Specifying and verifying requirements of real-time systems

Ravn, Anders P.; Rischel, Hans; Hansen, Kirsten Mark

Published in:
I E E E Transactions on Software Engineering

Link to article, DOI:
10.1109/32.210306

Publication date:
1993

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

Link back to DTU Orbit

Citation (APA):
Abstract—An approach to specification of requirements and verification of design for real-time systems is presented. A system is defined by a conventional mathematical model for a dynamic system where application specific states denote functions of real time. Specifications are formulas in duration calculus, a real-time interval logic, where predicates define durations of states. Requirements define safety and functionality constraints on the system or a component. A top-level design is given by a control law: a predicate that defines an automaton controlling the transition between phases of operation. Each phase maintains certain relations among the system states; this is analogous to the control functions known from conventional control theory. The top-level design is decomposed into an architecture for a distributed system with specifications for sensor, actuator, and program components. Programs control the distributed computation through synchronous events. Sensors and actuators relate events with system states. Verification is a deduction showing that a design implies requirements.

Index Terms—Real-time systems, requirements engineering, specification, verification.

I. INTRODUCTION

An engineer who is designing an embedded computer system must have a deep insight in the properties of the controlled physical processes. When the system is safety critical, it becomes particularly important that this insight be made explicit and forms the basis for design. This was already observed by Heninger [12] in connection with the A7 flight program, and the viewpoint is pursued in later work by Parnas, for example [32]–[34] and appears also in [18]. It has led to increased focus on requirements engineering [4]. The challenge, however, is to find suitable mathematical theories and notations that allow a designer to record such insight. It is also crucial, but often neglected, that the theory shall make it practical for a designer to use mathematical reasoning when checking that a design conforms to the requirements. In our work with case studies within the Provably Correct Systems (ProCoS) project [3] we have studied the problem of specifying and verifying total system requirements [13], [35], [37]. This paper describes the resulting approach to requirements engineering, design, and verification for real-time control systems.

A central property of such systems is that they can be modeled by states changing over time. The theory of dynamic systems, see, e.g., [23], or more specifically control theory, see, e.g., [7], [22], are rather specialized, however. They concentrate on systems that can be described by a single invariant or control law in the form of a differential equation, or in the discrete case a difference equation with a fixed time step. In order to be really tractable, the equations are restricted to be linear, although some results have been reached for nonlinear systems, e.g., [17]. Dynamic systems theory is thus not readily applicable to systems with varying time steps, nondeterministic state changes, or with control laws that depend on modes of operation. Furthermore, the notations are not well suited for composition of interacting systems, see e.g., part XII of [46].

Composition seems the only way of dealing with the complex state spaces that arise whenever programming is involved, thus we were led to investigate logic where the intimate relationship between conjunction and composition of concurrently active systems [16] can be used as a composition principle. The logic should also be able to specify real-time constraints, and not just partial ordering of actions. There are many real-time logics, but they seem to fall into two broad classes: explicit time or implicit time. With explicit time formalisms, time is an ordinary variable, represented by event occurrence symbols as in RTL [19]. The time variable may also represent a time interval, see, e.g., [43], or it may be a variable in a temporal logic [35]. In explicit time logic, timing constraints are encoded as inequations over arithmetic expressions in time variables. Through some experiments we found this approach to be less satisfactory because there is no clear relationship between a natural language formulation of constraints and the resulting inequations. We observed that the timing constraints seem to be formulated as constraints on the duration of critical states. Implicit time logic, e.g., metric temporal logic [21], [35] or ISL [9] recognize this by using temporal operators that constrain the extent of a state. It is, however, not possible to express critical durations of the following form: "Within any period of length T, the critical state S must only occur c% of the time." Such formulations describe a constraint on the sum of an arbitrary number of extents and leads to a duration concept, i.e., a logic where time is observed through the accumulated presence of a state. It was through joint work in the ProCoS project that duration calculus emerged [47]. It is based on interval temporal logic [1], [2], [10], [29],...
[42] and the duration concept. Duration calculus gives us the means to describe critical durations, progress from state to state, and stability of a state. Furthermore, it has a set of rules that have been useful when reasoning about timing properties. Concurrently with development of duration calculus, we experimented with ways of applying the theory in a systematic manner to requirements specification, design, and verification. Through many iterations the following approach has emerged:

1) Top-level requirements are specified by constraints on a set of entities \( X \) representing the relevant physical states as a function of real time. The constraints are typically a conjunction of formulas. One class of formulas constrain the duration of critical states. Another class of formulas specify progress properties: within a certain time the system should move from one state to another.

2) A top-level design is given by some assumptions and a control law. The assumptions record the intrinsic design of the physical system (the Nat relation in [34]). They are requirements that the system imposes on the environment, for instance, that certain states are considered physically impossible or that certain progress properties are ensured by physical processes. The control law consists of two parts:
   a) A finite state machine or automaton describing how control progresses through a number of phases. This is specified by a formula \( \phi \) over a phase control state \( P \).
   b) A set of phase requirements that for each phase define progress and stability constraints to be satisfied during that phase. A phase requirement determines also whether the phase is stable or control shall enter a new phase.

3) An architecture is defined. It consists of specifications for a set of concurrently operating sensor, actuator and program components that synchronize through events. This distributed system is controlled by a scheduler. The scheduler maintains a trace state \( \tau \), recording the sequence of events passed between components. In order to ensure that a conjunction of component specifications remain consistent, we cannot allow upper bounds on the duration of a phase (this reflects that a set of synchronized state machines can only move when all are ready). Upper bounds are thus rewritten as readiness to progress, using a private state \( \text{Ref} \) (cf. [16]) for each component. The scheduler ensures progress when all components indicate that they are ready to proceed. The system state \( X \) is also distributed as private states for components.

This is an outline of the approach; but we hasten to add that there are still many points that need investigation; we return to some of these points in the conclusion. A similar approach to Requirements Capture and top-level design is found in Parnas’ work [34]. Use of finite state automata to specify designs for safety-critical systems are also investigated in [18] and as a paradigm for fault-tolerant systems in [40]. It is also the basis for the ProCoS program specification language [31] and the Statechart formalism [15]. The decomposition into an architecture is primarily inspired by Roscoe and Reed’s work on timed CSP [38], [41], and He Jifeng’s work on real-time semantics [20].

Section II summarizes duration calculus. Section III presents the running example and the (informal) requirements. Section IV formalizes system requirements. Section V introduces a top-level design with a control law. The top-level design is verified in section VI. Section VII specifies the architecture. Verification is in Section VIII, followed by the conclusion in Section IX.

II. SPECIFICATION LANGUAGE

We use the well-known time-domain model of systems and control theory [7], [22], [23] for modeling systems. A system is described by a collection of states (often called state variables) that are functions of time, modeled by the real numbers.

Properties of systems are expressed by constraining the states over time. We wish to express requirements and design without explicit mentioning of particular time instants, and introduce a notation that is a real-time interval logic based on state durations [14], [47].

A. Syntax

We assume (upper case) names \( X, Y, \ldots \) for the states together with (notation for) their value domains \( \text{Type}_X \), \( \text{Type}_Y, \ldots \), given in a suitable specification language, such as \( \mathbb{Z} \) (cf. [44]) or VDM (cf. [5]). This language must comprise a number of standard data types with operators and constants, including the type \( \mathbb{R} \) of real numbers and the type \( \text{Bool} \) of Boolean. We denote the Boolean constants by \( tt \) and \( ff \) (the names \( \text{true} \) and \( \text{false} \) are reserved for duration formulas). We use lower-case names \( a, b, \ldots, x, y, \ldots \) to denote constants and variables of any type. As usual in mathematical logic, a variable is an arbitrary value that may be bound by a quantifier, whereas a constant is fixed in each interpretation.

State Expressions and State Assertions. A state expression may be of any type and is generated by

1) Any state, constant and variable is a state expression.
2) Any well-formed expression formed from an \( n \)-ary operator symbol and state expressions \( S_1, \ldots, S_n \) is a state expression.

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

Durations and Duration Terms. For any state assertion \( P \), \( \int P \) is a duration and of type \( \mathbb{R} \). A duration term is also of type \( \mathbb{R} \) and generated by

1) Durations, real constants, and real variables are duration terms.
2) Any well-formed expression formed from an \( n \)-ary operator symbol of type \( \mathbb{R} \) and duration terms \( r_1, \ldots, r_n \) is a duration term.

The symbol \( \ell \) is used as an abbreviation for \( \int tt \).

Duration Formulas. Any expression formed from an \( n \)-ary predicate symbol on \( \mathbb{R} \) and duration terms \( r_1, \ldots, r_n \) is an atomic duration formula. A duration formula is of type \( \text{Bool} \).
and generated by

1) Atomic duration formulas and the special symbols true and false are duration formulas.
2) If $D_1$ and $D_2$ are duration formulas, so are the expressions $(\neg D_1)$, $(D_1 \lor D_2)$ and $(\forall x)D_1$, where $x$ is a variable.
3) If $D_1$ and $D_2$ are duration formulas, so is the expression $(D_1 ; D_2)$.

B. Semantics

An interpretation $I$ of our formal system corresponds to a particular execution (run) of the system, where each state $X$ denotes a function

$$I(X) : [0, \infty) \rightarrow \text{Type}_X$$

giving the state as function of time from the start $t = 0$, and where each constant $a$ is interpreted as a value $I(a)$ of appropriate type. For given interpretation $I$, a valuation $V$ assigns a value $V(x)$ to each variable. By evaluating expressions for each point of time, the interpretation extends to state expressions, which denote functions of time. A constant $a$ or variable $x$ is hereby interpreted as the function with constant value $I(a)$ or $V(x)$.

