) *
operation   =  FOinfo *
```

In the check stage we shall discuss whether these in fact reflect the requirements.

Input event	Domain	Comment
OO	Airpos Airpos = SEQ OF NUMBER	Observed object with position
FOinfo	FOid × Info FOid = TOKEN Info = SET OF TOKEN	Info from operator
Halfsec	VOID	Half-second signal
Output event	**Domain**	**Comment**
FOout	FOdisp FOdisp = Dispos × FOid × Info Dispos = NUMBER × NUMBER	FO to display

Entities: Radar, Operator, Display, Clock

Figure 1: Interface Definition

In the next stage (Program Structuring) each event sequence determines a sequential *process*. We use a set of maximal sequences in order to minimize dynamic synchronization among the resulting set of parallel processes. Minimal synchronization is attractive because it reduces the potential for deadlock, and makes analysis of timing less dependent on the scheduler. Besides, it is also most efficient for a multiprogramming implementation. The set of parallel processes reflect requirements for independent execution of event sequences – a parallelism which is better kept, because it can only be eliminated by use of interleaving choices, which usually leads to a combinatorial explosion of the control structure of the program.

The notation does not contain a parallel operator, because nested parallelism would hinder the checks of functionality employed. This does, hovewer, not seem to be any practical restriction on describing the required behaviour for embedded systems.

3.3 Program Structuring

Event expressions are translated to an *abstract program* for a *process* following the well-known construction of a recognizing state machine for a regular grammar (or the JSP/JSD translation from syntactical structure to program structure, cf. [7], [8]). Given the control structure and the input-output operations, the required *computations* and *internal states* are derived and added to the abstract program by the following technique:

Each *output value* and *internal choice* in the program is defined as the result of a named *function*. The input parameters to this function can be: The latest input value for an event – or a state, representing accumulated input values (when required by the computation).

If a *state* is introduced, functions to accumulate input values are named and introduced in the abstract program immediately after the input operation. The domain of the function is the cartesian product of the state type and the event type, and the range is the state type.

When the functions operating on a given state are defined on events of a single process, the state is *local*, otherwise it is *shared*. Each shared state is placed in a *state handler* which is independent of the processes. The state handlers make the internal synchronization and the use of shared data among the processes explicit.

The abstract program is completed by introducing *device handlers* for each entity.

In the ACC-system there is only one output event FOout in the displaying process. It requires data from processes collection and operation. Consequently, a state handler FOtable is introduced and the abstract programs are extended with proper communication with the state handler (Device handlers have been left out, they are similar in structure to a state handler, but have no state).

```
PROCESS collection ;
   LOOP
      radar ? oo;
      FOtable ! newobs(oo)
   END LOOP
END PROCESS;

PROCESS operation;
   LOOP
      operator ? foinfo;
      FOtable ! newinfo(foinfo)
   END LOOP
END PROCESS;

PROCESS displaying;
   LOOP
      clock ? Halfsec;
      FOtable ! initscan;
      LOOP WHEN FOtable ? done EXIT;
         FOtable ? getfo(fodsp);
         display ! FOout(fodsp);
         FOtable ! nextfo
      END LOOP
   END LOOP
END PROCESS;
```

```
STATE HANDLER FOtable;
   TYPE Table;
   STATE t: Table;
   OPERATIONS
      newobs: Table × Airpos → Table
      newinfo: Table × FOid × Info → Table
      initscan: Table → Table
      getfo:   Table → FOdisp
      nextfo:  Table → Table
      done:    Table → BOOLEAN
END HANDLER;
```

3.4 Functionality Check

We focus on interfaces to entities in the environment, which will allow us to check:

1. The protocol used for this entity,
2. Sets of events triggering outputs to the entity, and
3. Events used in deriving an output value

For each entity a restricted event sequence is derived by removing (hiding) events from other entities. This is the protocol from the system to the entity. For the ACC, we would have, e.g.

$$(\mathsf{FOout}^*)^*$$

for the display – which is the correct protocol (= FOout*).

For each output event the set of event sequences containing this event is considered, and all prefixes to the event are calculated. The set of *input events* in any prefix is called a *trigger set* of events for the output event. For the ACC we have

$$\mathsf{Halfsec};\ \mathsf{FOout} \parallel \mathsf{FOout};\ \mathsf{FOout}$$

i.e. FOout has the trigger set {Halfsec}, which can be compared with the set of events used in deriving a value:

$$\{\mathsf{OO},\ \mathsf{FOinfo}\}$$

The operator might find it annoying that FOinfo does not imply FOout. If this is required, the design would have to be changed, e.g. the sequences displaying and operation merged to:

$$((\mathsf{Halfsec} \parallel \mathsf{FOinfo});\ \mathsf{FOout}^*)^*$$

If a design contains more than one event that occur in different event sequences the design may contain a potential *deadlock*. We recommend that a proof of absence of deadlocks uses a ordering of such events as a basis. This can possibly lead to design changes.

