Specification of Embedded, Real-Time Systems*

Jens U. Skakkebæk, Anders P. Ravn, Hans Rischel & Zhou Chaochen
Department of Computer Science
Technical University of Denmark, Bldg. 344
DK 2800 Lyngby, Denmark

Abstract

An approach to requirements specification and subsequent verification of designs for embedded, real-time systems is presented. A system is given by a conventional mathematical model for a dynamic system, where application specific state variables denote total functions of time. Specifications are formulas in a real-time, interval temporal logic, where atomic predicates define durations of states. Requirements are specified by a conjunction of formulas, which reflect safety and functionality constraints on the total system. A design specifies the behaviour of components and the conjunction of component specifications can be shown to imply the requirements. Designs can be refined in a similar fashion.

1 Introduction

Requirements engineering [4] for software that controls physical systems must investigate safety and functional requirements for the system as a whole. Requirements typically delimit expected behaviours over time for a combination of given physical processes with planned sensors, actuators and programmed computers. Requirements engineering is ideally completed with precise specifications for components and the design binding them together. This paper illustrates an approach to requirements engineering which has evolved during our work with case studies within the Provably Correct Systems (ProCoS) project [1, 13].

The approach uses a conventional time-domain model of mathematical systems theory or control engineering [12] and develops predicates describing properties of a total system in three steps:

1. An application domain system model of the equipment and its intended environment of use is defined. This defines the overall states of the system as functions of time. Requirements constrain this model.

2. A control model extends the system model with an explicit control strategy. The strategy is verified to imply the requirements under a set of assumptions about the intended environment of use.

3. A design model for a distributed system defines separate specifications for a set of interface units and programs. Interface units relate system states to event values under certain timing and approximation constraints. Programs implement the control strategy by controlling timing and order of events, and by computing relevant event values.

The approach is based on refinement of models. Each refinement removes some freedom (choice, non-determinism) by adding further constraints. At each stage the resulting model is verified to be contained in (or imply) the model of the previous stage.

The following sections introduce the specification language, and then discuss the system and control models with associated refinements in more detail, using a simple Railway Level Crossing as a running example. The approach has also been used on a simple Auto Pilot [14] (example due to Boyer and Moore) and on a Gas Burner [11].

2 Specification language

A system is described by a collection of state variables which are functions of Time, modelled by the real numbers. Properties of systems are expressed by constraints on the state variables. We wish to express requirements and design without explicit mentioning of time instants, and introduce a notation which is a real-time, interval logic [9] based on state durations: The Duration Calculus [2].

2.1 Syntax

We assume names for state variables \( X, Y, \ldots \) together with their value domains \( \text{Type}_X, \text{Type}_Y, \ldots \) given by declarations in a suitable specification language, here VDM (cf. [5]). The language should comprise names for constants and operators. The type \( \text{R} \) of real numbers should be available with the usual operators, and so should the type \( \text{Bool} \) of Boolean values with the usual propositional operators. The Boolean constants are denoted \( \text{tt} \) and \( \text{ff} \) (the names \( \text{true} \) and \( \text{false} \) are reserved for duration formulas). We use lower case names \( a, b, \ldots \) to denote constants or static variables of any type in the language. Static variables denote time-independent entities.
State expressions and state assertions

A state expression is generated by a constant, a static variable, a state variable or any (type correct) expression \( op(s_1, \ldots, s_n) \) formed from an operator symbol \( op \) and state expressions \( s_1, \ldots, s_n \).

A state assertion is a state expression with type \( \text{Bool} \).

Durations and duration terms

For any state assertion \( P \), \( \{ P \} \) is a duration. A duration term (of type \( \text{R} \)) is a duration, a real constant, a real static variable or the term \( op(r_1, \ldots, r_n) \), where \( op \) is an \( n \)-ary operator symbol of type \( \text{R} \) and \( r_1, \ldots, r_n \) are duration terms.

The symbol \( t \) is used as an abbreviation for the duration term \( \int t \).