An observation interval (or interval for short) is a closed and bounded interval $[b, e] \subseteq [0, \infty)$. For given interval $[b, e]$, the duration $\int_P$ of a state assertion $P$ denotes the real number

$$\int_b^e \chi_P(t) \, dt$$

where

$$\chi_P(t) = \begin{cases} 1 & \text{if } I(P)(t) = \text{true} \\ 0 & \text{if } I(P)(t) = \text{false} \end{cases}$$

which is the measure of the set of points in $[b, e]$ where the interpretation of $P$ has value true. It is assumed that all state assertions denote integrable functions of time. The interpretation then extends to duration terms and duration formulas on each interval: the interpretation of duration terms and atomic duration formulas is defined on each interval $[b, e]$ by evaluating expressions using the values of subterms on the same interval. The formulas true and false as well as composite duration formulas $\neg D, D_1 \lor D_2$ and $(\forall x)D$ are interpreted on each interval $[b, e]$ in the same way as usual logical formulas in predicate logic, cf. [11].

The interpretation of a “chop” formula $D_1; D_2$ on an interval $[b, e]$ use interpretations of the subformulas $D_1$ and $D_2$ on subintervals of $[b, e]$. It has the value true iff a "chop" point $m$ ($b \leq m \leq e$) can be found such that $D_1$ is true on $[b, m)$ and $D_2$ is true on $[m, e]$.

Validity. A duration formula $D$ holds on the interval $[b, e]$ in the interpretation $I$ iff $D$ has value true on $[b, e]$ for any valuation $V$ for $I$.

The formula $D$ holds from start for the interpretation $I$ iff it holds on any interval of the form $[0, T]$ for the interpretation $I$.

A duration formula $D$ is valid (a tautology) iff it holds for every interval $[b, e]$ in any interpretation $I$. (Note that the validity of a chop formula $D_1; D_2$ on an interval $[b, e]$ may depend on different chop points $m$ for different interpretations.)

It is sufficient for a formula to be valid that it holds from start for every interpretation $I$ (this uses that shifting all functions in an interpretation $I$ a fixed amount of time $b$ yields another interpretation $I'$, such that $D$ holds on $[0, e]$ for $I'$ exactly when $D$ holds on $[b, e]$ for $I$).

Finite Variability. Interpretation of states has been confined to integrable functions in order to make the concept of duration well defined. In order to have a well-founded induction, we require that interpretation of any state assertion $P$ has finite variability: any interval $[b, e]$ can be divided into finitely many subintervals with $I(P)$ constant on each open subinterval.

C. Specifications and Refinement

A specification for a system is a duration formula $D$. An interpretation $I$ is said to satisfy the specification if $D$ holds from start for $I$. For specifications $D_1$ and $D_2$ we say, that $D_2$ is a refinement of $D_1$ if any interpretation satisfying $D_2$ also satisfy $D_1$. It follows that $D_2$ is a refinement of $D_1$ if the duration formula

$$D_2 \Rightarrow D_1$$

is valid.

D. Deductions

A primary goal of using mathematical modeling is the ability to calculate properties of the model. The calculations for our notation are deductions, verifying that the validity of some formulas implies the validity of others.

It is a goal for the work on duration calculus to formalize deductions such that verification is done by calculations without any reference to a semantical model. The work has not yet reached that stage, but a number of useful axioms and deduction rules have been found. They are listed in the Appendix at the end of the paper. It has been proved (cf. [14]) that this formalization of the duration logic is a relative complete extension of real-valued interval temporal logic.

Most verifications for a real-time system consist of case analysis for a moderate number of cases. The individual cases have deductions that are mostly simple calculations. We give manually developed deduction outlines by necessity. The calculations, however, are simple and stereotypic, so there is reasonable hope for assistance from mechanical deduction assistants.

E. Abbreviations

We use standard abbreviations $\land, \Rightarrow, \Leftarrow, \exists$, and we introduce abbreviations for commonly used duration formulas, with a state assertion $P$, duration formulas $D, D_1, D_2$, and a positive time constant $t$ as shown in Table I. The abbreviations have the following semantics:

- $[\cdot]$ holds on point intervals $[b, h]$
- $[P]$ holds on $[b, e]$ if $P$ has value true almost everywhere in $[b, e]$ and if $b < e$
- $\Diamond D$ holds on $[b, e]$ if $D$ holds on some subinterval of $[b, e]$. 


TABLE I
ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Formula</th>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>( t = 0 )</td>
<td>Point</td>
</tr>
<tr>
<td>[P]</td>
<td>((t = 0) \land (t &gt; 0))</td>
<td>Almost everywhere ( P )</td>
</tr>
<tr>
<td>( \Diamond D )</td>
<td>( \text{true}; D; \text{true} )</td>
<td>Somewhere ( D )</td>
</tr>
<tr>
<td>( \Box D )</td>
<td>( \neg(\Diamond(\neg D)) )</td>
<td>Always ( D )</td>
</tr>
<tr>
<td>( D_1 \rightarrow D_2 )</td>
<td>((D_1; \text{true}) \Rightarrow (D_2 \lor (D_1; D_2; \text{true})))</td>
<td>( D_2 ) follows ( D_1 )</td>
</tr>
<tr>
<td>( D_1 \sim t \rightarrow D_2 )</td>
<td>((D_1; \text{true}) \Rightarrow (((t \leq t) \lor ((t \leq t); D_2)) )</td>
<td>( D_1 ) leads to ( D_2 ) in time ( t )</td>
</tr>
</tbody>
</table>

\( \Box D \) holds on \([b, e]\) if \( D \) holds on any subinterval of \([b, e]\).
\( D_1 \rightarrow D_2 \) holds on an interval with \( D_1 \) holding on an initial subinterval if \( D_2 \) holds on some subinterval from the point (if any) where \( D_1 \) ceases to hold.
\( D_1 \sim t \rightarrow D_2 \) holds on an interval with \( D_1 \) holding on an initial subinterval if \( D_2 \) starts holding within time \( t \).

The following rules of precedence are used:
first: \( 
\neg, \Box, \Diamond \)
second: \( \lor, \land, \Rightarrow \)
third: \( \rightarrow, \sim, \sim t \rightarrow \)

The logical operators are overloaded: they are used for state assertions as well as duration formulas, but the meaning can be inferred from the type of the operands, e.g., \( \lor \) denotes disjunction for state assertions in \( [P_1 \lor P_2] \) and disjunction for duration formulas in \( [P_1 ; [P_2]] \).

III. A GAS BURNER SYSTEM

Our example is a simplified version of a computer controlled (on-off) gas burner described in [45]. This is a safety-critical system as an accident may occur if an excessive amount of unburned gas leaks to the environment. Small gas leaks cannot be avoided during ignition. A burning flame may also be blown out causing some gas to leak before the failure is detected. The gas burner is controlled by a thermostat and the gas is ignited by an ignition transformer, cf. Fig. 1. The informal requirements for the gas burner can then be formalized using duration formulas on the system model.

1) For safety, gas must never leak for more than 4 s in any period of at most 30 s:

\( Req_1 \equiv \langle [t \leq 30] \Rightarrow \int (\text{Gas} \land \neg \text{Flame}) \leq 4 \rangle \)

This is a critical duration constraint.

2) Heat request off shall result in the flame being off after 60 s:

\( Req_2 \equiv \langle \neg \text{Heatreq} \Rightarrow (\langle [t \sim 60] \rightarrow \langle [\neg \text{Flame}] \rangle) \rangle \)

This is a guarded progress constraint. The guard is \( \langle \neg \text{Heatreq} \rangle \) and the progress is from an initial point \( \langle [t] \rangle \) to \( \langle [\neg \text{Flame}] \rangle \) within 60 s.

3) Heat request shall, after 60 s, result in gas burning unless an ignite or flame failure has occurred:

\( Req_3 \equiv \langle \text{Heatreq} \Rightarrow (\langle [t \sim 60] \rightarrow [\text{Flame}] \rangle \lor \Diamond \text{IgniteFail} \lor \Diamond \text{FlameFail} \rangle \)

An ignite failure happens when gas does not ignite within 0.5 s:

\( IgniteFail \equiv \langle [\neg (\text{Gas} \land \text{Ignition}) \Rightarrow (\langle [t \sim 0.5] \rightarrow [\text{Flame}] \rangle) \rangle \)

The flame fails if it disappears while gas is supplied:

\( FlameFail \equiv \langle \neg [\text{Gas}] \Rightarrow \neg (\langle [\text{Flame}] \rightarrow [\neg \text{Flame}] \rangle) \rangle \)

A formula \( \neg \Diamond ([P] ; [\neg P]) \) means stability of \( P: P \) cannot change to \( \neg P \).

We use seconds as the time unit.
The total system requirements is the conjunction of the critical duration constraint $Req_1$ and the guarded progress constraints $Req_2$ and $Req_3$

$$Req_{All} \equiv Req_1 \land Req_2 \land Req_3$$

The requirements should hold for any interval $\Box Req_{All}$

V. CONTROL MODEL AND CONTROL LAW

A control law expresses a top-level design of the system. It consists of a control function and assumptions about the behavior of the environment. The control function defines interaction between the control system and the environment. The assumptions define preconditions for the control law. In order to formalize the control law, the control model is formed. It is the system model with an additional state $P$ that records the phase of operation of the controller.

A. Gas Burner Phases

We use a simple version of the control law in [45] (especially simplified with respect to error recovery) for the gas burner. It has the following phases:

- **Idle**: Awaits heat request; no gas and ignition. It enters the Purge phase on heat request.
- **Purge**: Pauses for 30 s, and then Ignitel is entered.
- **Ignitel**: Starts ignition and gas supply; enters the Ignite2 phase after 1 s.
- **Ignite2**: Monitors the flame and enters the Burn phase if flame is sensed within 1 s.
- **Burn**: Ignition is switched off, but gas is still supplied. The Burn phase is stable until heat request goes off. The Idle phase is then entered and the gas is turned off.

