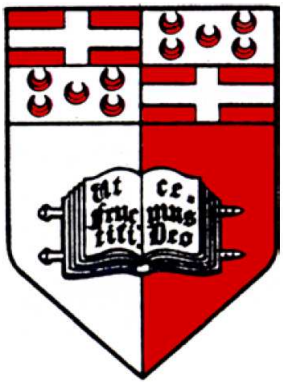
University of Malta



Runtime Monitoring of Duration Calculus Assertions for Real-Time Applications

by
Karlston D'Emanuele

A dissertation submitted in fulfillment of the degree of Masters of Science

Department of Computer Science and A.I.
Faculty of Science
University of Malta
September 2006

Supervisor:

Dr. Gordon Pace

External Examiner:

Dr. Raymond Pascal

Other Examiners:

Dr. Kevin Vella

Mr. Mike Rosner

"If in other sciences we should arrive at certainty without doubt and truth without errors, it behooves us to place the foundations of knowledge in mathematics."

Roger Bacon
English Mathematician (1220–1292)
[Opus Majus, bk. 1, ch. 4]

Abstract

An open question which is commonly asked in software development is whether the implemented artefact follows the requirements specified. From the early days of computing, a number of projects, ideas and techniques have been proposed to prove software correctness. One of these techniques is validation, which verifies the system during execution.

Duration Calculus is a powerful logic notation that evaluates property satisfaction by applying the Reimann integral over property values within an interval. Therefore, Duration Calculus not only determines whether a property is being satisfied but also the duration of the property satisfiability. Duration Calculus notation considers time as a real valued variable, which together with the evaluation of calculi formulae becomes undecidable. In this dissertation, we restrict the notation to a subset that is decidable, discrete-time and deterministic. The decidability property is important in order to evaluate the system correctness at runtime. On the other hand, the restriction to discrete-time together with the determinism of the notation reduce the side-effects of the inserting observers in the actual system thus guaranteeing the correctness of the verified program.

After restricting Duration Calculus notation to the suitable subset of operators, we propose a framework for defining monitors and integrating them with the system code. Our framework allows monitors to be defined using the mathematical notation, which through a pre-compiler is converted to its Lustre semantics and stored inside Abstract Syntax Trees. The synchronous data-flow programming language Lustre is used for the notation semantics because the resource requirements for the monitors can be predetermined. To keep the benefits obtained by using Lustre, the monitoring platform is also defined in Lustre. The final step before executing the system is to integrate the monitors inside the code. The weaving of monitors with the system is performed through the concept of annotated assertions, which are converted into function calls to the Lustre based evaluation engine to determine the properties satisfiability.

We conclude our research by showing the concept of Interval Temporal Logic validation as an aspect, within the Aspect-Oriented Programming (AOP) framework. This concept can be used to facilitate the design of more robust and flexible validation engine simply by defining notation semantics.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Dr. Gordon Pace for his continuous assessments, guidance and knowledge sharing during the entire stages of the dissertation. I also thank him for forwarding the concepts of synchronous programming and Aspect-Oriented Programming that were of major help for the dissertation.

Thanks go to Mr. Joseph Cordina for his assistance in checking the mathematical formulae and for his support. I would also like to thank my colleagues for their support and the good time passed at the University during the research. I take this occasion to thank the Semantics and Verification Research Group for their useful comments and knowledge sharing. I am grateful to the Department of Computer Science and Artificial Intelligence for providing the necessary space and facilities to perform this dissertation with the least inconveniences as possible.

My greatest gratitude goes to my parents who have always given their support and help whenever was possible. I also thank my brother who although not well literate on the subject has shared his ideas and for proof reading the dissertation from the grammatical side. Finally but not least I would like to thank my friends for their support and help throughout the dissertation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> "...unexpected behavior, by definition, violates an application's formal specification, the specification casts a wide net for catching software exceptions, exceptions that you might miss. Therefore, using formal specifications to generate exception handling routines produces a robust hybrid program having multiple levels of recovery paths. The additional levels shield the application from worst-case scenarios that would otherwise crash it."
>
> Doron Drusinsky [Dru01]

A common question that arises during software development is whether the system being implemented is correct according to the given specifications. With the complexities reachable today in software development, guaranteeing correctness of the software is becoming more intractable. Over the past years a number of projects have been commenced in order to propose possible solutions to manage and control software correctness.

From the many projects and ideas proposed in other papers, a solution which is of prime interest to us is about executing runtime monitors in parallel to software code. Runtime monitors can be of different nature ranging from simple propositional statements inserted as conditional checks to the use of temporal logic based automata. Runtime monitors verify the software along its execution path, thus they only guarantee the correctness of the system along a single path. Although this is a drawback of the approach it pays off in resource requirements when compared to the more elaborate process of formal verification. Formal verification consists in performing logical inference over the system specifications to determine their correctness.

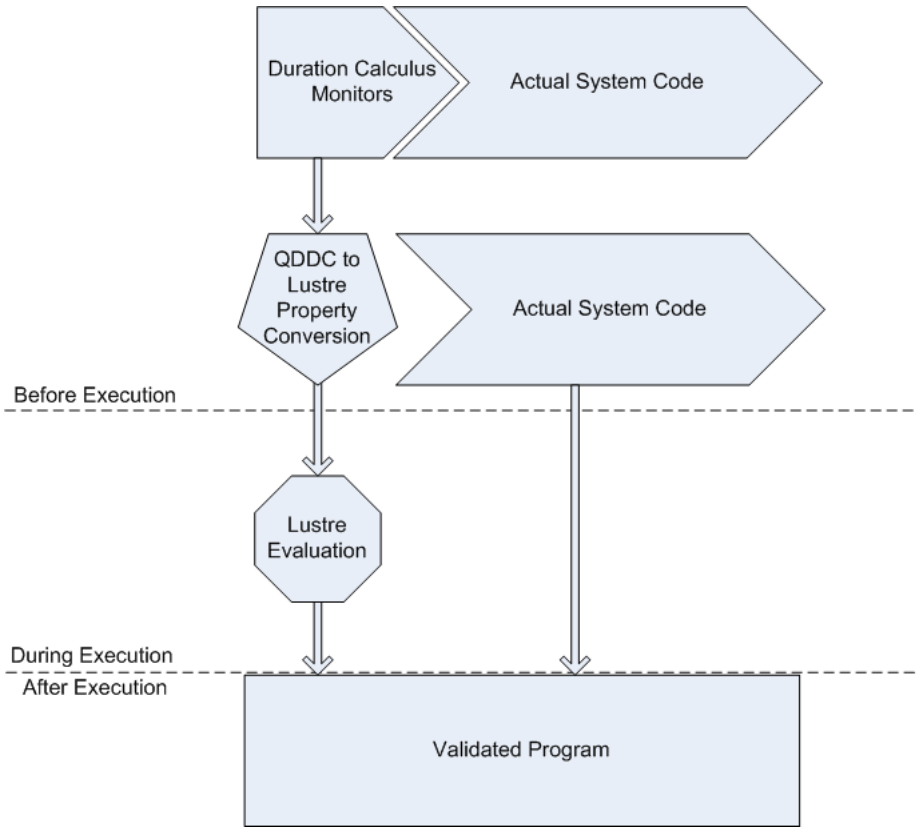


Figure 1.1: Abstracted view of monitors evaluation during program execution.

In this research we are concerned with performing validation using runtime monitors based on a subset of Duration Calculus. Duration Calculus is an Interval Temporal Logic that applies the Reimann integral over properties values to determine their satisfiability. This is possible by considering time over the real numbers. Duration Calculus is highly expressive and its consideration of time as a real number makes it undecidability. Undecidability is surmounted by restricting Duration Calculus notation to its discrete and deterministic subset.

The runtime monitors presented in the dissertation require a library that provides a platform for evaluation. The platform proposed is based on the synchronous data-flow programming language Lustre. Lustre endows the runtime monitors with the capacity of predetermining the real memory and time requirements to evaluate properties. Therefore, runtime monitors are not only suitable for real-time applications due to the mathematical notation but endow the user to predetermine the impacts of the system over the actual system implementation.

1.1 Objectives

Time dependent behaviour is best captured using temporal logics. The purpose of this dissertation is to study the use of Duration Calculus, an interval temporal logic, as a runtime monitor for capturing the correctness of system behaviour. Temporal logics can specify time in both abstract and real-time notation. Abstract notation hides the clock time dimension and gives a means of abstract measure, like clock cycles. Real-time notation provides a means of clock time measurement, say five seconds. Real-time specification is very difficult to handle because one has to take into consideration the temporal side-effects of introducing a monitoring system. Thus, for the purpose of this study the validation engine is designed to handle only abstract description of time.

General Duration Calculus is undecidable and thus requires to be restricted to a decidability subset. This dissertation uses the set named as Quantified Discrete-Time Duration Calculus (QDDC) [Pan00] as the decidable subset of Duration Calculus.

Gonnord *et al* [GHR04] showed that QDDC requires the use of non-deterministic automata. The evaluation of non-deterministic automata requires the tracing of different paths at non-deterministic branches as to determine the system correctness. As a direct result the system requirements increase drastically. For the purpose of this dissertation, deterministic automata are preferred as to predetermine the resource requirements of monitors. Endorsing the work of Gonnord *et al*, the dissertation concentrates on defining a suitable deterministic subset for the monitoring system.

The objectives of the dissertation are summarised below:

1. to identify the discrete and deterministic subset of duration calculus (based on the work done by Gonnord *et al* [GHR04]);
2. define a validation engine for the discrete and deterministic subset of duration calculus using Lustre as a platform;
3. propose a generic and platform independent framework for integrating the validation engine inside the system code; and
4. analyse the outcome to propose suitable enhancements for the validation engine in general.

Figure 1.2 provides an overview of how the dissertation combines the discrete and deterministic subset of duration calculus and Lustre to provide the monitoring solution.

An extended objective of the dissertation is to propose the concept of *Interval Temporal Logic-based validation as an aspect* in terms of the new arising concept of Aspect-Oriented

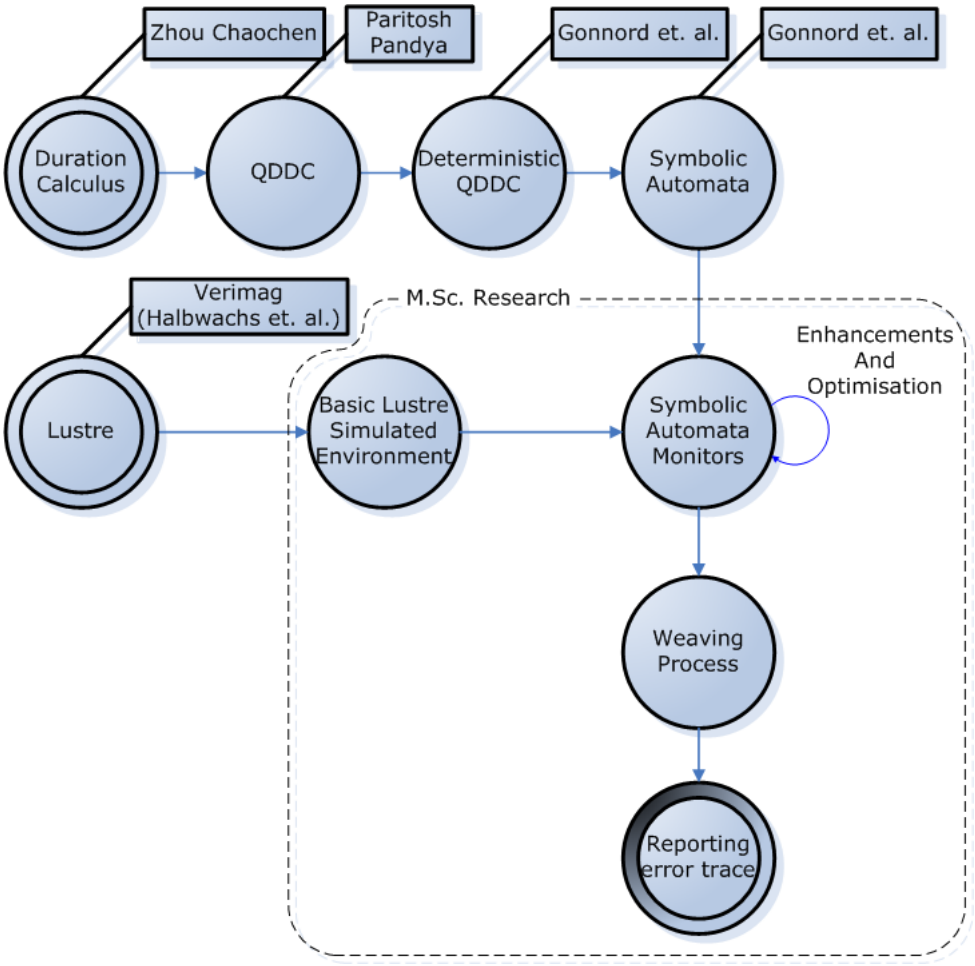


Figure 1.2: Overview diagram of dissertation work.