Duration formulas

If \( A \) is any \( n \)-ary predicate symbol on \( \text{R} \) and \( r_1, \ldots, r_n \) are duration terms, then \( A(r_1, \ldots, r_n) \) is an atomic duration formula. Atomic duration formulas, the symbols true and false, \( (\neg D_1) \), \( (D_1 \land D_2) \), \( (D_1 \lor D_2) \), and \( (\forall x)D_1 \), where \( x \) is a static variable and \( D_1, D_2 \) are duration formulas, are duration formulas of type \( \text{R} \). We use standard abbreviation \( \land \), \( \lor \), \( \exists \) for both state assertions and duration formulas, and we introduce abbreviations for commonly used duration formulas:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Formula</th>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \top )</td>
<td>( { P } )</td>
<td>( { P } ) is a duration</td>
</tr>
<tr>
<td>( \bigcirc D )</td>
<td>( \bigcirc D ) is true</td>
<td></td>
</tr>
<tr>
<td>( D_1 \lor D_2 )</td>
<td>( D_1 \lor D_2 ) is true</td>
<td></td>
</tr>
<tr>
<td>( \neg (\bigcirc D) )</td>
<td>( \neg (\bigcirc D) ) is false</td>
<td></td>
</tr>
</tbody>
</table>

The following precedence rules are used

1. \( \neg, \land, \lor \)
2. \( \bigcirc, \top, \bigwedge \)
3. \( \Rightarrow \)

2.2 Semantics

A (particular) behaviour \( B \) of a system assigns a function \( [0, \infty) \rightarrow \text{Type}_X \) to each state variable \( X \), and selects a value \( \forall x(z) \) for each static variable \( z \). Each state expression then denotes a function obtained by evaluating the expression for each point of time. For a state assertion \( P \) it seems reasonable to demand finite variability: For each behaviour, any observation interval can be divided into finitely many subintervals with \( P \) constant on each (open) subinterval.

An observation interval is a closed and bounded interval \([b, e]\) \( \subset [0, \infty) \). For a given interval the duration \( \int P \) of a state assertion \( P \) denotes the real number \( \int_b^e \{ P(t) = tt \text{ then } 1 \text{ else } 0 \} \text{dt} \) which is the measure of the set of points where \( P \) has value \( tt \). For any behaviour \( B \) and interval \([b, e]\) duration terms denote real values and atomic duration formulas denote Boolean values.

The values of composite duration formulas are obtained by the usual interpretation of the logical operators and quantification, cf. e.g. [6]. The value of a "chop" formula \( D_1 \lor D_2 \) is \( tt \) if and only if the interval \([b, e]\) can be divided into \([b, m]\) and \([m, e]\) such that \( D_1 \) is \( tt \) on \([b, m]\) and \( D_2 \) is \( tt \) on \([m, e]\).

A duration formula \( D \) holds on the interval \([b, e]\) for the behaviour \( B \) just when it holds for every behaviour \( B \) and every interval \([b, e]\). It is sufficient for a formula to be valid, that it holds from start for every behaviour \( B \).

2.3 Specifications and refinement

A specification for a system is a duration formula \( D \). A behaviour \( B \) satisfies the specification if \( D \) holds from start for \( B \).

For specifications \( D_1 \) and \( D_2 \) we say, that \( D_2 \) is a refinement of \( D_1 \) if for any behaviour satisfying \( D_2 \) also satisfy \( D_1 \). It follows, that \( D_2 \) is a refinement of \( D_1 \) if the duration formula \( \forall D_2 \Rightarrow D_1 \) is valid.

2.4 Proof system, Verification

It is almost certain, that the general Duration Calculus is undecidable, and hence we cannot expect to find a complete set of axioms and proof rules. The proofs in this paper may, however, be based on the set of axioms and proof rules given below. In the following, \( P \) denotes a state assertion, \( r \) a non-negative real number and \( D \) denotes a formula in the Duration Calculus.

**Axiom 1** \( \int fP = 0 \)

**Axiom 2** \( \int P \geq 0 \)

**Axiom 3** \( \int P_1 + \int P_2 = \int (P_1 \lor P_2) + \int (P_1 \land P_2) \)

**Axiom 4** \( (\int P = r_1) ; (\int P = r_2) \iff \int P = r_1 + r_2 \)

**Axiom 5** If \( P_1 \iff P_2 \) is a valid state assertion then \( \int P_1 = \int P_2 \) is an axiom.

The following induction rule is sound due to the finite variability of states.