We use the simple error recovery procedure of returning to Idle. If a flame is not sensed within 1 s in Ignite2 (ignite failure), or if the flame disappears during the Burn phase (flame failure), then the Idle phase is entered and the gas is turned off. The 30 s Purge pause ensures a sufficient distance between periods with leaking gas.

B. Control Automaton

We describe the possible phase transitions by means of a finite state automaton with states corresponding to the phases. The automaton is shown in Fig. 2. The automaton is defined in the control model using the state $P$ giving the current phase of the system.

$$P: \{idle, purge, ignitel, ignite2, burn\}$$

Its value domain is the finite set of phase names, and the following state assertions describe the individual phases

- $\text{Idle} \equiv P = \text{idle}$
- $\text{Purge} \equiv P = \text{purge}$
- $\text{Ignitel} \equiv P = \text{ignitel}$
- $\text{Ignite2} \equiv P = \text{ignite2}$
- $\text{Burn} \equiv P = \text{burn}$

![Fig. 2. Phase transitions for the gas burner.](image)

The automaton is defined by untimed progress constraints

$$Phases \equiv \text{Init} \land \text{Trans}$$

where $\text{Init}$ expresses that the automaton starts in the Idle phase

$$\text{Init} \equiv [\_|\_] \rightarrow [\text{Idle}]$$

and $\text{Trans}$ defines the phase transitions

$$\text{Trans} \equiv \Box (\begin{align*}
&([\text{Idle}] \rightarrow [\text{Purge}]) \\
&\land ([\text{Purge}] \rightarrow [\text{Ignitel}]) \\
&\land ([\text{Ignitel}] \rightarrow [\text{Ignite2}]) \\
&\land ([\text{Ignite2}] \rightarrow ([\text{Burn}] \lor [\text{Idle}])) \\
&\land ([\text{Burn}] \rightarrow [\text{Idle}])
\end{align*})$$

i.e. Idle is followed by Purge, Purge is followed by Ignitel, etc., cf. Fig. 2.

Phase Requirements. The predicate $\text{Phase\_Req}$ specifies the monitoring and control of system states for each phase

$$\text{Phase\_Req} \equiv \Box (\text{Idle\_Req} \land \text{Purge\_Req} \land \text{Ignitel\_Req} \land \text{Ignite2\_Req} \land \text{Burn\_Req})$$

In the formulas, the constant $\varepsilon_1$ denotes an upper bound on progress. Any move will be performed within time $\varepsilon_1$. The constant $\varepsilon_2$ gives, on the other hand, a lower bound on stability. The system will not move before time $\varepsilon_2$. We assume that $0 < \varepsilon_2 < \varepsilon_1$.

The Idle phase is stable at least $\varepsilon_2$ beyond $\text{Heat\_req}$. The Idle phase is left before $\text{Heat\_req}$ has lasted $\varepsilon_1$. During the Idle phase Gas and Ignition are turned off within $\varepsilon_1$.

$$\text{Idle\_Req} \equiv (\begin{align*}
&([\sim \text{Heat\_req}; \ell \leq \varepsilon_2] \Rightarrow \sim \Diamond ([\text{Idle}; [\sim \text{Idle}])) \\
&\land ([\text{Idle}] \land [\text{Heat\_req}] \Rightarrow \ell \leq \varepsilon_1) \\
&\land ([\text{Idle}] \Rightarrow (\_|\_ \sim \varepsilon_1 \Rightarrow [\sim \text{Gas} \land \sim \text{Ignition}]))
\end{align*})$$

Notice that $\ell \leq \varepsilon_1$ in the second clause can be rewritten to the progress constraint

$$[\_] \sim \varepsilon_1 \rightarrow \text{false}$$

In conjunction with the untimed phase constraint, it will give a next phase within $\varepsilon_1$. 

---

Authorized licensed use limited to: Danmarks Tekniske Informationscenter. Downloaded on November 27, 2009 at 04:43 from IEEE Xplore. Restrictions apply.
A completed Purge phase lasts approximately 30 s, and Gas and Ignition are off within \( \varepsilon_1 \)

\[
\text{PurgeReq} \equiv \\
(\neg \text{Purge}; \ell \leq 30 \Rightarrow \neg \diamond (\neg \text{Purge}; \neg \text{Purge})) \\
\wedge (\neg \text{Purge}; \ell \leq 30 + \varepsilon_1) \\
\wedge (\neg \text{Purge}; \ell \leq 30 + \varepsilon_1) \\
\]

The first clause denotes an unconditional stability of Purge for at least 30 s.

A completed Ignite1 phase lasts approximately 1 s, and Gas and Ignition are on within \( \varepsilon_1 \)

\[
\text{Ignite1Req} \equiv \\
(\neg \text{Ignite1}; \ell \leq 1 \Rightarrow \neg \diamond (\neg \text{Ignite1}; \neg \text{Ignite1})) \\
\wedge (\neg \text{Ignite1}; \ell \leq 1 + \varepsilon_1) \\
\wedge (\neg \text{Ignite1}; \ell \leq 1 + \varepsilon_1) \\
\]

The Ignite2 phase does not enter Idle after less than 1 s and it does not enter Burn after less than \( \varepsilon_2 \) of Flame. It lasts at most \( 1 + \varepsilon_1 \) and it is left within \( \varepsilon_1 \) if the Flame comes on. The Ignite2 phase maintains Gas and Ignition

\[
\text{Ignite2Req} \equiv \\
(\neg \text{Ignite2}; \ell \leq 1 \Rightarrow \neg \diamond (\neg \text{Ignite2}; \text{Idle})) \\
\wedge (\neg \text{Ignite2}; \ell \leq 1 + \varepsilon_1) \\
\wedge (\neg \text{Ignite2}; \ell \leq 1 + \varepsilon_1) \\
\]

Note, that Trans implies that Burn can only be entered from Ignite2. Hence \( \neg \text{Burn} \) in the second clause could be replaced by \( \neg \text{Ignite2} \).

The Burn phase persists \( \varepsilon_2 \) beyond Heatreq and Flame. It is left within \( \varepsilon_1 \) if Heatreq or Flame goes off. During the Burn phase Gas is maintained, but Ignition is switched off within \( \varepsilon_1 \)

\[
\text{BurnReq} \equiv \\
(\neg \text{Ignite2} \wedge \text{Flame}; \ell \leq \varepsilon_1) \\
\Rightarrow \neg \diamond (\neg \text{Ignite2}; \neg \text{Ignite2}) \\
\wedge (\neg \text{Ignite2} \wedge \neg \text{Heatreq} \vee \neg \text{Flame}; \ell \leq \varepsilon_1) \\
\wedge (\neg \text{Ignite2} \wedge \neg \text{Ignite2}) \\
\]

C. Assumptions

The control law also needs assumptions about the physical processes in the system. For the gas burner we have the following assumptions:

1) No gas results in no flame within 0.1 s.

\[
\text{Asm}_1 \equiv (\neg \text{Gas}) \Rightarrow (\neg \text{Flame}; \neg \text{Flame}) \\
\]

2) Gas does not ignite when the ignition transformer is not operating.

\[
\text{Asm}_2 \equiv \\
(\neg \text{Ignition}) \Rightarrow \neg \diamond (\neg \text{Flame}; \text{Flame}) \\
\]

which should hold on any interval

\[
\text{Asm} \equiv \square (\text{Asm}_1 \wedge \text{Asm}_2) \\
\]

VI. CORRECTNESS OF CONTROL LAW

A control law is correct if it implements the requirements under the given assumptions. The formalized requirements, assumptions and control function allow correctness to be expressed formally as the implication

\[
\text{Asm} \wedge \text{Phases} \wedge \text{PhaseReq} \Rightarrow \text{Req} \\
\]

We have \( \text{Phases} = \text{Init} \wedge \text{Trans} \), where Init decomposes into \( \text{Init} \) or \( \text{Idle}, \text{true} \). It is easy to see that requirements hold for the point interval, thus \( \text{Init} \Rightarrow \text{Req} \). For \( \text{Idle}, \text{true} \), we use that the automaton returns to Idle, such that this case is subsumed by

\[
\text{Asm} \wedge \text{Trans} \wedge \text{PhaseReq} \Rightarrow \text{Req} \\
\]

We now consider separate cases for each requirement:

1) \( \text{Asm} \wedge \text{Trans} \wedge \text{PhaseReq} \Rightarrow \text{Req}_1 \)
2) \( \text{Asm} \wedge \text{Trans} \wedge \text{PhaseReq} \Rightarrow \text{Req}_2 \)
3) \( \text{Asm} \wedge \text{Trans} \wedge \text{PhaseReq} \Rightarrow \text{Req}_3 \)

A. Verification of Req1

We first use PhaseReq to estimate the duration of the critical state

\[
\text{Leak} \equiv \text{Gas} \wedge \neg \text{Flame} \\
\]

for each phase.