3.5 Timing Analysis

Real time constraints are found in most embedded systems. They are checked by estimating the time taken by certain sequences ("cycles") of the abstract programs. The calculations are illustrated in the following.

These estimates are calculated under the assumption that events occur periodically. The execution time for an input operation, e.g. the OO-event is then estimated by: $2 + \mathcal{V}(OO)$, where the 2 time-units is an estimate of internal processing time and $\mathcal{V}(\mathsf{OO})$ is the (external) waiting time. For access to a handler, the delay caused by other processes has to be considered. Assuming queuing discipline for accesses, we can estimate the time for e.g. access to FOtable as: $2 * N$, where $N = 3$ is the number of processes having access.

Calculation would then give a cycle time for the collector of: $2 + \mathcal{V}(\mathsf{OO}) + 6$. The requirements state 500/10 OOs per second. Which (assuming a time unit of 1 millisec.) would leave $50 - 8$ time units. The collector is safe, provided that the interarrival time is greater than 8. If this cannot be guaranteed an input-buffer would alleviate the problem.

A similar calculation for displaying would (using 500 as an upper limit for the number of FOout's) give a cycle time $\mathcal{V}(\mathsf{Halfsec}) + 500 * \mathcal{V}(\mathsf{FOout}) + 500 * 2 + 8$. Even if $\mathcal{V}(\mathsf{FOout}) = 0$ this figure exceeds by far 500, so the cycle would not be completed before the next Halfsec event arrives. We conclude that the displaying process is going to be too slow because of an excessive number of FOout operations.

These calculations are engineering calculations, and are not supported by theory. Some initial investigations suggest, however, that they can be supported by the model of timed CSP developed by Reed and Roscoe [17].

3.6 Functional Decomposition

This stage uses well-known, informal techniques for functional and modular decomposition (cf. [15]) to refine operations and states of the abstract program.

4 Discussion

Those experiences with the method which we consider of general interest for developers of methods and theory, and for software developers in general are summarized in the following. In order to make certain points, we shall compare certain stages and techniques with similar stages in JSD, which we consider to be one of the better of the widely used commercial methods.

4.1 Notations

A positive reaction from the users of the method has been that the documentation is concise. To quantify that statement, we give the following figures for the documentation of the software for an instrumentation system that monitors and analyses vibrations in large machinery:

Stage:	Pages:
Interface definition	9
Event sequences	7
Abstract programs	38
Total	56

The program source texts are approximately *350 pages*. The manager of the team had previously completed a project of similar size. In that project they used diagrams to document their data and function design. This design document amounted to well over 400 pages, and of course

it was not maintained during development (The method used was *not* JSD).

Our conclusion is that graphical notations might be easy to learn, but their use places a superfluous burden on a professional development team.

4.2 Theory

Apart from our belief that an engineering discipline should be based on suitable mathematical theories, a theory base have practical implications:

- Use of data types for events and in signatures for operations eliminate introduction of vaguely definied 'actions'. In JSD and similar methods 'actions' and 'operations' are uninterpreted symbols – inexperienced or overworked developers may be tempted to hide problems in perhaps uninterpretable symbols.
- Use of process algebra allows systematic checks of a design by calculations.

There are two points where further development of theory is urgently needed:

- Real time models. There is considerable activity in this area, but the models and theories have to be based on realistic assumptions about the problem domain. We can recommend the discussion in section III of [14].
- A basis for specification that integrates behavioural specification, state value specification of operations, and timing specifications. A formal specification language would allow the checks on requirements to be formalized.

4.3 Interface definition

JSD has starting point in *real-world entities* and analyzes and model their relevant *actions*, while we start the design process from the *interface* to the system and defines it in terms of *events* and *data-domains*. We certainly agree on the relevance of modelling in *requirements* engineering, but we do *not* believe that it is possible to obtain useful generality for a *design* by modelling "the world" – at least not in the considered problem area. The modelling gives the conceptual framework, but the functional requirements defines the part thereof to be implemented in the program.

A remark which we have heard from several users, is that it is hard to get started. It is a nontrivial task to select events at the proper level of abstraction. In one case, which lead to some dismay, the designers started out very systematically with wire-level signals, and they were unable to get meaningful event sequences out of that. Our diagnosis was that the group had started from the wrong documents. They had used the specification of prototype hardware in place of the functional specification (the requirements) for the product.