**Induction Rule** If \( D(\int \{ P \}) \) is provable, and \( D(X \lor X) \lor \{ P \}) \lor \{ X ; \neg P \}) \) is provable from \( D(X) \), then \( D(\text{true}) \) has been proved.

It has a dual, backward induction rule.

3 System model

The first step in formalising requirements to a system is to construct a system model. Requirements are constraints on this model. This section builds a system model for our running example: A railway level crossing.
3.1 Informal description
The following informal description is based on the description found in [10] and on discussions with engineers from DSB (Danish National Railways):

The railway level crossing is a crossing between a single track railway and a road. For simplicity it is assumed that all trains passing the crossing will travel in the same direction. The crossing is shown in figure 1.

Figure 1: Railway Crossing

Road traffic is controlled by a gate at each side of the railway. The gates close only when road traffic is not stuck in the crossing.

Train traffic is controlled by a signal on the side of the tracks along which the trains approach. The signal indicates: “stop” or “go” for oncoming trains.

Sensors keep track of trains in the system. A sensor is placed in a reasonable distance from the signal such that a train will reach the first sensor point before it reaches the signal. A train enters the system, whenever it is determined by this sensor. A train has left the crossing, whenever the sensors determine that the rear end of the train has passed the crossing.

When a train approaches, the gates are closed, provided road traffic is not stuck in the crossing. The signal is only set to “go” after the gates have closed.

When no trains are approaching or passing, the signal must be set to “stop” and the gates opened.

The main objective of the system controlling the gates and the signal is to ensure safety: The system must never allow train and road traffic to pass the crossing at the same time.

Furthermore, the system must ensure that both road traffic and trains are able to pass the crossing within some reasonable time.

The device that controls the system is the Railway Crossing Control System (RCS). The RCS gets input data from the train sensors and the gate sensors, and sends commands to the signal and the gates.

3.2 State variables
The state models a signal, gates, road traffic and trains.

The Signal can be “stop” (ff) or “go” (tt)

Signal : Bool

The gates can be open, closed, opening or closing

Gates : {open, closed, opening, closing}

The road traffic can be stopped at the crossing, be stuck in the crossing or be free to cross

Traffic : {stopped, stuck, free}

Trains are either approaching or passing

Appr : N-set
Pass : N-set

where a train, identified by a unique, natural number i, is approaching (i ∈ Appr) if some part of the train is between the sensor and the signal, and passing (i ∈ Pass) if some part of the train is between the signal and the end of the crossing. Trains are active if either approaching or passing (or both). We define three state assertions expressing the state of the trains

Passing © Pass ≠ ∅
Approaching © Appr ≠ ∅
Active © Act ≠ ∅

where Act = Appr U Pass

3.3 Requirements
The requirements concern safety and function

Req © □(SafReq A FunReq i) i=1

Safety requirement

If the gates are not closed or road traffic is stuck in the crossing, the trains must not pass

SafReq © [(Gates ≠ closed) V (Traffic = stuck)] → [¬Passing]

Functional requirements

1. The road traffic should maximally be held back for a predefined period of time, Tstpped

FunReq1 © [Traffic = stopped] ⇒ ℓ ≤ Tstpped

2. When all trains have left the railway crossing, the gates must be open for at least time, Topen

FunReq2 © [Active]; [¬Active]; [Active] ⇒ f(Gates = open) > Topen

3. Provided the road traffic is not stuck, a single train must be able to pass within time, Tactive

FunReq3 © [i ∈ Active ∧ (Traffic ≠ stuck)] ⇒ ℓ ≤ Tactive
4 Control model

We now turn to selecting the assumptions and a control strategy, such that the controlled system can be proved to satisfy the requirements under the given assumptions.