We define Limits to be the following conjunction of estimates

\[
\text{Limits} \equiv \\
(\text{Idle} \Rightarrow \text{Gas} \leq \varepsilon_1) \\
\wedge (\text{Purge} \Rightarrow \text{Gas} \leq \varepsilon_1) \\
\wedge (\text{Ignite1} \Rightarrow \ell \leq 1 + \varepsilon_1) \\
\wedge (\text{Ignite2} \Rightarrow \ell \leq 1 + \varepsilon_1) \\
\wedge (\text{Burn} \Rightarrow \text{Gas} \leq 2 \cdot \varepsilon_1) \\
\]

The deduction of

\[
\text{Asm} \wedge \text{Trans} \wedge \text{PhaseReq} \Rightarrow \text{Limits} \\
\]

can be split into simple cases for each conjunct. We take the first one in great detail

\[
\text{IdleReq} \wedge \text{Idle} \\
\Rightarrow (\neg \text{Gas} \Rightarrow \neg \text{Ignition}) \\
\Rightarrow (\neg \text{Gas} \Rightarrow \neg \text{Gas}) \\
\Rightarrow \text{Gas} \leq \varepsilon_1 \vee (\text{Gas} \leq \varepsilon_1, \text{Gas} = 0) \\
\Rightarrow \text{Gas} \leq \varepsilon_1 \vee \text{Gas} = 0 + 0 \\
\]
and omit the deductions for the next three cases. The last case is

\[ \Box \text{BurnReq} \land \Box \text{Asm2} \land \Box \text{Burn} \]
\[ \implies \Box (\neg \text{Flame} \implies \ell \leq \varepsilon_1) \land \Box \text{Asm2} \]
\[ \land (\lceil \neg \text{Ignition} \rceil) \]
\[ \implies \Box (\neg \text{Flame} \implies \ell \leq \varepsilon_1) \land \]
\[ (\lceil \neg \text{Flame} \rceil \land \ell \leq \varepsilon_1) \]
\[ \implies \ell \leq \varepsilon_1 \lor (\ell \leq \varepsilon_1; \text{Flame}) \]
\[ \lor (\ell \leq \varepsilon_1; \neg \text{Flame}) \]
\[ \land \left( (\neg \text{Flame} \land \ell \leq \varepsilon_1) \right) \]
\[ \implies \ell \leq 2 \cdot \varepsilon_1 \]

where we have used

\[ \neg \Diamond (\neg \text{Flame} \land \text{Flame}) \]
\[ \implies \text{Flame} \lor \neg \text{Flame} \lor (\text{Flame} \land \neg \text{Flame}) \]

The second step in the verification is also a case analysis. The cases are defined by the initial phase of an arbitrary interval. The premise is

\[ \text{Pre}_1 \equiv \text{Trans} \land \Box \text{Limits} \land \ell \leq 30 \]

in

1) \[ \text{Pre}_1 \land (\Box \text{Idle}; \text{true}) \implies f\text{Leak} \leq 2 \cdot \varepsilon_1 \]
2) \[ \text{Pre}_1 \land (\Box \text{Purge}; \text{true}) \implies f\text{Leak} \leq 2 + 7 \cdot \varepsilon_1 \]
3) \[ \text{Pre}_1 \land (\Box \text{Ignitel}; \text{true}) \implies f\text{Leak} \leq 2 + 6 \cdot \varepsilon_1 \]
4) \[ \text{Pre}_1 \land (\Box \text{Ignite2}; \text{true}) \implies f\text{Leak} \leq 1 + 5 \cdot \varepsilon_1 \]
5) \[ \text{Pre}_1 \land (\Box \text{Burn}; \text{true}) \implies f\text{Leak} \leq 4 \cdot \varepsilon_1 \]

The calculations follow a pattern, where previous results are used

\[ \text{Pre}_1 \land (\Box \text{Idle}; \text{true}) \]
\[ \implies \text{Pre}_1 \land \]
\[ (\Box \text{Idle}) \]
\[ \lor (\Box \text{Idle} \land (\Box \text{Purge}; \text{true}) \land \ell \leq 30)) \]
\[ \implies \text{Limits} \land (\Box \text{Idle} \lor (\Box \text{Idle} \land \Box \text{Purge})) \]
\[ \implies f\text{Gas} \leq \varepsilon_1 \lor (f\text{Gas} \leq \varepsilon_1; f\text{Gas} \leq \varepsilon_1) \]
\[ \implies f\text{Gas} \leq 2 \cdot \varepsilon_1 \]

\[ \text{Pre}_1 \land (\Box \text{Purge}; \text{true}) \]
\[ \implies \text{Pre}_1 \land (\Box \text{Purge}) \lor (\Box \text{Burn}; \Box \text{Idle}; \text{true}) \]
\[ \implies \text{Limits} \land (\Box \text{Burn} \lor (\Box \text{Burn}; \Box \text{Idle}; \text{true})) \]
\[ \implies f\text{Leak} \leq 4 \cdot \varepsilon_1 \]

The cases for Ignitel and Ignite2 are similar, and so is

\[ \text{Pre}_1 \land (\Box \text{Ignitel}; \text{true}) \]
\[ \implies \text{Pre}_1 \land (\Box \text{Ignitel}) \]
\[ \lor (\Box \text{Ignitel}; \Box \text{Ignite2}; \Box \text{Idle}) \]
\[ \implies f\text{Leak} \leq 2 + 7 \cdot \varepsilon_1 \]

The case analysis shows that an interval beginning in a Purge phase is most critical, and that \text{Reg}_1 will be satisfied as long as \varepsilon_1 \leq 2/7.

\section*{B. Verification of \text{Req}_2}

We start by proving that \neg \text{Heatreq} maintains the Idle phase

\[ \text{IdleReq} \land \neg \text{Heatreq} \land (\Box \text{Idle}; \text{true}) \]
\[ \implies \neg \Diamond (\Box \text{Idle}; \neg \Box \text{Idle}) \]
\[ \land (\Box \text{Idle} \lor \Diamond (\Box \text{Idle}; \neg \Box \text{Idle})) \]
\[ \implies \Box \text{Idle} \]

The deduction of \text{Reg}_2 is a case analysis, depending on the initial phase of the interval. The premise \text{Pre}_2 is defined

\[ \text{Pre}_2 \equiv \neg \text{Heatreq} \land \Box \text{Asm2} \land \Box \text{Trans} \land \Box \text{PhaseReq} \]

and the cases are

1) \[ \text{Pre}_2 \land (\Box \text{Idle}; \text{true}) \land \ell > 2 \]
\[ \implies (\ell \leq 0.1 + \varepsilon_1; \neg \text{Flame}) \]
2) \[ \text{Pre}_2 \land (\Box \text{Burn}; \text{true}) \land \ell > 3 \]
\[ \implies (\ell \leq 0.1 + 2 \cdot \varepsilon_1; \neg \text{Flame}) \]
3) \[ \text{Pre}_2 \land (\Box \text{Ignitel}; \text{true}) \land \ell > 5 \]
\[ \implies (\ell \leq 1.1 + 3 \cdot \varepsilon_1; \neg \text{Flame}) \]
4) \[ \text{Pre}_2 \land (\Box \text{Ignite2}; \text{true}) \land \ell > 7 \]
\[ \implies (\ell \leq 2.1 + 4 \cdot \varepsilon_1; \neg \text{Flame}) \]
5) \[ \text{Pre}_2 \land (\Box \text{Purge}; \text{true}) \land \ell > 38 \]
\[ \implies (\ell \leq 32.1 + 5 \cdot \varepsilon_1; \neg \text{Flame}) \]

The individual calculations are of the usual form:

\[ \text{Pre}_2 \land (\Box \text{Idle}; \text{true}) \land \ell > 2 \]
\[ \implies (\Box \text{Asm2} \land \Box \text{Idle}; \text{true}) \land \ell > 2 \]
\[ \implies \Box \text{Asm2} \land (\ell \leq \varepsilon_1; \neg \text{Gas}) \]
\[ \implies \ell \leq \varepsilon_1 + 0.1; \neg \text{Flame} \]

where we have used that \neg \text{Heatreq} maintains Idle. The other cases are similar and the deductions are omitted.

\section*{C. Verification of \text{Req}_3}

Assuming \varepsilon_1 < 0.5, which is compatible with \varepsilon_1 \leq 2/7, we deduce that \text{Ignitel2} only leads to \text{Idle} in case of an \text{IgniteFail}.

\[ \Box \text{Ignitel2Req} \land (\Box \text{Ignitel1}; \Box \text{Ignite2}; \Box \text{Idle}) \]
\[ \implies (\Box \text{Gas} \land \Box \text{Ignition}; \ell \geq 1 \land \Box \text{Flame} \implies \ell \leq 0.5)); \text{true} \]
\[ \implies (\Box \text{Gas} \land \Box \text{Ignition}; \neg (\ell \leq 0.5; \text{Flame})) \]
\[ \implies \Box \text{IgniteFail} \]

and that a \text{Burn} phase is preceded by \text{Flame}. This is shown by contradiction

\[ \Box \text{Ignitel2Req} \land (\Box \text{Burn}); \text{true} \]
\[ \implies (\Box \text{Burn}; \text{true}) \land \neg (\text{Ignitel2Req}; \text{Burn}) \]
\[ \implies \text{false} \]
We also deduce that Burn maintains the Flame unless there is a FlameFail

\[ \text{BurnReq} \land ([\text{Flame} \land \text{Gas}]; \text{[Burn]} \rangle \]
\[ \Rightarrow \text{[Gas]} \land ([\text{Flame}]; \text{true}) \]
\[ \Rightarrow \text{[Flame]} \]
\[ \lor (([\text{Gas}] \land \Diamond ([\text{Flame}]; \neg \text{Flame}])) \]
\[ \Rightarrow \text{[Flame]} \lor \text{FlameFail} \]

and finally that Heatreq without FlameFail maintains a Burn phase with Flame

\[ \Box((\text{BurnReq} \land \neg \text{FlameFail}) \land \text{[Heatreq]} \]
\[ \land ([\text{Gas} \land \text{Flame}]; \text{[Burn]}); \text{true}) \]
\[ \Rightarrow \Box((\text{BurnReq} \land \neg \text{FlameFail}) \land \text{[Heatreq]} \]
\[ \land (([\text{Gas} \land \text{Flame}]; \text{[Burn]} \]
\[ \lor (([\text{Gas} \land \text{Flame}]; \text{[Burn]} \]
\[ ; \neg \text{[Burn]}); \text{true}) \]
\[ \Rightarrow ([\text{Flame}]; [\text{Flame} \land \text{Burn}]) \]
\[ \lor \Diamond(\text{BurnReq} \]
\[ \land (((\text{[Burn]} \land \text{[Heatreq} \land \text{Flame}]) \]
\[ ; \neg \text{[Burn]})) \]
\[ \Rightarrow ([\text{Flame}]; [\text{Flame} \land \text{Burn}]) \]