Programming. This concept frees the design of validation from the complexities involved in the weaving of monitors.

1.2 Report Layout

The dissertation consists of three parts – Background, Discrete and Deterministic subset of Duration Calculus Assertions, and Validation and Aspect-Oriented Programming.

The first part consists in providing the necessary background and a review of related projects. The logic notations, both Duration Calculus (DC) and Quantified Discrete Duration Calculus (QDDC), are defined in terms of Church's lambda calculus, as it provides fundamental relations to practical computing [Pen99]. The literature review introduces the traditional assertion as basic monitoring systems. The chapter continues by reviewing related projects while comparing their approach with our solution. The chapter concludes by fitting the proposed real-time monitoring in the Monitoring-Oriented Programming (MoP) concept.

Part II is the dissertation contribution to the Interval Temporal Logic validation area. Chapter 4 proposes a monitoring restricted logic based on QDDC together with both mathematical and execution semantics. The execution semantics of the logic are then used in Chapter 5 to perform state-by-state validation. The chapter also details the design of a validation engine to monitor systems using a Lustre simulated environment. The second part ends by illustrating two validation engine prototypes. The power of the monitoring system and easiness to use in prototyping is founded by two case studies – a mine pump and an answering machine – as depicted in Chapter 7.

The last part of the report shows how validation is an aspect and how Aspect-Oriented Programming (AOP) tools can be used to weave the monitoring system with a program.

Part I

Background

Chapter 2

Duration Calculus

Duration Calculus is a formal notation introduced by Chaochen *et. al.* [CHR91]. Duration Calculus is based on Interval Temporal Logic (ITL) by Moszkowski [Jos01], thus its notation removes the necessity of quantifying and explicitly mentioning time in the formulae. As with ITL, Duration Calculus uses intervals to determine property satisfaction, which is determined by applying the Reimann's integral over the property truth-time values.

In this chapter we introduce the Duration Calculus notation using Church's lambda calculus (λ-calculus). λ-calculus is adopted as it provides good understanding of how the Duration Calculus notation is evaluated and it simplifies the mapping from mathematical notation to implementation.

The high expressiveness of Duration Calculus leads the notation to be undecidable. In order to define a runtime Duration Calculus monitoring system the second section of this chapter restricts the notation to discrete-time, based on the work done by Pandya [Pan00].

2.1 Duration Calculus Syntax and Semantics

A number of Interval Temporal Logic notations exist [BMN00, Pet99] but they share a common drawback – the fact that time is considered as a discrete variable. In reality, time is continuous and using the notations mentioned before may lead to some system state loss. On the contrary, Duration Calculus models time and behaviour over the real number line [Jos01, CHR91, Rav94]. This is possible through the use of Reimann's integral as a measure of satisfiability over time.

Definition 2.1. *(Time.)* Time is a real valued variable that indicates an instance in the system's life-cycle.

$$\mathbb{T}\text{ime} \stackrel{df}{=} \mathbb{R}$$

Verifying systems throughout their entire life-time is very difficult especially if they do not have any terminating condition. To surmount this problem time is divided into a number of sections referred to as intervals. The following are some important definitions and lemmas that relate to intervals. The interval definitions are later used to define the notation of Duration Calculus.

Definition 2.2. *(Set of intervals.)* The collection of possible divisions of a system timeline.

$$\mathit{Intv} \stackrel{df}{=} \{[b, e] \mid b, e \in \mathbb{T} \cdot b \leqslant e\}$$

For simplification, the boundaries defining the interval are indicated by subscripts b and e. Subscript b represents the start boundary and subscript e represents the end boundary. For example, given any interval, $\mathcal{I}$, $\mathcal{I}_b$ denotes the interval start point and $\mathcal{I}_e$ the end point of the same interval. Formally, for any interval $[s, f]$,

$$[s, f]_b \stackrel{df}{=} s$$

$$[s, f]_e \stackrel{df}{=} f$$

Lemma 2.3. (Set of point intervals.) *Let $\mathcal{I}$ be an interval. Then the set of intervals consisting of a single point is given by:*

$$\text{Intv}_0 \stackrel{df}{=} \{\mathcal{I} \mid \mathcal{I}_b = \mathcal{I}_e\}$$

It must be noted that $\text{Intv}_0 \subset \text{Intv}$.

Definition 2.4. *(Length of interval.)* Time is modelled as a real number. Therefore, the difference between the start and end points of an interval denotes the length of interval.

$$\#(\mathcal{I}) \stackrel{df}{=} \mathcal{I}_e - \mathcal{I}_b$$

Definition 2.5. *(Interval meeting.)* Two intervals are said to meet, or consecutive, if and only if the end point of the first interval and starting point of the second interval are the same.

$$\mathcal{I}; \mathcal{J} \stackrel{df}{=} \mathcal{I} \cup \mathcal{J} \text{ provided that } \mathcal{I}_e = \mathcal{J}_b$$

Definition 2.6. *(State variables.)* A state variable is an atomic boolean variable whose truth varies over time.

$$\text{state variable} \in \mathbb{T} \mapsto \mathbb{B}$$

Definition 2.7. *(True.)* The state variable true returns **true** irrespective of time.

$$\mathbb{1} \stackrel{df}{=} \lambda t : \mathbb{T} \cdot \boldsymbol{true}$$

Likewise,

Definition 2.8. *(False.)* The state variable false returns **false** irrespective of time.

$$\mathbb{0} \stackrel{df}{=} \lambda t : \mathbb{T} \cdot \boldsymbol{false}$$

Definition 2.9. *(State expressions.)* Expressions consisting of logic operators applied to state variables.
For any state variables P and Q

$$\begin{aligned}
\neg P &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot \neg P(t) \\
(P \vee Q) &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot P(t) \vee Q(t) \\
(P \wedge Q) &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot P(t) \wedge Q(t) \\
(P \Rightarrow Q) &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot P(t) \Rightarrow Q(t) \\
(P \Leftrightarrow Q) &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot P(t) \Leftrightarrow Q(t)
\end{aligned}$$

Definition 2.10. *(Duration formula.)* A duration formula is a mapping of state expressions evaluated over an interval to boolean values.

$$\textit{Duration formula} \in \textit{Intv} \mapsto \mathbb{B}$$

Duration Calculus definitions are now introduced based on the definitions given above. An important assumption considered here is that variables have finite variability in a finite interval. In other words, a variable can change its truth value for a finite number of times over finite intervals. This assumption aids in avoiding the Zeno's dichotomy paradox [Wei05].

Definition 2.11. *(Duration.)* The duration of a state expression P in an observation interval $\mathcal{I} \in \textit{Intv}$ is obtained by calculating the area under the behaviour-time graph covered by the interval. In mathematics the area is obtained by taking the Reimann boundary integral of the curve. Therefore,

$$\int P = n \stackrel{df}{=} \lambda \mathcal{I} : \textit{Intv} \cdot \int_{t \in \mathcal{I}} P(t)\, dt = n$$

Definition 2.12. *(Duration expressions.)* Duration expressions consists of duration formulae bound with basic logic operators.
For any duration formulae D and E

$$\begin{aligned}
\neg D &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot \neg D(\mathcal{I}) \\
D \vee E &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot D(\mathcal{I}) \vee E(\mathcal{I}) \\
D \wedge E &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot D(\mathcal{I}) \wedge E(\mathcal{I}) \\
D \Rightarrow E &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot D(\mathcal{I}) \Rightarrow E(\mathcal{I}) \\
D \Leftrightarrow E &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot D(\mathcal{I}) \Leftrightarrow E(\mathcal{I})
\end{aligned}$$

Definition 2.13. *(Chop.)* The chop operator is an arbitrary interval divisor where the left duration formula holds in the first subinterval and the right duration formula in the remaining interval.
For any D and E duration formulas

$$D^\frown E \stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot \exists \mathcal{J}, \mathcal{K} : Intv \cdot \mathcal{I} = \mathcal{J}; \mathcal{K} \wedge D(\mathcal{J}) \wedge E(\mathcal{K})$$

Using the definitons above a number of duration operators can be introduced.

Definition 2.14. *(Not of duration equal to.)* The duration of a state expression P in an observation interval $\mathcal{I} \in Intv$ is not equal to a constant.

$$\int P \neq n \stackrel{df}{=} \neg \left(\int P = n \right)$$

Definition 2.15. *(Duration of at least.)* In an observation interval $\mathcal{I} \in Intv$ the duration of state expression P is at least of n units.

$$\int P \geqslant n \stackrel{df}{=} \left(\int P = n \right)^\frown \mathbf{true}$$

Definition 2.16. *(Of greater duration than.)* A state expression P has a duration greater than a constant value.

$$\int P > n \stackrel{df}{=} \left(\int P \geqslant n \right) \wedge \left(\int P \neq n \right)$$

Endowed with the above formulae the comparison operators are lifted to compare duration formulae.

Definition 2.17. *(Duration equality.)* Two state variables are of equal duration if their duration value is identical for the same interval.

$$\int P = \int Q \stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot \exists n : \mathbb{R}^+ \cdot (\int P = n) \wedge (\int Q = n)$$

Definition 2.18. *(Duration $\neq$.)* Two state variables are said to be of non-equal duration if their duration are of different length for the same observational interval.

$$\int P \neq \int Q \stackrel{df}{=} \neg(\int P = \int Q)$$

Definition 2.19. *(Duration $\geqslant$.)* A duration of a state variable is greater than or equal to another state variable's duration if for the same interval, the number of truths of the first state variable is at least equal to the number of truths of the second state variable.

$$\int P \geqslant \int Q \stackrel{df}{=} (\int P = \int Q)^\frown \boldsymbol{true}$$

Definition 2.20. *(Duration $>$.)* A duration of a state variable is greater than another's state variable duration if and only if for the same interval the first state variable evaluates to true more frequently than the second state variable.

$$\int P > \int Q \stackrel{df}{=} (\int P \geqslant \int Q) \wedge (\int P \neq \int Q)$$

Other comparison operators can be defined in a similar way.

Theorem 2.21. (Duration of true.) *A tautological state variable is equal to the length of the observational interval.*

$$\left(\int \mathbb{1} = n\right)(\mathcal{I}) \;\Leftrightarrow\; \#(\mathcal{I}) = n$$

Proof. The proof is obtained directly from Definitions 2.11 and 2.4.

$$\begin{array}{rcll} \left(\int \mathbb{1} = n\right)(\mathcal{I}) & \stackrel{df}{=} & \int_{t \in \mathcal{I}} \mathbb{1}(t)\, dt = n & \text{Definition 2.11.} \\ & = & [t]_{\mathcal{I}} = n & \text{Evaluation of integral.} \\ & = & \mathcal{I}_e - \mathcal{I}_b = n & \text{Expanding integral result.} \\ & = & \#(\mathcal{I}) = n & \text{Definition 2.4.} \end{array}$$

□

To simplify the reading of duration formulae, *len* will be used interchangeably with $\int \mathbb{1}$.

$$len \stackrel{df}{=} \int \mathbb{1}$$

Corollary 2.22. (Upper duration limit.) *The duration of a state variable P can reach its maximum value if and only if it is a tautology within the interval.*

$$\int P \leqslant \int \mathbb{1}$$

Lemma 2.23. (Duration of a negated state variable.) *For any state variable P the duration of P's complement within an interval is equal to the length of the interval less the duration of P.*

$$\left(\int \neg P = n\right) = \left(\int \mathbb{1} - \int P = n\right)$$

Theorem 2.24. (Duration of false.) *A contradictory state variable is equal to the constant function zero.*

$$\int \mathbb{0}(\mathcal{I}) = 0$$

Proof. The proof is again obtained directly from Definition 2.11.

$$\begin{array}{lll} \int \mathbb{0}(\mathcal{I}) = \int_{t \in \mathcal{I}} \mathbb{0}(t)\ dt & & \text{Definition 2.11.} \\ = [\mathbb{0}t]_{\mathcal{I}} & & \text{Evaluation of integral.} \\ = 0 & & \text{Expanding integral result.} \end{array}$$

□

Corollary 2.25. (Lower duration limit.) *The duration of a state variable P can reach its least value if and only if it is a contradiction within the observation interval.*

$$\int P \geqslant \int \mathbb{0}$$

Theorem 2.26. (Addition of durations.) *The combined duration of two state variables is the sum of the values where either P or Q is true together with the values where both are satisfied simultaneously.*
For any state variables P and Q

$$\int P + \int Q = \int (P \vee Q) + \int (P \wedge Q)$$

As with any logic-based models, state variables are not enough to describe all the features and properties within a system. To extend the expressiveness of Duration Calculus, state variables are lifted to predicates as in interval temporal logic [CHR91].