4.1 Assumptions

The assumptions constrain the system environment. In the case of the railway level crossing, it is necessary to constrain the behaviour of both the road traffic and the trains, as well as the devices.

\[ \text{ASM} \equiv \square(\bigwedge_{i=1}^{2} \text{RTasm } i \land \bigwedge_{i=1}^{5} \text{Tasm } i \land \bigwedge_{i=1}^{3} \text{Dasm } i) \]

Road traffic assumptions

1. When running freely, the road traffic might eventually either be stopped properly by the gates or become stuck in the crossing. Stopped and stuck road traffic might in turn become free again.

\[ \text{RTasm1} \equiv \]

\[ (\text{Traffic } = \text{stopped}) \rightarrow (\text{Traffic } = \text{free}) \]

\[ \land (\text{Traffic } = \text{free}) \rightarrow \]

\[ (\text{Traffic } = \text{stopped}) \lor (\text{Traffic } = \text{stuck}) \]

\[ \land (\text{Traffic } = \text{stuck}) \rightarrow (\text{Traffic } = \text{free}) \]

2. Road traffic is stopped, if and only if the gates are not open.

\[ \text{RTasm2} \equiv \]

\[ \text{Traffic } = \text{stopped} \Leftrightarrow \text{Gates } \neq \text{open} \]

Train assumptions

1. Trains must only pass if the signal is “go”

\[ \text{Tasm1} \equiv \text{[Passing]} \Rightarrow \text{[Signal]} \]

2. An active train travels in one direction only (approaches initially and passes finally).

\[ \text{Tasm2} \equiv \]

\[ (i \notin \text{Act} \rightarrow (i \in \text{Appr} \land i \notin \text{Pass})) \]

\[ \land (i \in \text{Appr} \land i \notin \text{Pass} \rightarrow (i \in \text{Pass})) \]

\[ \land (i \in \text{Pass} \rightarrow (i \notin \text{Act})) \]

3. The last train in a series of trains passes the crossing before leaving the crossing.

\[ \text{Tasm3} \equiv \]

\[ (\neg \text{Active} \rightarrow (\text{Approaching} \land \neg \text{Passing})) \]

\[ \land (\text{Approaching} \land \neg \text{Passing} \rightarrow \text{Passing}) \]

\[ \land (\text{Passing} \rightarrow (\neg \text{Active} \land \neg \text{Active})) \]

\[ \land (\text{Active} \rightarrow \text{Passing}) \]

4. The trains do not linger if the signal is “go”

\[ \text{Tasm4} \equiv \text{[Signal} \land \text{Active}] \Rightarrow t \leq \text{Tsched} \]

5. The railway tracks are not overloaded with trains

\[ \text{Tasm5} \equiv \left[ \text{Active} \right] \land \left[ \neg \text{Active} \right] \land \left[ \text{Active} \right] \]

\[ \Rightarrow t > \text{Tactive} + \text{Twait} \]

\[ + \text{Tgateopen} + \text{Topen} \]

The assumptions 1, 2 and 4 are examples of “obvious” parts of train behaviours; but we have to state them explicitly in order to use them in a verification.

Device assumptions

1. It takes at most \( \text{Tgateclose} \) for the gates to close, if the road traffic is not stuck in the RLC

\[ \text{Dasm1} \equiv \left[ \text{Gates } = \text{closing} \land \text{Traffic } \neq \text{stuck} \right] \]

\[ \Rightarrow t \leq \text{Tgateclose} \]

2. It takes at most \( \text{Tgateopen} \) for the gates to open

\[ \text{Dasm2} \equiv \left[ \text{Gates } = \text{opening} \right] \Rightarrow t \leq \text{Tgateopen} \]

3. The physical properties of Gates constrain the value of gates to the cycle: open, closing, opening, open (in that order)

\[ \text{Dasm3} \equiv \]

\[ (\text{Gates } = \text{open} \rightarrow (\text{Gates } = \text{closing})) \]

\[ \land (\text{Gates } = \text{closing} \rightarrow (\text{Gates } = \text{closed})) \]

\[ \land (\text{Gates } = \text{closed} \rightarrow (\text{Gates } = \text{opening})) \]

\[ \land (\text{Gates } = \text{opening} \rightarrow (\text{Gates } = \text{open})) \]

4. The signal switches between “stop” and “go”

\[ \text{Dasm4} \equiv \left[ \neg \text{Signal} \right] \rightarrow \left[ \text{Signal} \right] \land \left[ \text{Signal} \right] \rightarrow \left[ \neg \text{Signal} \right] \]