We can now proceed to the main deduction. This is also a case analysis, depending on the initial phase of the interval. We assume that \( \epsilon_1 \leq 1 \). The premise \( P_{\text{pre}} \) is defined

\[ P_{\text{pre}} = [\text{Heatreq} \land \text{Trans} \land \text{PhaseReq} \]
\[ \land \Box \neg \text{IgniteFail} \land \Box \neg \text{FlameFail} \]

and the cases are
1. \( P_{\text{pre}} \land ([\text{Ignite1}]; \text{true}) \land \ell > 5 \)
   \[ \Rightarrow \ell \leq 2 + 2 \cdot \epsilon_1; [\text{Flame}] \]
2. \( P_{\text{pre}} \land ([\text{Purge}]; \text{true}) \land \ell > 36 \)
   \[ \Rightarrow \ell \leq 32 + 3 \cdot \epsilon_1; [\text{Flame}] \]
3. \( P_{\text{pre}} \land ([\text{Idle}]; \text{true}) \land \ell > 37 \)
   \[ \Rightarrow \ell \leq 32 + 4 \cdot \epsilon_1; [\text{Flame}] \]
4. \( P_{\text{pre}} \land ([\text{Burn}]; \text{true}) \land \ell > 38 \)
   \[ \Rightarrow \ell \leq 32 + 5 \cdot \epsilon_1; [\text{Flame}] \]
5. \( P_{\text{pre}} \land ([\text{Ignite2}]; \text{true}) \land \ell > 40 \)
   \[ \Rightarrow \ell \leq 33 + 6 \cdot \epsilon_1; [\text{Flame}] \]

The individual calculations are

\[ P_{\text{pre}} \land ([\text{Ignite1}]; \text{true}) \land \ell > 5 \]
\[ \Rightarrow P_{\text{pre}} \land ((\ell \leq 1 + \epsilon_1 \land [\text{Ignite1}]) \]
\[ ; ([\text{Burn}] \lor [\text{Idle}]; \text{true}) \]
\[ \Rightarrow P_{\text{pre}} \land ((\ell \leq 1 + \epsilon_1 \land [\text{Ignite2}]) \]
\[ ; ([\text{Burn}] \lor [\text{Idle}]; \text{true}) \]
\[ \Rightarrow \ell \leq 2 + 2 \cdot \epsilon_1; [\text{Flame}] \]

where we have used the above results about Ignite2 and Burn.

The second case can, as usual, be reduced using the previous result:

\[ P_{\text{pre}} \land ([\text{Purge}]; \text{true}) \land \ell > 36 \]
\[ \Rightarrow P_{\text{pre}} \land (\ell \leq 30 + \epsilon_1; ([\text{Ignite1}]; \text{true}) \land \ell > 5) \]
\[ \Rightarrow \ell \leq 32 + 3 \cdot \epsilon_1; [\text{Flame}] \]

and the third follows the pattern

\[ P_{\text{pre}} \land ([\text{Idle}]; \text{true}) \land \ell > 37 \]
\[ \Rightarrow \Box \text{IdleReq} \land \ell > 37 \]
\[ \land ([\text{Idle} \land \text{Heatreq}] \land \ell \leq 32 + 3 \cdot \epsilon_1 \land [\text{Flame}] \]
\[ \Rightarrow \ell \leq 32 + 4 \cdot \epsilon_1; [\text{Flame}] \]

The last cases have a surprise, because the observation may start just after a FlameFail in the Burn phase

\[ P_{\text{pre}} \land ([\text{Burn}]; \text{true}) \land \ell > 38 \]
\[ \Rightarrow P_{\text{pre}} \land ([\text{Burn}]; \text{true}) \land \ell > 38 \]
\[ \land (([\neg \text{Flame}]; \text{true}) \lor ([\text{Flame}]; \text{true})) \]
\[ \Rightarrow (P_{\text{pre}} \land ([\text{Burn}]; \text{true}) \land ([\neg \text{Flame}; \text{true}) \lor [\text{Flame}]) \]
\[ \Rightarrow (P_{\text{pre}} \land \ell > 38 \land \ell \leq 32 + 5 \cdot \epsilon_1; [\text{Flame}] \]
\[ \Rightarrow \ell \leq 32 + 5 \cdot \epsilon_1; [\text{Flame}] \]

and

\[ P_{\text{pre}} \land ([\text{Ignite2}]; \text{true}) \land \ell > 40 \]
\[ \Rightarrow P_{\text{pre}} \land \ell > 40 \land (\ell \leq 1 + \epsilon_1; [\text{Burn}]; \text{true}) \]
\[ \Rightarrow \ell \leq 33 + 6 \cdot \epsilon_1; [\text{Flame}] \]

In summary: the control law specifies a correct design if the constants \( \epsilon_1 \) and \( \epsilon_2 \) are chosen such that

\[ \epsilon_1 < 2/7 \quad \text{and} \quad 0 < \epsilon_2 < \epsilon_1 \]

Notice that the latency \( \epsilon_2 \) can be arbitrarily small.

The control law can almost directly be used to implement the system with a processor that accesses sensor and actuator states. The progress constraints map to assignments, whereas stability constraints map to delays, either unconditionally or in a sensor-reading loop.

There is, however, an element of parallel processing in sensor readings. This indicates that a further decomposition of the design is useful.

VII. ARCHITECTURE

An architecture specifies a collection of selected components together with an interconnection scheme. A component is either a subsystem or an elementary component: a program, a sensor, an actuator or a timer. A program implements a state machine that takes the system through specified phases. In the general case, state transitions may depend on values computed from data collected in previous phases. A sensor monitors a physical system state, and it is ready to communicate the
state value to a program when requested to do so. An actuator controls a physical system state, and it is ready to change the state to a communicated value from a program when so requested. A timer implements delays giving lower bounds on the duration of phases.

The interconnection scheme that we propose aims at a distributed system, where the components execute concurrently and synchronize through instantaneous, shared events. The synchronous communication paradigm is inspired by CSP [16].

The interconnection scheme uses a set of events, Event, called the system alphabet. Each component C synchronizes on a subset \( \alpha C \subseteq \text{Event} \) called the C-component alphabet.

Each component has a designated state \( \text{Ref}_C \), which records the subset of \( \alpha C \) that is refused at a given time. Events that are not in \( \alpha C \) are never refused.

A scheduler observes the refusal states for all components and allows the components to move whenever no one refuses an event. Whenever such a move occurs, it is recorded in a history or trace of events

\[ tr : \text{Event}^* \]

It is the only shared state in the architecture and it is only read by components. Refusal states are private for components, and the system states are distributed among sensors and actuators.

### A. Architecture of the Gas Burner

For the gas burner we use the architecture shown in Table II. In this case, where there is only one program component, system and program have the same alphabet.

<table>
<thead>
<tr>
<th>Component</th>
<th>Alphabet</th>
<th>Private States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat request sensor HS</td>
<td>{HeatOn, HeatOff}</td>
<td>( \text{Ref}_{HS} : \text{P}_0 HS )</td>
</tr>
<tr>
<td>Flame sensor FS</td>
<td>{FlOn, FlOff}</td>
<td>( \text{Ref}_{FS} : \text{P}_0 FS )</td>
</tr>
<tr>
<td>Gas actuator GA</td>
<td>{GasOn, GasOff}</td>
<td>( \text{Ref}_{GA} : \text{P}_0 GA )</td>
</tr>
<tr>
<td>Ignition actuator IA</td>
<td>{IgnOn, IgnOff}</td>
<td>( \text{Ref}_{IA} : \text{P}_0 IA )</td>
</tr>
<tr>
<td>1 s timer T1</td>
<td>{Seti, Outi}</td>
<td>( \text{Ref}_{T1} : \text{P}_0 T1 )</td>
</tr>
<tr>
<td>30 s timer T30</td>
<td>{Set30, Out30}</td>
<td>( \text{Ref}_{T30} : \text{P}_0 T30 )</td>
</tr>
<tr>
<td>Program P</td>
<td>Event</td>
<td>( \text{Ref}_P : \text{P}_0 P )</td>
</tr>
</tbody>
</table>

### B. General Properties

Trace \( tr \) accumulates the history of events for the system, i.e., an event occurring at time \( t \) is appended to the previous value of \( tr \). The trace is empty initially

\[ \text{Inittr} \triangleq [ ] \rightarrow [ tr = () ]; \text{true} \]

and \( tr \) is an increasing function of time (in the prefix ordering of sequences of events)

\[ \text{TrIncr} \triangleq [ tr = s_1 ]; [ tr = s_2 ] \Rightarrow [ s_1 < s_2 ] \]

This constraint allows a finite set of events to occur at one time.

Event \( e \) is not appended to trace if it is refused by any component \( C \)

\[ WfRef \triangleq [ \bigvee_{C \in \text{Comp}} e \in \text{Ref}_C ] \Rightarrow \text{stable}(tr \upharpoonright \{e\}) \]

where \( \text{Comp} \) is the finite set of components and where

\[ \text{stable}(tr \upharpoonright A) \triangleq \exists s : A^* \cdot (tr \upharpoonright A = s) \]

expresses stability for the projection \( tr \upharpoonright A \) of the trace \( tr \) on a subset \( A \subseteq \text{Event} \).

Constraints \( WfRef \) and \( \text{TrIncr} \) allow a finite set of events to occur instantaneously and does not force anything to happen. The actual occurrence of events is controlled by a Scheduler.