Definition 2.27. *(Almost everywhere true predicate.)* A state formula P is true for the entire non-point intervals.

$$\lceil P \rceil \stackrel{df}{=} \left(\int P = \int \mathbb{1}\right) \wedge \left(len > 0\right)$$

Definition 2.28. *(Point interval predicate.)* A predicate that is true only for a point interval is denoted by $\lceil\ \rceil$.

$$\lceil\ \rceil \stackrel{df}{=} len = 0$$

Definition 2.29. *(True predicate.)* When a predicate holds for both continuous and point intervals it is said to be a tautological predicate.

$$\boldsymbol{tt} \stackrel{df}{=} len \geqslant 0$$

Theorem 2.30. (Predicate implication.) *Given a predicate P*

$$\lceil P \rceil \Rightarrow \int \neg P = 0$$
$$\lceil\ \rceil \Rightarrow \int P = 0$$

Theorem 2.31. (Duration monotonicity.) *For any states P and Q if $P \Rightarrow Q$ then the duration of P must be less or equal to that of Q.*

$$\lceil P \Rightarrow Q \rceil \Rightarrow (\int P \leq \int Q)$$

Proof. The order preservation in duration formula is conserved through the duration notation semantics.

$\lceil P \Rightarrow Q \rceil \Rightarrow \lceil \neg P \vee Q \rceil$	Implication equivalence
$\Rightarrow \big(\int (\neg P \vee Q) = \int \mathbb{1}\big) \wedge \big(\int \mathbb{1} > 0\big)$	Definition 2.27
$\Rightarrow \int (\neg P) + \int Q - \int (\neg P \wedge Q) = \int \mathbb{1}$	Theorem 2.26
$\Rightarrow \int \mathbb{1} - \int P + \int Q - \int (\neg P \wedge Q) - \int \mathbb{1} = 0$	Theorem 2.23
$\Rightarrow \int P = \int Q - \int (\neg P \wedge Q)$	Basic algebra
$\Rightarrow \int P \leq \int Q$	For some $\int X \geq 0$

□

Using the definition of chop, the two basic time operators of modal logic can be defined as follows.

Definition 2.32. *(Eventually.)* A duration formula D is said to become true if there is a subinterval where it evaluates to true.

$$\Diamond D \stackrel{df}{=} \boldsymbol{tt}^\frown D^\frown \boldsymbol{tt}$$

Definition 2.33. *(Always.)* A duration formula D is always true if it holds over all subintervals within the observation interval.

$$\Box D \stackrel{df}{=} \neg\Diamond\neg D$$

Theorem 2.34. (Duration in subintervals.) *The duration of a state in an interval is the duration of the state in each subinterval.*

Given a state formula P and an interval divided into two subintervals of length r and s respectively.

$$\left(\int P = r + s\right) \Leftrightarrow \left(\int P = r\right)^\frown\left(\int P = s\right)$$

Duration Calculus syntax

$$
\begin{array}{lcl}
<\text{state variable}> & \in & \mathbb{T} \mapsto \mathbb{B} \\
<\text{natural number}> & \stackrel{df}{=} & \mathbb{N}
\end{array}
$$

$$
\begin{array}{lcl}
<\text{state expression}> & ::= & <\text{state variable}> \\
& | & <\text{state expression}> \ \vee \ <\text{state expression}> \\
& | & <\text{state expression}> \ \wedge \ <\text{state expression}> \\
& | & <\text{state expression}> \Rightarrow <\text{state expression}> \\
& | & <\text{state expression}> \Leftrightarrow <\text{state expression}> \\
& | & \neg <\text{state expression}>
\end{array}
$$

$$
\begin{array}{lcl}
<\text{d.c. formula}> & ::= & \int <\text{state expression}> = <\text{natural number}> \\
& | & \int <\text{state expression}> \geqslant <\text{natural number}> \\
& | & \int <\text{state expression}> = \int <\text{state expression}> \\
& | & \int <\text{state expression}> \geqslant \int <\text{state expression}> \\
& | & \lceil <\text{state expression}> \rceil \\
& | & <\text{d.c. formula}> \ \wedge \ <\text{d.c. formula}> \\
& | & <\text{d.c. formula}> \ \vee \ <\text{d.c. formula}> \\
& | & <\text{d.c. formula}> \Rightarrow <\text{d.c. formula}> \\
& | & <\text{d.c. formula}> \Leftrightarrow <\text{d.c. formula}> \\
& | & \neg <\text{d.c. formula}> \\
& | & <\text{d.c. formula}> \ ^\frown <\text{d.c. formula}> \\
& | & \Diamond(<\text{d.c. formula}>) \\
& | & \Box(<\text{d.c. formula}>)
\end{array}
$$

2.1.1 Duration Calculus examples

Consider a simple property stating that over a given interval a loop will not exceed the 30 seconds run in total. The property is represented in Duration Calculus as,

$$\int loop \leqslant 30\, seconds$$

A less rigid control over the loop can be specified as,

$$\Box\,(\lceil loop \rceil \Rightarrow len \leqslant 30\, seconds)$$

The latter formula states that the loop can be executed several times given that every execution never exceeds 30 seconds. On the contrary, the earlier version allows the loop to enter and exit its body for a number of times given that the total execution time does not exceed the 30 seconds interval.

Now consider a more elaborate example. A printer device is designed to determine whether the printer is working smoothly or requires a service. Assume that the printer prints 14 pages per minute. The property that determines the printer state can be specified as

$$\Box\left(\lceil\, spooling\,\rceil^\frown\lceil\, printing\,\rceil^\frown\lceil\, printer\ spool\ clearing\,\rceil \Rightarrow len = \frac{pages}{14} + k\right)$$

The constant, k, is the time required by the device to prepare for printing and to clear the resources used.

2.2 Quantified Discrete-time Duration Calculus (QDDC)

Duration Calculus considers time over the real-numbers thus making it undecidable [Rav94, Pan02b]. In this section Duration Calculus is restricted to discrete-time intervals, in order to make it decidable. The definitions presented are based on the work of Pandya [Pan00].

2.2.1 QDDC syntax and semantics

Duration Calculus is restricted to discrete-time thus the notation is lifted over discrete values.

Definition 2.35. *(Discrete-time.)* Time defined as one dimensional variable ranging over natural numbers.

$$\mathbb{T}ime_{\mathbb{N}} \stackrel{df}{=} \mathbb{N}$$

Definition 2.36. *(Length of interval.)* The length of an interval is the natural number difference between the start and end points.

$$\eta\, op\, c \stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot \left(\mathcal{I}_e - \mathcal{I}_b\right) op\, c$$

where, $op \in \{=, \neq, >, \geqslant, <, \leqslant\}$ and c is a numeric constant.

QDDC inherits the definitions and properties of state variables and expressions from Duration Calculus. However, since dealing with discrete-time, the definition of state expressions is extended to provide means for determining the previous and next values of state expressions.

Definition 2.37. *(Previous value.)* The previous operator, $-P$, provides the immediate past value of a state expression, P. That is, given a state expression, P, at time, t, then the immediate previous value of P is the truth of P at time $t-1$. Using λ-notation this is defined as

$$-P \stackrel{df}{=} \lambda t : \mathbb{T}_{\mathbb{N}} \cdot \begin{cases} P(t-1) & t > 0 \\ \mathit{undefined} & t = 0 \end{cases}$$

The following table illustrates the relation between current and previous value of a state expression using the previous operator.

time	0	1	2	3	4
P	**true**	**false**	**true**	**false**	**true**
$-P$	*undefined*	**true**	**false**	**true**	**false**

Definition 2.38. *(State expressions.)* State expressions consist of boolean logic operators applied over state variables.
For any state variables P and Q

$$\begin{aligned} \neg P &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot \neg P(t) \\ P \vee Q &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot P(t) \vee Q(t) \\ P \wedge Q &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot P(t) \wedge Q(t) \\ P \Rightarrow Q &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot P(t) \Rightarrow Q(t) \\ P \Leftrightarrow Q &\stackrel{df}{=} \lambda t : \mathbb{T} \cdot P(t) \Leftrightarrow Q(t) \end{aligned}$$

Discrete-time endows QDDC to determine the number of times a predicate is satisfied within an interval. This provides an alternative way of dealing with the duration ($\int$) operator of Duration Calculus. Section § 2.1 showed the equivalence between the duration operator and Reimann's bounded integral, which in case of frequent and consistent sampling can be evaluated as the summation of values.

Definition 2.39. *(Number of occurrences.)* For any predicate P, the number of occurrences of P is equal to the number of times P evaluates to **true**.

$$\Sigma P \; op \; c \stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot \left(\sum_{i=\mathcal{I}_b}^{\mathcal{I}_e - 1} \begin{cases} 1 & \textbf{\textit{if}}\ P(i) \\ 0 & otherwise \end{cases} \right) \; op \; c$$

where, $op \in \{=, \neq, >, \geqslant, <, \leqslant\}$.

Duration Calculus predicates are defined over the notation of duration. The restriction to discrete-time invalidates the duration notation, hence, predicates are redefined.

Definition 2.40. *(Continuous time predicates.)* A predicate P is satisfied if for any observational interval the state variable P is constantly true except for the interval end point.

$$\lceil\!\lceil P \rceil\!\rceil \stackrel{df}{=} (\eta > 1) \wedge (\Sigma P = \eta)$$

Contrary to Duration Calculus where one cannot reason about the truth values of state variables at point intervals, in QDDC it is possible to verify the satisfaction of predicates within a point interval.

Definition 2.41. *(Point interval.)* For any predicate P,

$$\lceil P \rceil^0 \stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot \mathcal{I}_e = \mathcal{I}_b \wedge P(\mathcal{I})$$

Definition 2.42. *(Duration formula.)* A duration formula is the mapping of a state expression to a boolean value over an interval.
For any duration formulae D and E

$$\begin{aligned} \neg D &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot \neg D(\mathcal{I}) \\ D \vee E &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot D(\mathcal{I}) \vee E(\mathcal{I}) \\ D \wedge E &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot D(\mathcal{I}) \wedge E(\mathcal{I}) \\ D \Rightarrow E &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot D(\mathcal{I}) \Rightarrow E(\mathcal{I}) \\ D \Leftrightarrow E &\stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot D(\mathcal{I}) \Leftrightarrow E(\mathcal{I}) \end{aligned}$$

The chop and modal operators, sometimes and always, are inherited from Duration Calculus and lifted to discrete-time. The introduction of these three temporal operators empowers the notation with a number of additionally derivable operators.

Definition 2.43. *(Chop.)* Like in Duration Calculus the chop operator is an arbitrary interval divisor where the first duration formula is true in the first subinterval and the second formula is true in the second subinterval.
For any D and E duration formulas

$$D^\frown E \stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot \exists \mathcal{J}, \mathcal{K} : Intv \cdot \mathcal{I} = \mathcal{J}; \mathcal{K} \wedge D(\mathcal{J}) \wedge E(\mathcal{K})$$

Definition 2.44. *(Eventually.)* A duration formula D is eventually true if there exists a subinterval in which D evaluates to true.

$$\Diamond D \stackrel{df}{=} \boldsymbol{tt}^\frown D^\frown \boldsymbol{tt}$$

Definition 2.45. *(Always.)* A duration formula D is true for all subintervals within an interval if there is no subinterval where D is false.

$$\Box D \stackrel{df}{=} \neg \Diamond \neg D$$

Definition 2.46. *(True for the entire intervals.)* For any interval consisting of two subintervals, where the second interval is a point interval equal to the first interval end point, a predicate P is true for the entire interval if and only if it constantly holds in both subintervals.

$$[\![P]\!] \stackrel{df}{=} \left([\![P]\!]^\frown \lceil P \rceil^0 \right) \vee \lceil P \rceil^0$$

Pandya enriches the QDDC notation by introducing two arrow operators. The arrow operators endow the calculus with means to define dependency behaviour between two state variables.