4.2 Control strategy

The control strategy is selected by the system designer with the purpose of defining an implementable control satisfying the requirements under the assumptions. The following strategy is a formalisation of the obvious finite state control, cycling through phases with passive, approaching and passing trains. This is expressed by the predicate

\[ \text{RCS} \equiv \square(\bigwedge_{i=1}^{7} \text{RCS } i) \]

Approaching trains

1. The gates will remain open when no trains are present

\[ \text{RCS1} \equiv (\neg \text{Active}) \land \left[ \text{Gates } = \text{open} \right] \land \left[ \text{true} \right] \]

\[ \Rightarrow \text{Gates } = \text{open} \]

2. If trains are present, the gates are open for at most \( \text{Treat} \)

\[ \text{RCS2} \equiv \left[ \text{Gates } = \text{open} \land \text{Active} \right] \Rightarrow t \leq \text{Treat} \]

3. It takes at most \( \text{Tnts} \) before the signal is “go” when the gates have closed

\[ \text{RCS3} \equiv \left[ \text{Gates } = \text{closed} \right] \land \neg \text{Signal} \land \left[ \text{Active} \right] \]

\[ \Rightarrow t \leq \text{Tnts} \]
Passing trains

4. The gates remain closed as long as the signal is "go".

\[ RCS4 \equiv [\text{Signal}] \Rightarrow [\text{Gates} = \text{closed}] \]

5. The signal remains "go", while trains are present

\[ RCS5 \equiv ([\text{Active}] \land ([\text{Signal}] \land \text{true})) \]

6. The signal will only indicate "go" for at most \( T_{\text{inactive}} \) after the trains have left

\[ RCS6 \equiv [\neg \text{Active} \land \text{Signal}] \Rightarrow \ell \leq T_{\text{inactive}} \]

7. The gates will remain closed for at most \( T_{\text{wait}} \) after all trains have left

\[ RCS7 \equiv ([\text{Gates} = \text{closed}] \land \neg \text{Active} \land \neg \text{Signal}] \Rightarrow \ell \leq T_{\text{wait}} \]

### 4.3 Verification of requirements

In order to verify that the formal specification of the Control System satisfies the requirements, one must prove: \( ASM \land RCS \Rightarrow \text{Req.} \)

The verifications all have the same structure: From an arbitrary interval, where the antecedent part of the requirement holds, it is shown through several steps of deduction that the consequent holds. The assumptions and the control strategy can be used freely in the deduction, since they all hold for an arbitrary interval (the \( \Box \) distributes over conjunction). Since it is shown to hold for an arbitrary interval it also holds for any subinterval.

#### The safety requirement

The safety requirement is verified without any timing constraints

\[
\begin{align*}
\Rightarrow & \quad (\text{Gates} \neq \text{closed}) \lor (\text{Traffic} = \text{stuck}) & \{\text{Traffic}\} \\
\Rightarrow & \quad (\text{Gates} \neq \text{closed}) \lor (\text{Traffic} \neq \text{stopped}) & \{\text{RTAsm2}\} \\
\Rightarrow & \quad (\text{Gates} \neq \text{closed}) \lor (\text{Gates} = \text{open}) & \{\text{Gates}\} \\
\Rightarrow & \quad (\text{Gates} \neq \text{closed}) \lor (\text{Gates} \neq \text{closed}) & \{\text{RCS4}\} \\
\Rightarrow & \quad \neg \text{Signal} & \{\text{TAAsm1}\} \\
\Rightarrow & \quad \neg \text{Passing}\end{align*}
\]

#### The functional requirements

The proof of the first functional requirement ensuring the flow of the road traffic relies on the maximum opening and closing times for the gates, and the calculated maximum time the gates can be closed:

\[
\begin{align*}
\Rightarrow & \quad \text{Traffic} = \text{stopped} \quad & \{\text{RTAsm2}\} \\
\Rightarrow & \quad \text{Gates} \neq \text{open} \quad & \{\text{DAsm3}\} \\
\Rightarrow & \quad (\text{Gates} = \text{closing} \lor [\ell]) \lor (\text{Gates} = \text{closed} \lor [\ell]) \lor (\text{Gates} = \text{opening} \lor [\ell]) & \{\text{RTAsm2}\} \quad \{\text{DAsm1}\} \\
\Rightarrow & \quad (\ell \leq T_{\text{gateclose}} \lor [\ell]) \lor (\text{Gates} = \text{closed} \lor [\ell]) \lor (\text{Gates} = \text{opening} \lor [\ell]) & \{\text{Lemma}\} \\
\Rightarrow & \quad (\ell \leq T_{\text{gateclose}} \lor [\ell]) \lor (\ell \leq T_{\text{closed}} \lor [\ell]) \lor (\text{Gates} = \text{opening} \lor [\ell]) & \{\text{DAsm2}\} \\
\Rightarrow & \quad (\ell \leq T_{\text{gateclose}} \lor [\ell]) \lor (\ell \leq T_{\text{closed}} \lor [\ell]) \lor (\ell \leq T_{\text{gateopen}} \lor [\ell]) & \{\text{DurCalc}\} \\
\Rightarrow & \quad \ell \leq T_{\text{gateclose}} + T_{\text{closed}} + T_{\text{gateopen}} &
\end{align*}
\]

where we have used the following lemma:

The time that the gates are closed is limited by a sum of the time the gates are closed before the signal indicates "go", the active time with the signal on "go", the time the signal is "go" after the trains have left the crossing, and finally the time the gates can be closed while the system is inactive.

\[
\text{Lemma} \equiv \Box ([\text{Gates} = \text{closed}] \Rightarrow \ell \leq T_{\text{closed}})
\]

Thus, we should choose \( T_{\text{stopped}} \) such that:

\[
T_{\text{gateclose}} + T_{\text{closed}} + T_{\text{gateopen}} \leq T_{\text{stopped}}
\]

The second and third functional requirements are proved in a similar manner.

### 5 Refining the control model

The control model presented above (from now on referred to as \( RCS_0 \)) specified a set of properties of the control system. This control model can be refined to introduce a less abstract control model \( RCS_1 \), a design. In this model, the control system is defined by describing the communication between a controlling processor and some interface units.

#### 5.1 Informal description

The refined control model comprises:

- A controlling processor
- A unit monitoring the positions of the trains
- A unit controlling the train signal
- A unit controlling the gates
- A unit monitoring the gates

The communication history between the processor and the units is captured by extending \( RCS_0 \) with a state variable

\( tr : \text{Event}^* \)
which contains a finite trace (i.e. list) of events. An event occurring at time \( t \) is appended to the previous value of \( tr \). Hence, events occur at the points in time where the value of \( tr \) changes, and \( tr \) holds the history of events occurring prior to or at time \( t \).

A detailed description of the refinement can be found in [15].

5.2 Correctness of refinement

The control strategy \( RCS_1 \) is a correct refinement of the original control strategy \( RCS_0 \) if

\[
RCS_1 \Rightarrow RCS_0
\]

Since \( RCS_0 \) together with the assumptions has been shown to satisfy the requirements, it follows that

\[
RCS_1 \wedge ASM \Rightarrow Req
\]

\( RCS_0 \) is a conjunction of 7 formulas: \( RCS_1, \ldots, RCS_7 \). Verifying that \( RCS_0 \) holds whenever \( RCS_1 \) holds, is therefore done by separately verifying each of the conjuncts. The complete calculations can be found in [15].

6 Conclusion

We have illustrated an approach to requirements engineering for real-time systems using a mathematical specification of system requirements and system design, where a design is verified by calculations. In the first stage a system model is built. Requirements for the system together with assumptions for the system environment are stated. A control model is then constructed, and it is proved that the control model satisfies the requirements whenever the assumptions hold. Design for a particular control system is then introduced and verified to preserve the properties of the control model. The end result is a set of verified specifications, which clearly expresses the responsibilities of component implementors.

In further refinement steps towards concrete programs, we need to know the relationship between programming language semantics and duration formulas. In [3] the Duration Calculus is used to give a real-time semantics to communicating sequential processes and also gives various specifications of schedulers for shared processors. Furthermore, the Duration Calculus has in [7] been used to give semantics to circuits, to prove the correctness of a circuit transformation, and to give a precise definition of delay-insensitive circuits. Current work is going on to extend Duration Calculus to a version with probabilities [8] as well as to mechanise the calculations using existing theorem provers.

References