The general properties of trace and refusals are collected in the formula

\[ \text{General} \triangleq \text{Inittr} \land \Box (\text{TrIncr} \land \text{WfRef}) \]

### C. Scheduler

The scheduler is the synchronization agent for the distributed system

\[ \text{Scheduler} \triangleq \Box (\text{TrOne} \land \text{TrStep} \land \text{Progress}) \]

Progress is ensured by insisting that the trace \( tr \) remains stable at most \( \delta_1 \) when some event \( e \) is not refused by all components

\[ \text{Progress} \triangleq \text{stable}(tr) \land [ \bigwedge_{C \in \text{Comp}} e \notin \text{Ref}_C ] \Rightarrow \ell < \delta_1 \]

It follows that if an event \( e \) is accepted by all components for time \( \delta_1 \) then the trace becomes extended with some event (which may be different from \( e \)).

We shall not use true concurrency,\(^2\) so we introduce special constraints that force events to happen one at a time and with a minimal distance \( \delta_2 \)

\[ \text{TrOne} \triangleq [ tr = s_1 ]; [ tr = s_2 ] \]

\[ \Rightarrow [ \# s_2 \leq \# s_1 + 1 ] \]

\( ^2 \)If we allow true concurrency, the refusal shall distinguish between e.g., willingness to participate in each of two events and willingness to participate in both events simultaneously. This distinction is possible with a more complicated definition of the refusal.
and

\[ TrStep \triangleq [\text{tr} \not= s_1]; [\text{tr} = s_1]; [\text{tr} \not= s_1] \]

\[ \Rightarrow t > \delta_2 \]

where \# denotes the length of a sequence. Consistency dictates, that \( \delta_2 < \delta_1 \).

D. Program

In order to specify the program for the gas burner we introduce the following regular expression over the alphabet Event

\[
\text{cycle} = \text{IgnOff} \cdot \text{GasOff} \cdot \text{HeatOn} \cdot \text{Set30}.
\text{Out30} \cdot \text{Set1} \cdot \text{IgnOn} \cdot \text{GasOn}.
\text{Out1} \cdot \text{Set1}.
(\text{FlOn} \cdot \text{IgnOff} \cdot (\text{HeatOff} + \text{FlOff}) + \text{OUT1})
\]

\[ \text{ptrace} = \text{pref} (\text{cycle}^*) \]

where "pref" denotes the prefix closure operation on a regular expression.

The program is specified by a refusal constraint: the program refuses an event \( e \) if and only if the event does not extend the program trace \( \text{tr}_{p} (= \text{tr}) \) to a member of ptrace.

\[ \text{Program} \triangleq \exists e \in \text{Refp} \iff [\text{tr}^*(e) \notin \text{ptrace}] \]

It can be verified that trace \( \text{tr} \) always belongs to ptrace

\[ \text{Scheduler} \land \text{General} \land \text{Program} \Rightarrow \text{Ptrace} \]

where

\[ \text{Ptrace} \triangleq \square ([] \lor [\text{tr} \in \text{ptrace}]) \]

In order to express the phase in terms of the trace, we introduce the projection \( \text{tr}_{\text{In}} = \text{tr} \upharpoonright \text{In} \) of trace \( \text{tr} \) onto the subalphabet

\[ \text{In} \triangleq \{ \text{FlOn}, \text{FlOff}, \text{HeatOn}, \text{HeatOff}, \text{Out1}, \text{Out30} \} \]

It is obvious that \( \text{tr}_{\text{In}} \) will have the projected property of Ptrace

\[ \text{PIntrace} \triangleq \square ([] \lor [\text{tr}_{\text{In}} \in \text{inptrace}]) \]

where the regular expressions

\[ \text{incycle} = \text{HeatOn} \cdot \text{Out30} \cdot \text{Out1}. \]

\[ (\text{FlOn} \cdot (\text{HeatOff} + \text{FlOff}) + \text{Out1}) \]

\[ \text{inptrace} = \text{pref}(\text{incycle}^*) \]

are the projections of cycle and ptrace on In.

The phases are now described by

\[ \text{Idle} \leftrightarrow \text{tr}_{\text{In}} \in \text{incycle}^* \]

\[ \text{Purge} \leftrightarrow \text{tr}_{\text{In}} \in \text{incycle}^* \cdot \text{HeatOn} \]

\[ \text{Ignite1} \leftrightarrow \text{tr}_{\text{In}} \in \text{incycle}^* \cdot \text{HeatOn} \cdot \text{Out30} \]

\[ \text{Ignite2} \leftrightarrow \text{tr}_{\text{In}} \in \text{incycle}^* \cdot \text{HeatOn} \cdot \text{Out30} \cdot \text{Out1} \]

\[ \text{Burn} \leftrightarrow \text{tr}_{\text{In}} \in \text{incycle}^* \cdot \text{HeatOn} \cdot \text{Out30} \cdot \text{Out1} \cdot \text{FlOn} \]

It follows that the phase changes correspond to occurrences of the in-event as shown in Fig. 3, i.e.,

\[ \text{General} \land \text{Scheduler} \land \text{Program} \Rightarrow \text{Phases} \]

E. Sensors

The heat request sensor (HS) is specified as a conjunction

\[ \text{HeatReqSensor} \triangleq \square ([\text{HSRef} \land \text{HSReady}]) \]

of a stability constraint: The heatOn event is refused if heatreq has been on for less than time \( \delta_2 \) and the heatOff event is refused if heatreq has been off for less than time \( \delta_2 \)

\[ \text{HSRef} \triangleq \]

\[ ([\text{Heatreq}] \cdot t \leq \delta_2 \Rightarrow [\text{HeatOn} \in \text{RefHS}]) \]

\[ \land ([\text{Heatreq}] \cdot t \leq \delta_2 \Rightarrow [\text{HeatOff} \in \text{RefHS}]) \]

and a progress constraint: The heatOn event is not refused when heatreq has been on for time \( \delta_1 \) and the heatOff event is not refused when heatreq has been off for time \( \delta_1 \)

\[ \text{HSReady} \triangleq \]

\[ ([\text{Heatreq}] \Rightarrow ([t \sim \delta_1 \Rightarrow [\text{HeatOn} \notin \text{RefHS}]) \]

\[ \land ([\text{Heatreq}] \Rightarrow ([t \sim \delta_1 \Rightarrow [\text{HeatOff} \notin \text{RefHS}]) \]

Flame sensor (FS) has a similar specification

\[ \text{FlameSensor} \triangleq \square ([\text{FSRef} \land \text{FSReady}]) \]

where

\[ \text{FSRef} \triangleq \]

\[ ([\text{Flame}] \cdot t \leq \delta_2 \Rightarrow [\text{FlOff} \in \text{RefFS}]) \]

\[ \land ([\text{Flame}] \cdot t \leq \delta_2 \Rightarrow [\text{FlOn} \in \text{RefFS}]) \]

and

\[ \text{FSReady} \triangleq \]

\[ ([\text{Flame}] \Rightarrow ([t \sim \delta_1 \Rightarrow [\text{FlOff} \notin \text{RefFS}]) \]

\[ \land ([\text{Flame}] \Rightarrow ([t \sim \delta_1 \Rightarrow [\text{FlOn} \notin \text{RefFS}]) \]

F. Actuators

The gas actuator (GA) is specified as a conjunction

\[ \text{GasActuator} \triangleq \square ([\text{GARedy} \land \text{GAAct}) \]

of two progress constraints: the gas actuator never refuses GasOn or GasOff events

\[ \text{GARedy} \triangleq [\text{RefGA} = […] \lor […] \]

Fig. 3. Phase transitions and in-events.
and, for an empty trace, the gas is off. Otherwise, the value stabilizes \( \delta_1 \) after the latest GasOn or GasOff event

\[
GAAction \equiv \\
\begin{align*}
(\text{tr}(GA) = \emptyset) \Rightarrow \neg \text{Gas} & \\
\land (\text{last}(\text{tr}(GA)) = \text{GasOff}) \\
& \Rightarrow ([1] \sim \delta_1 \Rightarrow \neg \text{Gas}) & \\
\land (\text{last}(\text{tr}(GA)) = \text{GasOn}) \\
& \Rightarrow ([1] \sim \delta_1 \Rightarrow \text{Gas})
\end{align*}
\]

where last(s) denotes the last element in the (nonempty) sequence s.

The ignition actuator (IA) has a similar specification

\[
IgnitionActuator \equiv \Box (IAReady \land IAAccept)
\]

where

\[
IAReady \equiv [\text{Ref} IA = \{\}] \lor []
\]

and

\[
IAAction \equiv \\
(\text{tr}(IA) = \emptyset) \Rightarrow \neg \text{Ignition} & \\
\land (\text{last}(\text{tr}(IA)) = \text{Ignition}) \\
& \Rightarrow ([1] \sim \delta_1 \Rightarrow \neg \text{Ignition}) & \\
\land (\text{last}(\text{tr}(IA)) = \text{Ignition}) \\
& \Rightarrow ([1] \sim \delta_1 \Rightarrow \text{Ignition})
\]

G. Timers

The 1-s timer (T1) is specified as a conjunction

\[
T1Ready \equiv \\
(\text{tr}(T1) = \emptyset) \lor [\text{Set1} \not\in \text{Ref} T1] & \\
\land (\text{last}(\text{tr}(T1)) = \text{Ignition}) \\
& \Rightarrow ([1] \sim \delta_1 \Rightarrow \neg \text{Ignition}) & \\
\land (\text{last}(\text{tr}(T1)) = \text{Ignition}) \\
& \Rightarrow ([1] \sim \delta_1 \Rightarrow \text{Ignition})
\]

of two progress constraints: the timer never refuses the set-timer event Set1, and accepts the time-out event Out1 at most 1 s after the last set-timer event.

\[
T1Ref \equiv \\
[\text{tr}(T1) = s]; [\text{tr}(T1) = s' \neg (\text{Set1})]; [\text{Out1} \not\in \text{Ref} T1] \\
\Rightarrow \ell > 1
\]