Definition 2.47. *(Leads to.)* Given two predicates P and Q, P leads to Q as soon as and after P has been true for at least δ time.

$$[\![P]\!] \stackrel{\delta}{\rightarrow} [\![Q]\!] \stackrel{df}{=} \Box \Big(\left([\![P]\!] \wedge \eta \geqslant \delta \right) \Rightarrow \left((\eta = \delta)^\frown [\![Q]\!] \right) \Big)$$

Definition 2.48. *(For at least.)* Given two predicates P and Q, Q is true at least for the first δ time of P's validity.

$$[\![P]\!] \stackrel{\delta}{\hookleftarrow} [\![Q]\!] \stackrel{df}{=} \Box (\lceil \neg - P \rceil^{0^\frown} ([\![P]\!] \wedge \eta < \delta) \Rightarrow [\![Q]\!])$$

Sometimes it is required that a state variable is checked for stability in order to provide reliability and consistency.

Definition 2.49. *(Stable.)* A predicate is said to be stable if when it becomes true it remains true for at least δ time.

$$stable(P, \delta) \stackrel{df}{=} \Box(\lceil \neg P \rceil^0 \frown \llbracket P \rrbracket \frown \lceil \neg P \rceil^0 \Rightarrow (\eta > \delta + 2))$$

QDDC syntax

$$
\begin{array}{lcl}
< \text{natural number} > & \stackrel{df}{=} & \mathbb{N} \\
< \text{state variable} > & \in & \mathbb{T} \mapsto \mathbb{B} \\
\text{op} & \in & \{=, >, \geq, <, \leq\}
\end{array}
$$

$$
\begin{array}{lcl}
< \text{state expression} > & ::= & < \text{state variable} > \\
& | & < \text{state expression} > \ \vee \ < \text{state expression} > \\
& | & < \text{state expression} > \ \wedge \ < \text{state expression} > \\
& | & < \text{state expression} > \Rightarrow < \text{state expression} > \\
& | & < \text{state expression} > \Leftrightarrow < \text{state expression} > \\
& | & \neg < \text{state expression} > \\
& | & - < \text{state variable} >
\end{array}
$$

$$
\begin{array}{lcl}
< \text{qddc formula} > & ::= & \llbracket < \text{state expression} > \rrbracket \\
& | & \lceil < \text{state expression} > \rceil^0 \\
& | & < \text{qddc formula} > \ \wedge \ < \text{qddc formula} > \\
& | & < \text{qddc formula} > \ \vee \ < \text{qddc formula} > \\
& | & < \text{qddc formula} > \Rightarrow < \text{qddc formula} > \\
& | & < \text{qddc formula} > \Leftrightarrow < \text{qddc formula} > \\
& | & \neg < \text{qddc formula} > \\
& | & < \text{qddc formula} > \frown < \text{qddc formula} > \\
& | & \exists < \text{state variable} > \cdot < \text{qddc formula} > \\
& | & \Sigma < \text{state expression} > \ op \ < \text{natural number} > \\
& | & \eta \ op \ < \text{natural number} > \\
& | & < \text{qddc formula} > \stackrel{\delta}{\rightarrow} < \text{qddc formula} > \\
& | & < \text{qddc formula} > \stackrel{\delta}{\hookleftarrow} < \text{qddc formula} > \\
& | & \Diamond(< \text{qddc formula} >) \\
& | & \Box(< \text{qddc formula} >)
\end{array}
$$

2.2.2 QDDC examples

Consider again the simple property stating that a loop should never exceed the 30 seconds run. The property can now be represented in QDDC as,

$$\Box(\lceil\!\lceil loop \rceil\!\rceil \Rightarrow \eta \leqslant 30\, seconds)$$

The formula specification above is similar to the second specification in section §2.1.1.

The restriction on property specification can also be seen in the printer device example. The design of the printer provides a mechanism to determine whether the printer is in good condition or requires a service. The printer is assumed to print 14 pages per minute. The property that determines the printer state can be specified as

$$\Box\left(\left(\lceil\, spooling \,\rceil^0 \frown \lceil\!\lceil\, printing \rceil\!\rceil \frown \lceil\, printer\ spool\ clearing \,\rceil^0\right) \Rightarrow \left(\eta = \frac{pages}{14}\right)\right)$$

The constant, k, introduced in the Duration Calculus example has been removed because the restriction to discrete-time allows point intervals to be specified. Thus, properties can be further fine-grained around the critical property.

2.3 Conclusion

Duration Calculus provides a powerful notation for expressing specifications over real-time intervals, however, it is undecidable. The undecidability property is controlled by restricting the notation to discrete-time. Nevertheless, the restriction eliminated certain flexibility of the original notation. As a payoff to this restriction, particular fine-graining and unambiguity are obtained.

Chapter 3

Validation

Validation is a runtime mechanism for checking systems behaviour during execution using formal specifications. From the early days of computing, validation always played a major role in developing correct and reliable systems. The first half of the chapter is a brief overview of the validation history. The chapter proceeds with a discussion on some projects that provide tools for using temporal logic specifications as runtime checks on system behaviour.

3.1 Assertions

Assertions are boolean functions describing the semantics of software elements in formal conditions [Mey97] placed at specific points inside the code. The conditions are placed where they are expected to be always **true** [Hoa01, Voa97, AS01]. In other words, they are placed as traps for detecting errors at an early stage in order to minimise their propagation during implementation and execution [MV04, Ziv03, Ros95]. To highlight their distinction from the code defining the program functionality, assertions are typically annotated as comments.

The concept of assertion testing for proving function correctness has been around from the early days of computing. The first historical appearance of the concept is in 1949 during a conference in Cambridge by Alan Turing. Around two decades later in the late 1960s Floyd and Hoare, independently, introduced formalisms as a driving mechanism for assertions and were the first to propose that assertions must be formulated as part of a program specifications and proven during design.

In the early days, the use of assertions had been limited to programs with a serial structure, that is non-concurrent systems. This was the case until Hehner showed that using Communicating Sequential Processes (CSP) model, the nature of programming is irrespective

to assertion modelling [Hoa01, MV04].

Assertions can be either weak or strong. Weak assertions are traps to detect errors related to the current execution state. A typical weak assertion checks a variable for correct initialisation (checks that it contains a valid value). Strong assertions are more expressive and provide a suitable description of the surrounding functionality. When strong assertions are used, they can be considered as interfaces between different parts of the system [Hoa01].

In general assertions define what functionality the surrounding code is supposed to do rather than how the functionality is performed [MV04, BJHL96, Ros95, Mey97], leading assertions to be typically used as preconditions and post-conditions of functions. Preconditions, post-conditions and also invariants are strongly emphasised in the programming language Eiffel, which implements the concept of Design by Contracts [AS01, Mey97].

Bellini *et. al.* [BMN00] generalises the classification of assertions in two categories: static and dynamic. Static assertions are atemporal, that is, their evaluation is fixed and time independent. Opposite to static assertions there are dynamic assertions, which are temporal formulas whose truth depends on the evaluation time. The latter type of assertions can be categorised on whether they describe safety or liveness properties. Safety properties state that "something bad" should never occur, while liveness properties state that eventually "something good" will happen [Pet99].

Assertion Type	**Assertion Properties**
static	Fixed and Time independent.
dynamic	Temporal formulas and Time dependent values. Classification of dynamic assertions: safety – "something bad" should never happen. liveness – "something good" will eventually happen.

Table 3.1: Assertion Classification

3.1.1 Atemporal assertions

Atemporal assertions consist of propositional logic formulae placed wherever they are expected to evaluate to **true**. Propositional logic consists of a set of elementary facts, that can be assigned either a **true** or **false** value, on which the basic boolean operators can be applied. These are the classical assertions found in programming languages. An atemporal assertion in ANSI-C looks like

```
assert(c >= 'a' && c <= 'z'); // the character is in the English alphabet.
```

Propositional logic assertions are suitable when the programmer requires to place an assumption before executing any particular code. For example, taking a function that evalu-

ates the Fibonacci numbers up to a particular number n, a typical atemporal assertion is that the value of n is greater than zero.

```
FIBONACCI NUMBERS(n)
1  assert(n ⩾ 0)
2  if n = 0
3     then return 0
4  if n = 1
5     then return 1
6     else  return Fibonacci Numbers(n − 1) + Fibonacci Numbers(n − 2)
```

A common atemporal assertions framework is Design by Contract, which is discussed next.

Design by Contract

The methodology of Design by Contract was introduced by Meyer for object-oriented programming languages. It was introduced as a tool for building software with "built-in reliability", that is, integrating assertion specification in analysis, design, implementation and documentation [Mey97]. The idea followed from Harlan Mills' article of 1975, which was titled "How to write correct programs and know it".

Design by Contract methodology views the system requirements as the application of a relation (the contract) between service providers (the routine performing a task) and clients (the callee of the routine). The word "contract" has significant meaning because it describes the rights (preconditions) and obligations (post-conditions) of the parties [Mey97]. The application of contracts endow Design by Contract methodology with the application of the non-redundancy principle, or as Meyer states it "guaranteeing more by checking less". This principle is opposed to what had been in practice and emphasised in that time, whereby to guarantee correctness of routines every step required the introduction of new checks.

Preconditions, typically labelled with the keyword `require`, provide an interface to the callee, which specify under which ideal conditions the routine is bound to return good results. For example, in the Fibonacci algorithm above, the assertion statement (line 1) can be placed as a precondition. This will allow the callee to know that the value passed as a parameter to the routine must be a positive integer number.

Post-conditions, labelled as `ensure`, on the other hand guarantee the callee that the value returned is in the correct range of results and format. Returning again to the Fibonacci algorithm a post-condition can be that the value returned is a positive number.

Design by Contract also introduces two additional types of assertions: class and loop invariants. Class invariants are properties that are expected to hold throughout the class

life-cycle and in all its instances. That is, the invariant is expected to hold on the creation of the instance, before and after every call to the class routines and exactly before the disposal (deletion) of the instance [Mey97, MV04].

Loops are a common construct in many programs, which result in the introduction of looping related problems – for example, a loop that never terminates or when the operations performed in the loop leads to an empty structure. To help in avoiding these problems Meyer uses loop invariants. Loop invariants define properties that are expected to hold in every loop iteration. For example, if a loop is defined to empty a stack instance, then a loop invariant would state that at the beginning of every loop the stack is not empty. Another invariant would be to check that at the end of the loop the stack is empty.

By emphasising the use of the four different types of assertions as an aid for avoiding large amount of conditional checking, Design by Contract provides a methodology to increase reliabilty of the software [Mey97].

The application of Design by Contract in Eiffel produced good results when compared to the simple propositional assertions provided by programming languages. Hence, a number of projects have been undertaken to port the Design by Contract principle into other languages. Some of these projects are Jass (Java with Assertions) [BFMW01], iContract [Kra98], .NET Contract Wizard [AS01], Spec# by Microsoft [BRLS04], Contract in C++ [PP99] and a simple implementation of Design by Contract for C++ [Gue00].

3.1.2 Temporal logic assertions

Atemporal assertions are independent of time. In reactive and real-time systems, time is very important. In order to expand an atemporal assertion to time, a new variable for time has to be introduced in order to apply quantification. For example, a predicate P that is expected to hold for the next 5 time units is defined as

$$\forall x \in [t, t+5] \cdot P(x)$$

In the above formula the variable t represents time. Quantification over time in large properties and systems will become awkward to deal with. Hence, to surmount the intractability arising by quantification over time, temporal logic was introduced.

Temporal logic introduces four new operators to atemporal logic, which substitute quantification over and direct reference to time. The four operators deal with past and future time and allow the specification of whether the property holds for the entire time or will eventually hold. The introduction of the operators is possible because temporal logic considers the *next possible value* relationship between the domain and the range [BMN00].