4.4 Event sequences

The technique of using maximal event sequences leads to robust and efficient implementations of a design. Getting nice event sequences takes some time; but there has not been complaints about that. Most engineers like to discuss the architecture of a system in terms of its event structures.

In contrast JSD first derives a set of processes for the behaviour of real-world entities – in a later step transformations are performed in order to get a sequential program. The behaviour of the developed system versus the requirements plays a secondary role in the JSD development process.

4.5 Derivation of computations

Deriving computations from the requirements for output values (and for internal choices) is also accepted as a sound practice by most designers. Some, however, had to unlearn approaches learnt in database design, where the aim is to provide "useful" general functions. We take the stand that any wanted generality is part of the requirements. The design is not a place to introduce unstated requirements.

4.6 Shared states

A common problem in existing real-time software is missing discipline in the access to shared data. The symptoms are lack of data integrity (no control of critical regions) or deadlock (caused by waits for access). State handlers are a clean way of obtaining integrity of shared data and at the same time identifying the entire inter-process communication.

5 Future developments

We are presently advising a consortium which produces a CASE-tool for the method. In order to have a general framework in which to define notations both syntactically and semantically, we have advised them to use the LOTOS [12] notation with two exceptions:

1. Iteration ("*") is used instead of tail recursion. This leads to introduction of notation for a state of a process.
2. BSI/VDM domains[11] are used instead of algebraic specifications.

These changes make the notation considerable more compact. Furthermore we believe, though the details have not yet been worked out, that these constructs can be expanded into proper LOTOS.

The motives for using LOTOS as a base line are:

1. It assures us that we have consistent syntax and semantics
2. It allows this CASE-tool to use LOTOS-tools, e.g. simulators.

We are also enganged in the ProCoS project (Provably Correct Systems, Esprit BRA project 3104, cf. [1]). The aim of ProCoS is to contribute to the science and engineering

of constructing mathematically provably correct systems – in particular for safety critical applications. The current activities in ProCoS form in several ways a continuation (on a firm theory basis) of the work reported in this paper.

References

[1] D. Bjørner, et.al.: A ProCoS Project Description, ESPRIT BRA 3104, *BULLETIN EATCS* 39, 60-73, 1989

[2] D. Bjørner, C.B. Jones: *Formal Specification and Software Development*, Prentice-Hall, 1982

[3] O. Caprani, S. Lauesen, U. Ougaard: Design Principles for Dedicated Data Collecting Programs, *Large Scale Integration, Euromicro Symposium*, North-Holland 1978

[4] C. George, K. Havelund, M. Nielsen, K.R. Wagner: The RAISE Language, Method and Tools, in: *VDM '88 The way Ahead*, LNCS 328, 376-404, 1988

[5] P. Brinch Hansen: *The Architecture of Concurrent Programs*, Prentice-Hall, 1977

[6] C.A.R. Hoare: *Communicating Sequential Processes*, Prentice-Hall, 1985

[7] M. Jackson: *Principles of Program Design*, Academic Press, 1975

[8] M. Jackson: *System Development*, Prentice-Hall, 1983

[9] I. Jacobson: FDL: A Language for Designing Large Real Time Systems, *Information Processing 86*, 463-468, (H.-J. Kugler ed.), North-Holland, 1986.

[10] C.B. Jones: *Systematic Software Development using VDM*, Prentice-Hall, 1986

[11] P. Gorm Larsen: The Mathematical Semantics of the BSI/VDM Specification Language, *Information Processing 89*, 95-100, (G.X. Ritter ed.), North-Holland, 1989

[12] *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, (Ed Brinksma ed.), ISO 8807, 1988

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

[14] F. Jahanian, A.K-L. Mok: Safety Analysis of Timing Properties in Real-Time Systems, *IEEE Trans. SE*, Vol. 9, (1986)

[15] D.L. Parnas, P. C. Clement, D. M. Weiss: The Modular Structure of Complex Systems, *Proceedings of the 7th International Conference on Software Engineering*, 408-417, 1984

[16] A. P. Ravn, Hans Rischel, H. H. Løvengreen: A Design Method for Embedded Software Systems, *BIT* 28, 427-438, 1988

[17] G.M. Reed, A.W. Roscoe: Metric spaces as models for real-time concurrency, in: *Mathematical Foundations of Programming*, LNCS 298

[18] H. Rischel, B. G. Mortensen, A. P. Ravn: *Konstruktion af Formålsbundne Systemer*, Teknisk Forlag 1987 (in Danish)

[19] M. Spivey, *The Z Notation: A reference Manual*, Prentice-Hall, 1989

[20] P. Zave, W. Schell: Salient Features of an Executable Specification Language and Its Environment, *IEEE Trans. SE*-12, 312-325, 1986