From the timer specification it is easy to deduce

\[
[\text{tr}(T1) = s]; [\text{tr}(T1) = s' \neg (\text{Set1})] \\
\Rightarrow \ell > 1
\]

The other timer has a similar specification

\[
T30Ready \equiv \Box (T30Ready \land T30Ref)
\]

where

\[
T30Ready \equiv \\
([1] \lor [\text{Set30} \not\in \text{Ref} T30]) \\
\land (\text{last}(\text{tr}(T30)) = \text{Set30}) \land \text{stable}(\text{tr}(T30)) & \\
\Rightarrow ([1] \sim 30 \Rightarrow [\text{Out30} \not\in \text{Ref} T30])
\]

and

\[
T30Ref \equiv \\
[\text{tr}(T30) = s]; [\text{tr}(T30) = s\neg (\text{Set30})] \\
\Rightarrow \ell > 30
\]

H. Composition

The composite system is specified by the conjunction of the component specifications, the scheduler specification and the general properties

\[
\text{System} \equiv \Box (\text{Program} \land \text{HeatReqSensor} \land \text{FlameSensor} & \\
\land \text{GasActuator} \land \text{IgnitionActuator} \land \text{Timer1} & \\
\land \text{Timer30} \land \text{Scheduler} \land \text{General})
\]

VIII. CORRECTNESS OF ARCHITECTURE

We have already argued that the program implements the automaton

\[
\text{General} \land \text{Scheduler} \land \text{Program} \Rightarrow \text{Phases}
\]

It remains to verify that the architecture refines the Phase requirements.

Here we use that Program is the only component that delays output events, i.e., events in the set

\[
\text{Out} = \text{Event}_1\text{In}
\]

The scheduler ensures that the delay is, at most, \( \delta_1 \). Thus we have

\[
\text{System} \Rightarrow \text{Outprogress}
\]

where

\[
\text{Outprogress} \equiv \Box ([\text{tr} = s \land \text{nextOut}(s)] \Rightarrow \ell < \delta_1)
\]

and

\[
\text{nextOut}(s) \equiv \exists e : \text{Out} \cdot s\neg (e) \in \text{ptrace}
\]

From Outprogress it follows that a sequence of n out-events will take at most \( n \cdot \delta_1 \) time units to complete.

Each phase is defined by a sequence of out-events and is terminated by a single in-event or a choice between in-events, cf. the definition of ptrace and inptrace. The transition to a next phase will only happen when both the program and the sensor or timer are ready.

Using these facts about the trace and the results about Outprogress we have the following:

\[
\text{System} \Rightarrow \\
\Box (\text{LIdle} \land \text{LPurge} \land \text{LIgnite1} \land \text{LIgnite2} \land \text{LBurn})
\]
where the lemmas for the phases are

\[ L_{Idle} \equiv
\]
\[ ([Ignite_1] \Rightarrow ([] \sim 3 \cdot \delta_1 \Rightarrow
(stable(tr) \land [last_3(tr) = \langle IgnOff, GasOff \rangle])
\land \langle HeatOn \notin RefP \rangle)))
\land \langle [Idle]; \neg Idle \Rightarrow \Diamond \langle HeatOn \notin RefE \land Idle \rangle) \]

\[ L_{Purge} \equiv
\]
\[ ([Purge] \Rightarrow [last_3(tr)_{GA\text{Act}} = \langle IgnOff, GasOff \rangle]])
\land \langle [Purge]; \neg Purge \Rightarrow \Diamond \langle Out_{30} \notin Ref_{30} \land Purge \rangle) \]

\[ L_{Ignite_1} \equiv
\]
\[ ([Idle] \Rightarrow ([] \sim 4 \cdot \delta_1 \Rightarrow
(stable(tr) \land [last_3(tr) = \langle Set_{1, IgnOn, GasOn} \rangle]
\land \langle Out_{1} \notin RefP \rangle)))
\land \langle [Ignite_1]; \neg Ignite_1 \Rightarrow \Diamond \langle Out_{1} \notin Ref_{1} \land Ignite_1 \rangle) \]

\[ L_{Ignite_2} \equiv
\]
\[ ([Ignite_2] \Rightarrow [last_2(tr)_{GA\text{Act}} = \langle IgnOn, GasOn \rangle])
\land \langle [Ignite_2]; \neg Ignite_2 \Rightarrow \Diamond \langle Out_{1} \notin Ref_{1} \land Ignite_2 \rangle) \]

\[ L_{Burn} \equiv
\]
\[ ([Burn] \Rightarrow [last(tr)_{\text{Act}} = \langle GasOn \rangle])
\land \langle [Burn] \Rightarrow ([] \sim 2 \cdot \delta_1 \Rightarrow
(stable(tr) \land [last(tr) = \langle IgnOff \rangle]
\land \langle FL_{Off} \notin RefP, HeatOff \notin RefP \rangle))))
\land \langle [Burn]; \neg Burn \Rightarrow
\Diamond \langle FL_{Off} \notin Ref_{3} \lor HeatOff \notin Ref_{3} \rangle \land \langle [Burn] \rangle) \]

We have used \( last_n(s) \), \( n \geq 1 \) to denote the subsequence formed by the last \( n \) elements of \( s \) \( (last_n(s) \) is only defined when \( \#s \geq n \)).

We can now illustrate the verification of phase constraints by

\[ System \Rightarrow Idle\text{Req.} \]

Stability under \( \neg Heat\text{req} \) is verified by deduction to a contradiction. We assume that \( \varepsilon_2 \leq \delta_2 \)

\[ HS\text{Ref} \land \Box L_{Idle} \land \langle \neg Heat\text{req}; t \leq \varepsilon_2 \rangle \]
\land \langle [Idle]; \neg Idle \rangle
\Rightarrow \Box L_{Idle} \land \langle HeatOn \notin RefE \rangle
\land \langle [Idle]; \neg Idle \rangle
\Rightarrow \Diamond \langle HeatOn \notin RefE \land HeatOn \notin RefE \rangle
\Rightarrow false.

Progress to \( Purge \) now is shown by a similar deduction where we assume that \( 4 \cdot \delta_1 < \varepsilon_1 \)

\[ Scheduler \land \Box L_{Idle} \land HS\text{Ready}
\land \langle Idle \land Heat\text{req}; t > \varepsilon_1 \rangle
\Rightarrow Scheduler \land
\langle [Idle]; \neg Idle \Rightarrow \Diamond \langle HeatOn \notin RefE \rangle \land stable(tr) \rangle
\land \langle [Idle]; \neg Idle \Rightarrow \Diamond \langle HeatOn \notin RefE \rangle \land t > \varepsilon_1 \rangle
\Rightarrow false.

Progress to \( \neg Gas \) and \( \neg Ignition \) follows from

\[ \Box L_{Idle} \land GA\text{Act} \land IA\text{Act} \land [Idle]
\Rightarrow ([] \sim 4 \cdot \varepsilon_1 \Rightarrow \langle \neg Gas \rangle)
\land \langle [] \sim 4 \cdot \varepsilon_1 \Rightarrow \langle \neg Ignition \rangle \rangle

Stability of \( Purge \) is verified as follows:

\[ \Box T_{30}\text{Ref} \land \Box L_{Purge} \land \langle \neg Purge; t \leq 30 \rangle
\land \langle [Purge]; \neg Purge \rangle
\Rightarrow \Diamond \langle t \leq 30 \rangle
\land \langle [tr = s]; [tr = s \sim \langle Set_{30} \rangle]
\land \langle Out_{30} \notin Ref_{30} \rangle
\land \langle T_{30}\text{Ref} \rangle
\Rightarrow false.

Verification of progress in \( Ignite_1 \) requires \( 3 \cdot \delta_1 < \varepsilon_1 \).

For \( Ignite_2 \) we have a new kind of deduction in order to verify \( \langle Gas \land Ignition \rangle \) throughout the phase. We look for a contradiction of

\[ \Diamond \langle \neg (Gas \land Ignition) \rangle \land Ignite_2 \]

First, we have, from the properties of \( p\text{trace} \),

\[ System \land \langle true; [Ignite_2] \rangle
\Rightarrow true; [Purge]; [Ignite_1]; [Ignite_2] \]

We can now deduce

\[ System \land \langle [\neg (Gas \land Ignition) \land Ignite_2] \rangle
\Rightarrow \Box (Ignite_1 \land Ignite_2 \land GA\text{Act} \land IA\text{Act})
\land \langle [Purge]; [Ignite_1]
\langle [Ignite_2] \land \langle [\neg (Gas \land Ignition) \rangle\rangle
\Rightarrow (GA\text{Act} \land IA\text{Act}) \land
\langle (t \geq 1 - 3 \cdot \delta_1
\land [last_2(tr)_{GA\text{Act}} = \langle IgnOn, GasOn \rangle]
\land \langle true; [\neg (Gas \land Ignition) \rangle\rangle
\Rightarrow false.
provided that $\delta_1 \leq 1 - 3 \cdot \delta_1$; i.e., $4 \cdot \delta_1 \leq 1$ or $\delta_1 < 1/4$.

Maintenance of Gas in the Burn phase requires a similar argument.

In summary: whenever $\delta_1$ is chosen such that $5 \cdot \delta_1 < \varepsilon_1 < 2/7$ the distributed system will work. Notice that the minimal latency $\delta_2$ can be arbitrarily small because $\varepsilon_2$ has no lower bound. In other words: a faster computer will not invalidate the architecture. This is achieved by using explicit timers to implement delays.

IX. CONCLUSION

We have illustrated an approach to requirements engineering and design of real-time systems using mathematical specifications of system requirements and system design. We have demonstrated how mathematical reasoning is used in verifying that designs satisfy requirements and in proving that a more detailed distributed design satisfies an abstract centralized design.