Generally the future operators are denoted as **G** and **F** to mean the property holds for the entire interval and will eventually hold in the future, respectively. On the other hand, the past operators are denoted as **H** and **P**, where **H** implies that the property was always **true** in the past and **P** indicates that sometime in the past the property was **true**. Formally the operators can be defined as

Definition 3.1. *(Always in the future.)* A property P holds for the entire future.

$$\mathbf{G}P(t) \stackrel{df}{=} \forall t' : \mathbb{T}_{\mathbb{N}} \cdot t' > t \Rightarrow P(t')$$

Definition 3.2. *(Eventually true in the future.)* A property P will eventually evaluate to **true**.

$$\mathbf{F}P(t) \stackrel{df}{=} \exists t' : \mathbb{T}_{\mathbb{N}} \cdot t' > t \wedge P(t')$$

Definition 3.3. *(Always true in the past.)* A property P was always **true**in the past.

$$\mathbf{H}P(t) \stackrel{df}{=} \forall t' : \mathbb{T}_{\mathbb{N}} \cdot t' < t \Rightarrow P(t')$$

Definition 3.4. *(Sometime in the past.)* A property P was **true**sometime in the past.

$$\mathbf{P}P(t) \stackrel{df}{=} \exists t' : \mathbb{T}_{\mathbb{N}} \cdot t' < t \wedge P(t')$$

Temporal logics can be categorised as either linear or branching. Linear temporal logics consists of the subset of operators and transitions which have only one possible next or previous value. On the contrary, branching refers to the subset of logics where the previous or next values are determined by the input formulae. When the previous value is linear the logic is said to be left linear temporal logic, while if it branches then it is left branching. Right linearity and right branching refer to the next value [BMN00].

Figure 3.1: Linear and branching temporal logic diagram

Temporal invariants

Class invariants are atemporal, therefore, they lack in providing a mechanism to properly handle satisfiability related to time. Temporal invariants introduce four quantification operators over class invariants – always, eventually, never and already [MV04, GM03].

An "always" valid invariant is similar to class invariant, however it enforces the invariant check at the termination of the program if the instance is still in memory. An "eventually" valid invariant is an assertion that is expected to become valid during the instance execution. In other words, it must become **true** before the instance is deleted from memory. The "never" valid invariant is opposite to the "always" valid, thus, it must never evaluate to **true** during the class execution. The last type of invariants are the "already" valid invariants. "Already" valid invariants ensure that before executing some particular task a number of required tasks has been completed. For example, before reading a file an "already" valid invariant might assure that the file is first opened.

3.1.3 Interval temporal logic

Interval temporal logic extends propositional logic by defining a temporal structure that is bound in the past, unbound in the future, discrete and linear. The temporal structure measures time in terms of intervals, which provide a higher degree of abstraction; hence leading interval temporal logic to be ideal for specifying real-time systems in which time plays an important role.

The new operators introduced by interval temporal logic are: next ($\bigcirc(\phi)$), always ($\Box$), eventually ($\Diamond$) and chop ($\frown$). The "next" operator states that in the next interval the property enclosed in the brackets evaluates to **true**, this is different from the next operator in propositional temporal logic where it states that the property is **true** in the next evaluation sequence.

The operator "always true" states that the formula following the operator should never evaluate to false. The "always" operator also states that its truth should hold in all possible subintervals within the observation interval, whereas the "eventually true" operator indicates that the formula will become **true** before the end of the interval.

The "chop" operator[1] was first introduced in Choppy logic by Rosner and Pnueli [1986] as an extension to propositional temporal logic and is a binary operator. The "chop" operator allows an interval to be subdivided into any arbitrary two subintervals, where the left-hand side of the operator is satisfied in the first subinterval and the remaining part of the expression is satisfied in the second interval.

Finally interval temporal logic introduces an operator to measure the length of interval, Len($\mathcal{I}$). The operator returns the number of state transitions in the sequence enclosed by the interval. This endows the notation with the ability to determine and specify the duration of the observational intervals.

[1]The chop operator is defined formally in Definition 2.13

3.2 Related projects

The weaving of temporal based properties with application code has been under consideration for quite a long time. This section discusses some of the projects and compares them to the framework described in this dissertation.

3.2.1 EAGLE Flier

The EAGLE project consists of a general framework providing its own logic (named EAGLE), which monitors a program on state basis [BGHS04a, BGHS04b, BGHS03]. The EAGLE logic is rich enough to allow other temporal and real-time logics to be expressed on top of the notation. The logic also endows the framework with the capacity of performing checks without storing any execution traces.

The EAGLE framework is based on the METATEM project. The major difference is that METATEM builds traces state-by-state, whereas EAGLE checks traces state-by-state. Therefore, the EAGLE framework avoids the use of expensive operators such as backtracking and loop-checking. The framework can be used with any logic notation, given that the logic can be translated into EAGLE logic underneath.

The logic consists of a set of primitives which in combination with recursive parametrised equations and minimum/maximum fix-point semantics of three temporal operators can be used to construct formulae. The temporal operators provided are "next-time", "previous-time" and "concatenation". The "concatenation" operator is similar to the "chop" operator (Definition 2.43). As with temporal logics the EAGLE logic is extended with the *until* and *unless* operators.

The generalisation of the EAGLE framework makes it more expressive and powerful when compared to the Deterministic Discrete Duration Calculus Assertions (D^3CA) framework presented in this dissertation. However, D^3CA uses intervals to determine property satisfaction, thus making it ideal for real-time and reactive systems.

An outstanding difference in the implementation of EAGLE when compared to D^3CA is the explicit specification of whether a property is a safety or a liveness property. The necessity of such explicitness arises from the ambiguous use of **false**. In safety properties the boolean value **false** indicates that the property failed to be satisfied. However, in a liveness property the value **false** can indicate either the property *has not* been satisfied yet, or that the property failed to be satisfied. In D^3CA only safety properties are considered. However, the "eventually" ($\Diamond_b$) operator is introduced (as one of the extensions to the logic) to allow certain liveness properties.

Table 3.2 summarises the equivalences and differences between the EAGLE and D^{3}CA.

EAGLE	D^{3}CA
EAGLE logic statements.	QDDC statements.
EAGLE provides a platform for using different logics.	Deterministic QDDC specific framework.
EAGLE logic is a point logic (like LTL)	QDDC is an interval temporal logic.
No execution traces are stored.	No execution traces are stored.
Applies a state-by-state concept.	Adopts the synchrony hypothesis.
Deterministic execution.	Deterministic execution.
Evaluates properties by expressing them into simpler formulas.	Evaluates properties by expressing them into simpler formulas.
Explicit description of liveness and safety (using **min**/**max** prefixes).	Implicit description of liveness and safety (depending on the use of "eventually" or not).
false meaning depends on whether a liveness ("always") or safety ("eventually") is used.	**false** has only one meaning – the property was not satisfied.
Minimal separation of concerns.	Applies separation of concerns by defining properties in a separate XML file.
Combined with test cases generator (Extension [ABG$^+$04]).	Testing is performed manually or during actual system execution.

Table 3.2: EAGLE vs. Deterministic Discrete Duration Calculus Assertions

3.2.2 Java-Monitoring and Checking (Java-MaC)

Java-MaC is a prototype implementation of the more generic Monitoring and Checking (MaC) framework [KKL$^+$01]. The section succinctly describes the MaC framework and later compares it with D^{3}CA framework.

The MaC architecture, depicted in Figure 3.2, consists of two phases. The phases are referred to as *static* and *run-time*.

During the static phase the MaC architecture generates three modules that are used to monitor and check the program. The first module is a filter that observes changes in the program variables and reports the change to the second module, which identifies interesting events. The filter also acts as a mapping mechanism between the implementation variables and the high-level variables used in the formal specifications.

In MaC, formal specifications are described in two languages, primitive event definition language (PEDL) and meta event definition language (MEDL). The usage of two separate languages aid in separating concerns related to the high-level end of specifications and the

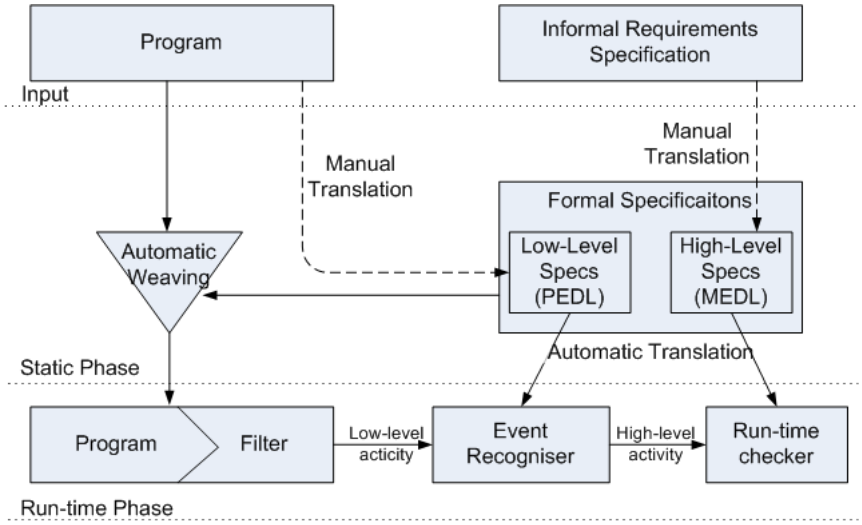


Figure 3.2: Overview of the MaC architecture [KKL+01].

low-level end of implementation.

The event recogniser is defined over PEDL-defined properties. On event recognition, an event instance is initialised and time-stamped. The event instance initialisation consists of the trace that caused the event together with a related value. The value of the event is then used by the run-time checker to determine the state's validity. The time-stamp is used to reflect the time when the event has been raised. The timestamp reflects the time in relation to the monitored program.

The run-time checker evaluates the system description described in MEDL with the event values received. When a property is not satisfied the run-time checker takes care of reporting the issue.

The run-time phase caters for execution of the system and performing the process of information gathering and flow between the different MaC modules.

MaC architecture and D^{3}CA framework have a number of similarities. Some of these are the use of three-valued logic variables, a mapper from implementation variables to specification variables and two phases of execution. The static phase is similar to the weaving performed by D^{3}CA. MaC and D^{3}CA generate the validation engine from a properties description file. The run-time phase where low-level changes are reported to the high-level checker is also quite similar. However, the MaC filter watches for changes in variables and calls/returns to/from functions, whereas the D^{3}CA framework requires the user to specify the interesting points of observation.

Although the general architecture of D^{3}CA and MaC is similar, both frameworks approach the solution from different angles. The MaC framework provides two description languages to specify the properties. The use of different languages and the duplication of property

specification becomes cumbersome in maintaining large systems. On the contrary, the use of QDDC properties in D^{3}CA is effortless.

The property checker provided by MaC and D^{3}CA also differs. In MaC the checker program is a separate process which communicates with the program using the TCP protocol. The separation of the validation engine from the system lessens the space and time requirements of the system. Nevertheless, it removes the direct synchrony provided by D^{3}CA between current state in implementation and automata state.

MaC	D^{3}CA
Uses two languages PEDL and MEDL to formally define the system.	Uses only QDDC for specifying properties.
Programming language independent framework.	Programming language independent framework.
Linear Temporal logic	Interval temporal logic.
Executes remotely over TCP.	Part of the program execution code.
Uses dedicated language	Uses programming language together with XML.
Thread safe.	Current implementation is not thread safe.
Uses three-valued logic variables.	Uses three-values logic variables.
Two phases: static and run-time.	Two phases: pre-compilation and run-time.

Table 3.3: MaC vs. Deterministic Discrete Duration Calculus Assertions

3.2.3 Temporal Rover

Temporal Rover is a propriatary project by Time-Rover. The project consists of three modules, Temporal Rover which performs run-time checking, DBRover which allows remote monitoring and ATG-Rover, which generates test cases for the high-level specifications being tested.

Temporal Rover specifies the properties to be observed in linear-time temporal logic (LTL) and metric temporal logic (MTL) [Dru00, Dru01, DF04]. LTL is used to specify the properties under test, however it lacks an adequate representation of time. To equip Temporal Rover with relative or real-time constraints, MTL is used in conjunction to LTL. The use of two temporal logics contrasts with D^{3}CA, which through the use of Discrete QDDC attains both time description and most of the power endowed by LTL. Nevertheless, MTL endows Temporal Rover with the ability to define properties in bounded time.

The properties are expressed as commented assertions. The logic used for describing the assertions does not provide adequate interval containment, hence the assertion has to be

Temporal Rover	D^3CA
Linear Temporal Logic statements.	QDDC statements.
LTL to explicit automata leading to exponential growth.	QDDC to symbolic automata leading to fixed sized automata.
Assertions as comments.	Assertions as comments.
Comments are converted to code in a new and identical file.	Comments are converted to code in a new and identical file.
Clock on call to assign.	Clock on call to synchronise.
Throws exceptions.	Reports error but can easily be enhanced to throw exceptions.
Support validation on separate processes using DBRover.	Only synchronous (same thread) validation is provided.
Assertions execute every cycle whether or not an exception occurs (can huamper performance).	Assertions execute every cycle if the assertion is valid for the observational interval, whether or not an exception occurs (can hamper performance).
Handles non-determinism	Non-determinism is not supported through QDDC restrictions.

Table 3.4: Temporal Rover vs. D^3CA

placed where their observation starts. Temporal Rover utilises a pre-compiler which given an annotated program creates a similar copy with the assertions replaced with the respective observation code.

The design of Temporal Logic allows non-deterministic or time-unbounded properties to be specified, which increase the validation's degree of complexity. The problem is surmounted by having each property provide some of the four action conditions. The conditions available are:

1. Successful so far – A log message indicating that the property has been satisfied till now. This condition is useful when checking safety properties.

2. Failed so far – A message reporting that the property has not yet been satisfied. Ideal for determining the satisfiability of liveness properties.

3. Finalise – A condition specifying the action to be taken when the assertion is considered to hold for the future once satisfied.

4. Next run – This condition specifies what property should be checked on the next encounter with the assertion.

The first two conditions are provided to keep a trace of the properties that are being evaluated and whether they are being satisfied or not. These properties are useful in situations where the boolean value **false** has an ambiguous meaning. The third condition, "finalise", provides a suitable option for optimising validation. Consider a formula that is

expected to become true during the execution, ◇Property. When the property evaluates to true it is considered to have been satisfied. Therefore, in the "finalise" condition the property can be forced to be removed from the validation system as it will always return true. The "next run" condition allows the property to be rechecked on every encounter. The 'next" condition also allows a property to be checked once and pass control to another assertion.

A dissimilarity between Temporal Rover and D^{3}CA is that the former provides remote monitoring, which aims for embedded systems. In embedded systems, memory and processing power are typically very small and limited. With the use of DBRover, Temporal Rover splits the monitoring in two separate entities. The first entity runs on the embedded system which takes care of computing small sub-property satisfaction and sending new information to the second entity. The second entity runs on a hosting PC, located anywhere, and takes care of running the run-time checking.

Concurrent systems are a common trend in applications. Temporal Rover allows concurrent systems to be specified, however this leads to race-conditions and deadlocks. Therefore Temporal Rover extends the assertions language with the capacity of defining locks at the specification level. The locks are used by assertions to check that while part of the specification is holding a lock, no other concurrent section gets the lock in the meantime.

In Drusinsky [Dru00] the algorithm used by Temporal Rover to perform run-time is reported to be of complexity $\mathcal{O}(n^2)$ where n is the size of the temporal formula being checked. Table 3.4 summarises the comparison between Temporal Rover and D^{3}CA.

3.2.4 Runtime Monitors Assertions (RTMAssertions)

Most of the work in validation consists in defining a framework for verifying properties at runtime, which provides its own mechanism for adding temporal properties and observation points. Thaker [Tha05] shows how temporal logic assertions can be integrated using traditional assertions and aspect-oriented programming, which developers of large scale systems are already used to.

LTL properties can be classified as past or future. Past properties require to keep information about the state variables as they go along. An example of a past property is $\mathbf{H}P$ which states the "the property P must have been always true in the past", Definition 3.3. On the contrary, future properties tend to specify the expected behaviour of the system. In most of the cases, future properties are easier to evaluate because once initialised they can constantly be checked for validity. An advantage of future properties over past properties is that no extra space is consumed for storing a trace of the execution. Thaker [Tha05] concentrates on future properties, although states that the same approach can be also taken

for the past properties.

When performing validation, it is important to specify what defines a state. In RTMAssertions, a state is defined as the tuple (*interesting point,* {*variable values*}), where the interesting point is defined by an annotation that breaks the program execution to perform validation. The set of variable values passes the current state to the monitoring assertions.

An emerging technique for separating different concerns is Aspect-Oriented Programming (AOP). AOP allows aspects that cross between execution points to be defined separately and then through annotations dynamically weaved into the targeted source code. LTL properties tend to stand globally over the system, therefore AOP is a good technique for providing a two layer programming; the The lower layer being the actual implementation and the top layer representing the system in LTL formulae.

The top layer of RTMAssertions is composed of two modules. The first module converts LTL formula into a representative Abstract Syntax Tree (AST), which is then dispatched to the second module. The AST is constructed by using standard compiling techniques over LTL formulae. The second module generates an RTM tree equivalent to the AST received. The RTM tree is then combined with an RTM framework, which provides the implementation of the RTM tree nodes. The coalescence of the tree, framework and program source code is performed by the AOP weaver.

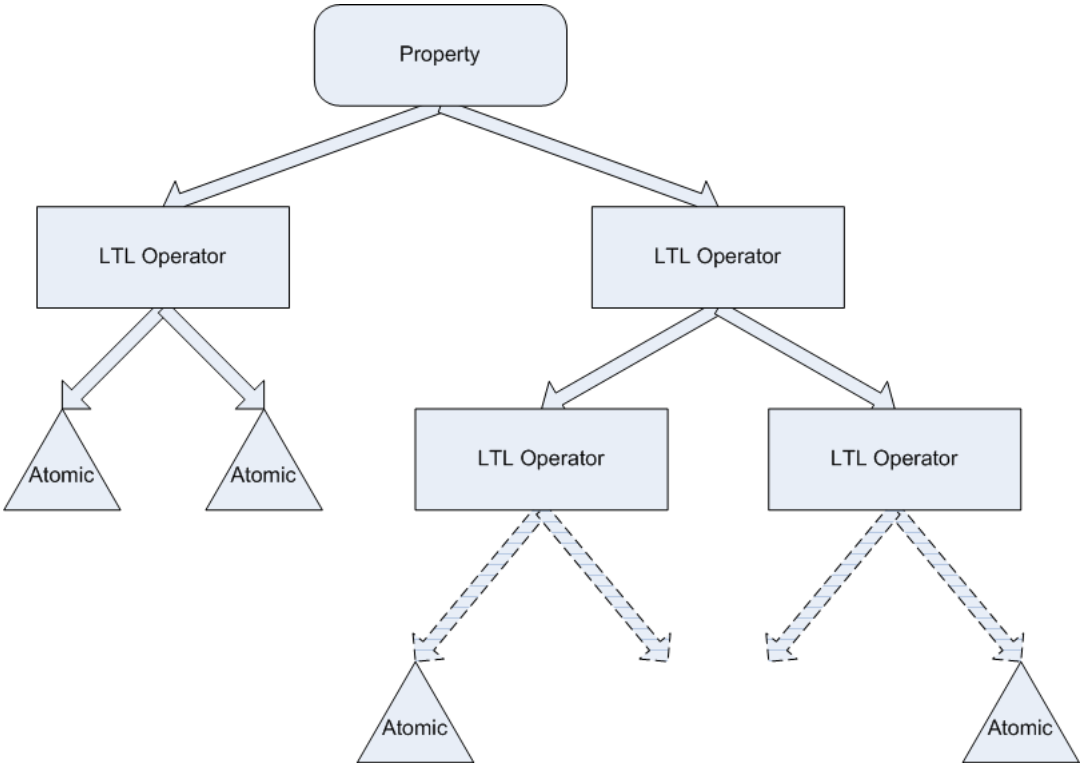


Figure 3.3: Evaluation decomposition of `begin(`P`)`

RTMAssertions are evaluated by initially decomposing them into tangible subformulae,

mainly to their atomics, Figure 3.3. Starting from the atomics *leaves of the RTM tree*, and moving back to the original formula *parent node of the tree*, the formulae are evaluated using the current system state *variable values*. The result of the formula evaluation can be either one of the traditional boolean values **true** or **false**, or **pending**, that is the formula cannot yet determine its final result. A formula is considered to be valid only if at the end of the execution its value is **true**.

Thaker [Tha05] has extended the system implementation with Java PathFinder, a model checker. The final results of RTMAssertions and Java PathFinder were compared and it was found that RTMAssertions are limited in their coverage. The results also showed that if the assertions are well placed around the code, the result over a single execution path is nearly equivalent to that of a model checker over the same execution path.

RTMAssertions and D^{3}CA are very similar. These similarities are outlined in Table 3.5. The main difference between the two is that RTMAssertions uses AOP to achieve separation of concerns while D^{3}CA uses a simple AOP like engine. Another difference is in the logic used to represent the system. RTMAssertions uses LTL which defines time as unbounded in the future, hence all properties must be constantly checked. On the contrary, D^{3}CA define properties over intervals, allowing properties to be checked within their respective interval.

RTMAssertions	D^{3}CA
Linear Temporal Logic statements are used to define properties.	QDDC statements are used to define properties.
LTL statements represented as AST.	QDDC statements are converted to Lustre expressions that are stored as AST.
Assertion point defined as a mark-up annotation.	Assertion point defined as a mark-up annotation.
Assertion comments are converted to code in a new and identical file.	Assertion comments are converted to code in a new and identical file.
Uses AOP tools.	Defines its own weaving engine.
Does not check for order of events – it is assumed in logic.	Checks for order of events using interval concatenation operator ("chop").

Table 3.5: RTMAssertions vs. D^{3}CA

3.2.5 Other projects

The three projects mentioned above are considered to be on the same lines of D^{3}CA. However, a number of other projects have been undertaken in the attempt to define a generic framework for using formal specifications at run-time. This section describes some other validation projects in a succinct way.

Temporal Specifications in Parallel Debugging. The projects described above, as well as D^3CA, require that the developer and the tester either be the same person or the tester has a good knowledge of the development process. They also increase the space and time requirements of the system. Kusper *et. al.* [KSL02] proposed the idea of using runtime assertions as part of a parallel debugger.

This idea consists in creating a program like structure which describes a state machine representing the system. Each state consists of the formulae that are expected to hold inside the state. During the debugging stage the debugger executes the program as usual in parallel to the state machine program. This allows both the visualisation of the program states together with an underlying runtime verification on the program.

The benefit of the approach is that the program tested and the program released are identical. It also releases the tester and developer from the mathematical knowledge imposed by temporal logics because the temporal specifications can be prepared by a third person. Therefore, the tester can concentrate in detecting errors, which may differ from the user's actual needs.

Java PathExplorer (JPaX). JPaX is a large project carried out by NASA in the attempt to define a framework that uses temporal logic specifications as part of runtime debugging [HR04]. JPaX modifies the Java bytecode through JTrek such that whenever observation points are encountered, an execution trace (a stream of events) is generated and submitted to the JPaX validation engine either through a plain file or over sockets. The validation engine checks the execution trace received with the temporal logic specifications provided. An additional feature supported by JPaX is concurrent analysis, which checks the execution trace for possible deadlocks and race conditions.

RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties. RT-MaC extends the MaC framework with time-bound temporal operators. The introduction of time-bound operators endow the MaC framework with the ability to specify quantitative properties required by real-time systems [SLS05]. MaC handles concurrent systems that introduce non-determinism. Non-determinism in real-time specifications is a complex issue to handle. RT-MaC surmounts the problem by allowing the use of probabilistic properties, on which it applies statistical analysis.

The MaC framework uses two languages, PEDL and MEDL, to describe properties, section §3.2.2. PEDL is a low-level specification language thus it is tied to the implementation and requires no changes. On the contrary, MEDL is a high-level specification language and is used to define mathematical-like notation. MEDL lacks in supporting time-bound operators that allow quantitative properties to be defined. RT-MaC extends MEDL with time-bound opertors similar to Metric Temporal Logic (MTL).

RT-MaC is more expressive and flexible than D^{3}CA. However, the use of MTL operators and the lack of supporting domain specific notation leads RT-MaC to be more difficult to use.

Monitoring Algorithms for MTL Specifications. Thati *et. al.* [TR04] presents a solution for using MTL properties at runtime without having to store the execution trace. That is, events are consumed whenever they are raised therefore releasing the application from having to store the execution trace. This produce/consume procedure is similar to the parallel debugging project.

The MTL properties are translated into executable code through the use of a tableaux, where each operator is represented by an algorithm. The use of a tableaux and the nature of MTL properties leads the monitoring system to grow exponentially with the size and number of MTL properties. The exponential growth contrasts with the linear growth of D^{3}CA. D^{3}CA achieves a linear growth by converting the QDDC formulae into symbolic automata.

Jassda. The Java library implementing Design by Contract, Jass, lead reserchers to try to develop something more robust. The first attempt was JassTA (Jass with Trace Assertions), which like Temporal Rover and D^{3}CA, uses a pre-compiler to integrate specifications inside the execution code. However, as resulted from all the projects using a pre-compiler, the space and time requirements of the system under test increase drastically.