Duration calculus has been our tool. This logic combines central properties of integral calculus with the serial composition ability of interval logic and the parallel composition properties of usual logic. Furthermore, it has a conventional dynamic system as model, which gives a strong link to well established mathematical theories used in control engineering.

The approach is still developing as we gain more understanding of the abilities of duration calculus, and as we see how good design principles are reflected in the formulas. We now discuss some issues in the three stages of the approach.

A. System Model and Requirements

As presented here, the requirements are straightforward formalizations of user expectations. It would be useful to have a more systematic approach for eliciting these expectations. One possibility that we would like to investigate is to derive the safety requirements from the results of a systematic safety analysis of the system. This could, for instance, be done by formalizing the results of fault tree analysis, cf. the approach in the British Ministry of Defence draft standard for software in safety critical systems [27], [28] or the approach suggested in [39].

B. Control Model and Law

The use of a finite state machine to give the control structure is not new, but we see it as a strength that the paradigm is used by other researchers (see the introduction). Our main reason for choosing such a restricted structure is that we would like a design to be consistent. It is fairly obvious that the specification of the automaton Phases has a model. We have taken some pains to give the phase-state constraints PhaseReq a form that ensures consistency. The system states HeatReq and Flame occur only in the premise of the individual phase requirements, and only in the form of single occurrences of mutually exclusive state assertions, e.g. $\neg$HeatReq and Heatreq. These states are thus free to vary in a model. The controlled states Ignition and Gas occur in the same form, but in the consequence for the individual phase requirements. It is thus possible to assign consistent values for each phase. We can also check that upper bounds on the duration of a single phase are higher than the possible lower bounds. For example, for Ignite2 we have an upper bound of $1 + \varepsilon_1$ and lower bounds of 0 or 1. Thus the Phases and PhaseReq constraints have a reasonable model. The assumptions Asm might spoil this, but we have used a form where premisses contain mutually disjoint assertions on the controlled states, and consequences are constraints on the free state Flame. This leaves Heatreq unconstrained, as we expect it to be.

We are less concerned with consistency of top-level requirements. In fact, we expect them to be inconsistent at the start of a development—user expectations are generally too high. They will be relaxed during the design verification activity because it would be impossible to find a consistent design that satisfies all of them.

The control law specification could be used directly for development of a program with shared variables, linking to such approaches as [6], [8], and [24]. We might add that a control law can be refined by expanding a phase into subphases, e.g., the division of the Ignite phase. We do not foresee problems with such serial refinements. They are analogous to refinement of sequential programs.

The present formulation of control laws would also aid in a generalization to hybrid systems [25], [26], and [30], where continuous physical states can be constrained by differential equations in the individual phases or in the assumptions. This requires an extension of the duration concept to properties of continuous states and a notation for initial values in an interval for such states. Such an extension was introduced in [37].

C. Verification

With the restricted form of predicates used in the verification, there is some indication that these might be decidable. This would allow mechanical support for the tedious parts of the calculations. If a finite state machine is inadequate or inconvenient, it is possible to add further control state variables and thus get the full power of a Turing machine. This will, however, make verification more difficult because the phases may be interrelated in nonobvious ways. It would also make mechanized verification support difficult.

D. Architecture, Components, and Scheduling

We have not pursued the state machine approach for a distributed architecture because of the involved proof obligations for shared variables. We have also refrained from pursuing an asynchronous event approach because timing constraints would have to be formulated as constraints on arrival and departure time for elements in unbounded buffers. We have seen dynamic buffer systems in practice, and they have not convinced us that arguments for timeliness are assisted by having global buffer pools. In the presentation, we have tried to build on the fundamental work by Reed on Timed CSP [38]. We have, however, yet to fully clarify the relations between our model and his hierarchy. The concept of a scheduler as an explicit component is elaborated in [48].


**APPENDIX**

**DEDUCTION SYSTEM, VERIFICATION**

General duration calculus is undecidable, and hence we cannot expect to find a complete set of axioms and deduction rules. The deductions in this paper may, however, be based on the set of axioms and deduction rules given as follows.

A suitably decorated $P$, $r$ or $D$ in the formulas denotes a state assertion, a nonnegative real number or a formula in duration calculus. Symbols $f_f$ and $t_t$ are the truth values for rules. The deductions in this paper may, however, be based on state assertions. Note that general laws of the propositional and predicate calculus are not included, whereas some laws of interval temporal logic are.

**Axiom 1:** $ff = 0$

**Axiom 2:** $fP \geq 0$

**Axiom 3:** $fP_1 + fP_2 = f(P_1 + P_2) + f(P_1 \land P_2)$

**Axiom 4:** If $r_1 \geq 0$ and $r_2 \geq 0$, then $(fP = r_1) \land (fP = r_2) \leftrightarrow fP = (r_1 + r_2)$

**Axiom 5:** If $P_1 \land P_2$ is a valid state assertion, then $fP_1 = fP_2$ is an axiom.

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

**Induction Rule:** If $D([\square])$ is deduced, and $D(X \lor (fP; fP_1) \lor (fP; \lnot fP))$ is deducible from $D(X)$, then $D(\square)$ is deduced.

It has a dual backward induction rule.

The following are rules of interval logic used in calculations.

**Interval Law 1:** Monotonic: If $D_1 \Rightarrow D_1'$ and $D_2 \Rightarrow D_2'$ then $D_1 \land D_2 \Rightarrow D_1' \land D_2'$.

**Interval Law 2:** Associative:

$D_1 \land (D_2 \land D_3) \Rightarrow (D_1 \land D_2) \land D_3$

**Interval Law 3:** False-Zero:

$false \leftrightarrow D; false \leftrightarrow false; D$

**Interval Law 4:** Point-Unit:

$D \leftrightarrow D; \square \leftrightarrow \square; D$

**Interval Law 5:** Chop-And:

$((D_1 \land D_2) \land D_3) \land D_4 \Rightarrow (D_1 \land (D_2 \land D_3)) \land D_4$

**Interval Law 6:** Chop-Or:

$D_1 \lor (D_2 \lor D_3) \Rightarrow D_1 \lor (D_2 \lor D_3)$

**Interval Law 7:** Chop-Neg:

$\neg(D_1 \land (D_2 \land D_3)) \Rightarrow (\neg D_2 \land \neg D_3 \land \neg D_1)

\neg((D_1 \land D_2) \land D_3) \Rightarrow (\neg D_1 \land D_2) \land (\neg D_2 \land D_1) \land \neg D_3$

**Interval Law 8:** Exists-Chop:

$\exists v: T \bullet D_1; D_2 \Rightarrow \exists v: T \bullet D_1; D_2$ provided $v$ does not occur free in $D_2$.

$D_1; \exists v: T \bullet D_2 \Rightarrow D_1; D_2$ provided $v$ does not occur free in $D_1$.

The following derived laws have been useful:

**Law 1:** Dur-Range: $0 \leq fP \leq t$

**Law 2:** Dur-Negation: $fP = t \leftrightarrow fP = 0$

**Law 3:** Dur-Chop-Add: Given a predicate over reals $R(r_1, \ldots, r_m)$, which is preserved under addition, i.e., $R(r_1, \ldots, r_m) \land (R(r_1', \ldots, r_s')) \Rightarrow R(r_1 + r_1', \ldots, r_m + r_s')$.

**Law 4:** State-Variation:

$\square \land \exists(\lnot fP; true) \lor (\lnot fP; true) \lor \exists(\lnot fP; true) \lor (\lnot fP; true)$

**Law 5:** State-And:

$P_1 \land P_2 \Rightarrow (P_1 \land P_2)$

**Law 6:** State-Or:

$P_1 \lor P_2 \Rightarrow (P_1 \lor P_2)$

**Law 7:** State-Negation:

$\lnot fP \Rightarrow \lnot fP$

**Law 8:** State-Implies:

$\lnot fP \Rightarrow fP_1 \Rightarrow (\lnot fP \Rightarrow fP_1)$

**Law 9:** State-Always:

$P \Rightarrow (\exists(\lnot fP \land fP_1))$

**Law 10:** State-Chop-And:

$P_1 \land (P_1 \land P_2) \Rightarrow (P_1 \land P_2)$

**Law 11:** State-Chop-Or:

$P_1 \lor (P_1 \land P_2) \Rightarrow (P_1 \land P_2)$

**ACKNOWLEDGMENT**

We wish to express our sincere gratitude to our ProCoS colleagues, especially Prof. D. Bjørner and Prof. C. A. R. Hoare, for encouragement, support and useful discussions, and to Prof. Z. Chaochen for his continuing efforts to develop and adapt duration calculus to our use. Furthermore we have had useful discussions with Prof. J. Madey about the approach.

**REFERENCES**


Anders P. Ravn (M'83) received the M.S. degree in computer science from the University of Copenhagen, Denmark, in 1973. He is Associate Professor with the Department of Computer Science, Technical University of Denmark, Lyngby, Denmark. His research interests are in the areas of software engineering principles for embedded computing systems and specification of total systems in real-time applications.

Prof. Ravn is a member of the Association for Computing Machinery.

Hans Rischel (M'91) received the M.S. degree in mathematics from the University of Copenhagen, Denmark, in 1960. He was Assistant and Associate Professor with the Department of Mathematics, University of Copenhagen, from 1962 to 1970. From 1970 to 1984 he was employed in industry. Since 1985 he has been Associate Professor with the Department of Computer Science, Technical University of Denmark. His research interest is in software engineering with special focus on the use of mathematical methods.

Prof. Rischel is a member of the Association for Computing Machinery.

Kirsten Mark Hansen received the M.S. degree in software engineering from the Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, in 1989. Since 1989, she has been with the Technical University as a research assistant. Her research interests are in the area of software engineering, especially design and verification of safety-critical and embedded real-time systems.