Jassda manages to reduce the space requirements of the observed system through the use of CSP channels that enhance the observing system with a suitable mechanism to describe concurrent systems. The CSP channels are integrated with the Java Debugger, hence, do not form part of the monitored system. A drawback of Jassda is that it does not provide time operators which are of critical necessity for checking real-time systems.

Runtime verification of Concurrent Haskell programs. Haskell is a lazy functional language that is hard to debug. The project proposes an LTL library which simplifies the debugging of concurrent Haskell programs [SH04]. The developer is in charge of defining the LTL properties as Haskell programs, and to insert verification points whenever they are necessary. Relying on the expressiveness of Haskell, the LTL library endows the developer with the ability to dynamically insert new LTL properties at runtime. The library in subject provides a powerful debugging mechanism, which on violation of an error returns a trace of the execution.

3.3 Monitoring-Oriented Programming (MoP)

A number of validation projects have been undertaken in the last decade. Each project provides its own mechanism and engine to integrate and evaluate formal specifications during the system execution. Chen and Roşu [CR03] suggest a generic and abstract framework, which attempts to simplify the integration of formal specifications written in any mathematical notation into monitoring code. The framework emphasises the concept of using monitors to validate program executions, hence, its name Monitoring-Oriented Programming (MoP).

MoP integrates the monitoring system without modifying the host language and compiler. This issue is similar to D^3CA and Temporal Rover but different from Java-MaC and Java PathExplorer that modify the compiler generated bytecode.

The integration of formal specifications as runtime monitors is likely to be required only during development and testing phases, and might not be present in the final product. To simplify the move from testing to release version, MoP suggests that formal specifications be introduced as annotations. An annotation is a tuple [CR03] consisting of

$$\{\text{logic specifier}, \text{logic specification}, \text{failure handler}\}.$$

The use of annotated tuples is similar to D^3CA and other validation tools, although the logic specifier entity in the annotation tuple is not required because they handle a single logic notation.

MoP concept enforces the need of a basic platform on which to define any logic notation. The platform endows the framework to be generic, flexible and provides a simple specification language, mainly primitive predicates and formulae. D^3CA also provides a similar platform by using a simple simulated Lustre environment that allows any logic notation to be encoded into symbolic automata with a predefined cost. MoP separates the monitoring code in two modules. The first module is the "code generator", which converts the supplied observation formulae into their representative code recognisable by the second module, the "logic engine". The code generator in D^3CA consists of two modules; the parser generating the abstract syntax tree of the formula that is used by Lustre to evaluate property validity.

MoP specifications separates the two concepts of generation and evaluation. The separation of concepts leads MoP to be more flexibile in the integration of new logic notations. MoP flexibility is furtherly enhanced by allowing the interaction between the code generators and logic engines free from any standardisation. D^3CA provides a similar separation by having the Lustre platform for low-level integration and the QDDC logic engine for defining the formal specifications.

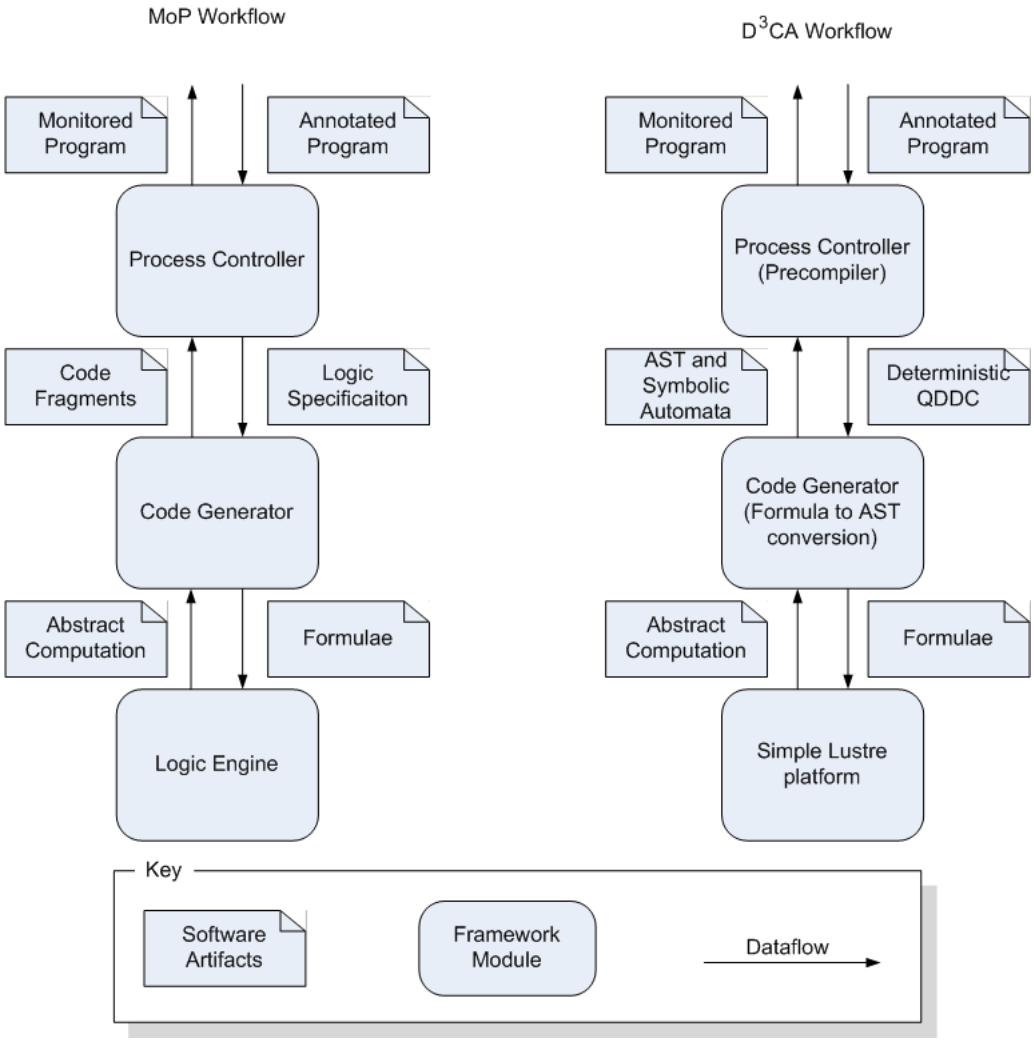


Figure 3.4: MoP [CR03] and D^{3}CA workflow comparison.

An important issue when using monitors for observing system behaviour is whether to use inline or offline monitoring. Inline monitoring is when the annotations are replaced with the monitoring program code by using the monitored system resources. The benefit of using inline monitoring is that the actual program state is used during the evaluation and the falsity of a property is immediately reported. The major disadvantage of inlining is that deadlocks or unexpected terminations are not triggered by the monitors. In most systems, especially embedded systems, the resources are limited and therefore inlining is infeasible. Offline monitoring allows the monitoring code to run as a separate entity from the monitored system. In offline monitoring the annotations are replaced with code that transmits the state to the monitoring program. An advantage of using offline monitoring is that it allows multiple programs to be checked simultaneously, which is ideal for concurrent systems where the execution path is non-deterministic. Nevertheless, it requires adequate program state representation in order to identify the instance when the program failure occurred. MoP does not specify whether the monitors generated by the code generators are inline or offline but rather leaves the decision to the person designing the code generator for the specific logic. The reason for no specification about the type of monitor is that each logic is specific to a particular domain, therefore whether inline or offline monitoring is chosen cannot be predetermined. D^3CA is specific to deterministic QDDC properties defined as inline properties. In order to use the determinstic QDDC presented later, Section §4.1, some of the notation semantics require to be changed.

The motivation behind MoP is to provide a general framework that can be considered as a light-weight formal method. Formal methods tend to analyse a system from all possible execution paths and view, therefore they are expensive and highly non-scalable. MoP provides a light version of formal methods where the analysis is performed on a single execution path and at runtime. The disadvantage is that although the verified path can be guaranteed to be correct, the system in general still cannot be guaranteed. On the other hand, scalability is obtained as a payoff of the approach.

From the above description of the MoP conceptual framework and its parallel relation with D^3CA, it is clear that the foundations of D^3CA maps exactly to the MoP concept, Figure 3.4. Therefore, D^3CA can be considered as a specific implementation of MoP with Deterministic QDDC as the testing logic notation.

Part II

Discrete and Deterministic subset of Duration Calculus Assertions (D^3CA)

Chapter 4

QDDC to Symbolic Automata

One of the solutions to model a system is by modelling it as automata. This chapter outlines the approach for defining a system model using automata generated using QDDC properties.

The first step towards defining runtime monitors, is to translate the formal specifications into automata. A common technique for defining (interval) temporal logic monitors as automata is to provide a number of transformation rules. The rules describe how the notation can be simplified in order to reflect program execution sequence. A detailed analysis of how logic notation semantics can be used to define the transformation rules is provided by Geilen [Gei01]. Following the principle proposed by Geilen and the work done by Gonnord *et al*, the monitors introduced in this dissertation are defined as symbolic automata. A symbolic automaton is a collection of formulae that represent the automaton states.

In section §2.2, Duration Calculus notation has been restricted to its decidable subset, QDDC. The non-determinism of QDDC makes it inappropriate for runtime monitors [GHR04] because handling non-determinism requires elaborated solutions. Non-determinism handling requires complex solutions because at a non-deterministic branch the correct execution path must be chosen before knowing what occurs in the future. Henceforth, to simplify the adoption and integration of QDDC monitors, the monitors presented are restricted to the deterministic subset of QDDC.

As to avoid QDDC non-determinism, we propose a further restricted Duration Calculus notation, which is deterministic and discrete. The obtained notation is defined in terms of QDDC and its execution semantics are provided in Lustre syntax. The synchronous dataflow programming language Lustre is used for the semantics as it allows resources required by monitors to be predetermined. The chapter provides a simple introduction to Lustre as to facilitate the understanding of the semantics presented.

4.1 Deterministic QDDC

The non-deterministic nature of QDDC requires the monitoring system to be equiped with knowledge about the future. During software executions states are created once they are entered, therefore, the past is known. The restriction of having only past values leads us to restrict QDDC to its deterministic subset.

It is necessary to define the concept of propositions that is the bases of the deterministic QDDC notation.

Definition 4.1. *(Propositions.)* Let us first define propositions as the bases of the notation. A proposition is a simple statement that can be either **true** or **false**.

$$
\begin{array}{rcll}
P & ::= & \textbf{false} & \\
 & | & \textbf{true} & \\
 & | & p & (p \in \text{variable} \mapsto \mathbb{B}) \\
 & | & \neg P & (\textbf{not}\ P) \\
 & | & P \vee P & (P\ \textbf{or}\ P) \\
 & | & P \wedge P & (P\ \textbf{and}\ P) \\
 & | & P \Rightarrow P & (P\ \textbf{implies}\ P) \\
 & | & P \Leftrightarrow P & (P\ \textbf{iff}\ P) \\
 & | & P \otimes P & (P\ \textbf{xor}\ P)
\end{array}
$$

The first restriction over QDDC is that the point interval, $\lceil P \rceil^0$, can only be used at the boundaries of intervals.

Definition 4.2. *(True at the beginning of the interval.)* A proposition, P, evaluates to true at the beginning of the interval.

$$\text{begin}(P) \stackrel{df}{=} \lceil P \rceil^0 \frown \boldsymbol{tt}$$

Definition 4.3. *(True at the end of the interval.)* A proposition, P, is true at the end of the interval.

$$\text{end}(P) \stackrel{df}{=} \boldsymbol{tt} \frown \lceil P \rceil^0$$

The QDDC length ($\eta \leqslant c$), the number of true evaluations of P ($\Sigma P \leqslant c$), constantly true for non-point intervals ($\lceil\!\lceil P \rceil\!\rceil$) are deterministic, thus they are left in our restricted subset. Here it must be noted that the set of comparison operators is restricted the less than ($<$), less than or equal to ($\leqslant$) and equal to ($=$) operators.

Definition 4.4. *(Age of a proposition.)* The age of a proposition, P, is determined by the number of consecutive true values at the end of the interval.

$$\text{age}(P) \leqslant c \stackrel{df}{=} \neg(\boldsymbol{tt} \frown \lceil\!\lceil P \rceil\!\rceil \wedge \eta > c)$$

Definition 4.5. *(Eventually true in the initial prefix.)* A duration formula is expected to become true during the initialisation of a subinterval.

$$\Diamond_b D \stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot D \widehat{\ } \boldsymbol{tt}$$

Definition 4.6. *(True for every initial prefix.)* A duration formula is expected to be true during the initialisation of a subinterval.

$$\Box_b D \stackrel{df}{=} \neg \Diamond_b \neg D$$

The chop operator in QDDC is non-deterministic as the interval division cannot be predetermined. Nevertheless, the chop operator can be used deterministically [GHR04]. The chop is deterministic when the interval division is determined by the first false evaluation of the right hand side expression.

Definition 4.7. *(then operator.)* The "then" operator provides a deterministic chop operator where the LHS of the operator is only executed on the first false occurrence of the RHS.

The definition of the "then" operator is composed of two expressions separated with the **or** operator. The first expression states that the formula D_1 is initially constantly true. After an interval of at least length 1 time unit, the formula D_1 evaluates to **false** for the first time. The instance when it evaluates to **false**, D_2 must evaluate to **true**.

In discrete-time intervals, the minimum length of an interval is of 1 time unit. Therefore, on the occasions where formula D_1 is **false** at the beginning of the interval, the first expression evaluates to **false** irrespective of D_2's value. To counter for these situations, the definition is extended with the second expression. The second expression states that if D_1 is initially **false**, then D_2 must be true for the entire interval.

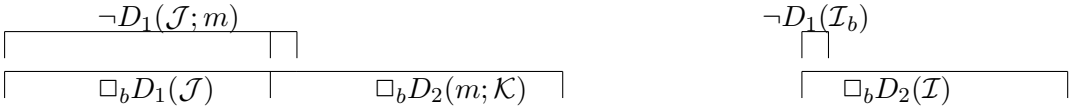


Figure 4.1: Block diagram for `then` operator

The "then" operator defined in QDDC terms can be misleading thus it is defined formally.

$$\begin{aligned} D_1 \, then \, D_2 \stackrel{df}{=} \lambda \mathcal{I} : Intv \cdot \exists \mathcal{J}, \mathcal{K} : Intv \cdot \exists m \in \mathcal{I} \cdot \\ \mathcal{I} = \mathcal{J}; m; \mathcal{K} \wedge \big(\Box_b D_1(\mathcal{J}) \wedge \neg D_1(\mathcal{J}; m) \wedge \Box_b D_2(m; \mathcal{K}) \big) \vee \\ \big(\neg D_1(\mathcal{I}_b) \wedge \Box_b D_2(\mathcal{I}) \big) \end{aligned}$$

Deterministic QDDC formulae can be combined using boolean logic operators.

Definition 4.8. *(Deterministic QDDC expression.)* An expression consists in the application of basic logic operators on deterministic QDDC formulae.
For any two formulae D_1 and D_2

$$\begin{aligned}
\neg D_1 &\stackrel{df}{=} \lambda\mathcal{I} : Intv \cdot \neg D_1(\mathcal{I}) \\
(D_1 \vee D_2) &\stackrel{df}{=} \lambda\mathcal{I} : Intv \cdot D_1(\mathcal{I}) \vee D_2(\mathcal{I}) \\
(D_1 \wedge D_2) &\stackrel{df}{=} \lambda\mathcal{I} : Intv \cdot D_1(\mathcal{I}) \wedge D_2(\mathcal{I}) \\
(D_1 \Rightarrow D_2) &\stackrel{df}{=} \lambda\mathcal{I} : Intv \cdot D_1(\mathcal{I}) \Rightarrow D_2(\mathcal{I}) \\
(D_1 \Leftrightarrow D_2) &\stackrel{df}{=} \lambda\mathcal{I} : Intv \cdot D_1(\mathcal{I}) \Leftrightarrow D_2(\mathcal{I})
\end{aligned}$$

A number of derivable operators can be introduced without loss of determinism. The first derivable operator is the "eventually true", whose semantics are restricted to the "eventually true in the initial prefix" as to keep determinism. Here it must be noted that although the "eventually true" operator can be defined with minor losses, the "always" operator cannot be introduced because it requires fresh counters to be created for each subinterval.

Definition 4.9. *(Eventually true.)* A deterministic QDDC formula must evaluate to true at least once within the interval.

$$\Diamond D \stackrel{df}{=} \llbracket \neg D \rceil^{\frown} \lceil D \rceil^{0} {}^{\frown} \boldsymbol{tt}$$

Definition 4.10. *(True for entire interval* $(\llbracket P \rrbracket)$*)* A state expression P holds for the entire interval including at the end point.

$$\llbracket P \rrbracket \stackrel{df}{=} \llbracket P \rceil^{\frown} \lceil P \rceil^{0}$$

Using the initial deterministic subset it is defined as,

$$\llbracket P \rrbracket \stackrel{df}{=} \llbracket P \rceil \vee end\,(P)$$

Definition 4.11. *(Must hold before* $(\lessdot)$*)* Given any two basic deterministic QDDC formulae F_1 and F_2, $F_1 \lessdot F_2$ states that F_1 must evaluate to true exactly one step before F_2 evaluates to true.

$$F_1 \lessdot F_2 \stackrel{df}{=} \left(\lceil F_2 \rceil^{0} \Rightarrow \lceil -F_1 \rceil^{0}\right)$$

Definition 4.12. *(For at least $\left(\overset{\delta}{\hookleftarrow}\right)$)* Given any two basic deterministic QDDC formulae F_1 and F_2, $F_1 \overset{\delta}{\hookleftarrow} F_2$ states that from the first instance formula F_1 becomes true and remains true then F_2 must also hold for at least δ steps.

$$F_1 \overset{\delta}{\hookleftarrow} F_2 \overset{df}{=} \Box(\lceil \neg - F_1 \rceil^{0\frown}(\llbracket F_1 \rrbracket \wedge \eta < \delta) \Rightarrow \llbracket F_2 \rrbracket)$$

Definition 4.13. *(Consecutive repetition.)* Consider a formula D that defines a cyclic behaviour. Given that the behaviour can be defined recursively then the formula can be said to apply for a number of times.

$$D^1 \overset{df}{=} D$$

$$D^k \overset{df}{=} D \, then \, D^{k-1}$$

Definition 4.14. *(Kleene Plus Closure.)* Sometimes it might be necessary that a duration formula is checked repeatedly for a number of times. The Kleene star closure specifies that a formula can hold for an unspecified number of consecutive intervals.

$$D^+ \overset{df}{=} \exists n : \mathbb{N} \cdot n > 1 \wedge D^n$$

4.2 Lustre environment

Lustre is a dataflow synchronous language [HCRP91]. Synchronous programming languages apply the *synchrony hypothesis*, which states that each reaction is performed instantaneously to external events [Hal98, BG92, Hou02]. In other words, the hypothesis states that programs and actions are considered to be atomic and take no time to execute.

Dataflow programming languages are ideal for implementing reactive and real-time systems because people from these fields are traditionally oriented in using network of operators for transforming flows of data [Hal98]. However, Lustre provides a restricted subset of dataflow primitives and structures. This subset consists of memory bounded constructs as to allow memory and time requirements to be predetermined and to be easily simulated in other languages.

For better understanding of algorithms presented in this report, a fragment of Lustre is presented in this section. It is beyond this section's scope to discuss Lustre as a programming language. Details on Lustre can be obtained from [HCRP91, Hal98, BCE$^+$03].

The synchrony hypothesis refers to time where the definition of time must include two aspects from the three charateristics of a chronological clock. A chronological clock provides partial ordering of events, simultaneity of events and delays between events [LG91]. Lustre

as a synchronous programming language provides a concept of clock that uses the program cyclic behaviour to define clock ticks (1 time unit). This type of clock is referred to as the basic clock. The binding of clock ticks with the programs cyclic behaviour allows the n *th* tick to provide the n *th* system state configuration, enforcing the no time concept of the hypothesis.

Lustre provides three basic data types: boolean, integer and real. Each variable is required to be declared and to be associated a data type on declaration. Expression variables are declared as "$X = E$" where X is the variable name and E the equation (expression) that provides the only possible definition for variable X.

A number of basic operators are provided, which include the traditional operators on data types, relational operators and the `if..then..else` conditional operator. The set of operators also contains some special operators, such as the "previous" and "followed by" operators amongst others.

Definition 4.15. *(Previous.)* The "previous" (`pre()`) operator returns the value of the state variable at the previous cycle except at the start of an interval where the previous value of a state variable is *undefined*. Formally,

$$\texttt{pre}(X) \stackrel{df}{=} \lambda t : \mathbb{T}_{\mathbb{N}} \cdot \begin{cases} X(t-1) & t > 0 \\ \textit{undefined} & t = 0 \end{cases}$$

Definition 4.16. *(Followed by.)* The second special operator is the "followed by" ($\rightarrow$). The operator is used to concatenate two streams together, where the first stream is taken as an initial value of the second stream.

$$X \rightarrow Y \stackrel{df}{=} \lambda t : \mathbb{T}_{\mathbb{N}} \cdot \begin{cases} X(0) & t = 0 \\ Y(t) & t > 0 \end{cases}$$

Example 4.17. *Consider a simple counter that add 1 to the previous value of a variable X. For the purpose of illustrating the effects of using the "followed by" and `pre()` operators, the previous value of X is initialised to* 1.

$X = 1 \rightarrow \texttt{pre}(X)+1$

cycle	*0*	*1*	*2*	*3*	*4*
X	*1*	*2*	*3*	*4*	*5*
`pre(`X`)`	undefined	*1*	*2*	*3*	*4*

Lustre allows well initialised memories to be used. To simplify the reading of algorithms, the `pre` operator is taken to be a well initialised. When the `pre()` is applied over a boolean variable then the initial value is taken as **false** while for integer variables the initial value is taken as 0.

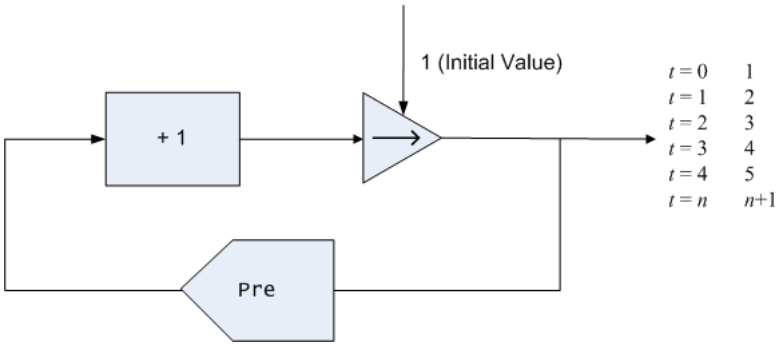


Figure 4.2: Counter example in Lustre with variable initialisation

In D^{3}CA , Lustre is used to simplify the validation process into an infinite loop over the sequence of dataflow primitives and structures. Also, by simulating Lustre in D^{3}CA allows the adoption of Gonnord *et. al.* transformations – from QDDC to symbolic automata [GHR04].

4.3 Helper functions

Programming languages, including Lustre, provide propositional logic, and some logic and numeric operators, which are enough to directly translate determinstic QDDC to their sematic code. Hence, a number of generic helper functions are defined as an aid to simplify the translation process.

Definition 4.18. *(*`after`*(p).)* An important function in interval temporal logic is to determine whether a state variable p is true or has been true at least once in the past. The `after`(p) is lazily evaluated as,

$$\texttt{after}(p) = \texttt{pre}(\texttt{after}(p)) \vee p$$

The `after` method returns true if the state variable P is **true** in the current clock tick. However, some properties might be required to hold in the past for the current proposition to hold. The `after` method would not directly suite this situation, therefore, a more restricting `after` is defined next.

Definition 4.19. *(*`strict_after`*(p).)* The state variable p has been true in the past.

$$\texttt{strict\_after}(p) = \texttt{pre}(p) \vee \texttt{pre}(\texttt{strict\_after}(p))$$

Definition 4.20. *(*`always_since`*(p, b).)* A state variable p has been true for the entire interval started at clock cycle b. The formula is lazily encoded as,

$$\texttt{always\_since}(p, b) = (\texttt{pre}(\texttt{always\_since}(p, b)) \vee b) \wedge p$$

Definition 4.21. *(*`first`*(p, b).)* The `first` method returns true only on the first oc-

currence of the proposition p from the start of interval, denoted as b. Defined in Lustre as,

$$\begin{aligned} \texttt{first}(p, b) \quad = \quad & \textbf{if } \texttt{after}\,(b) \\ & \textbf{then } p \wedge \neg(\texttt{strict\_after}(p)) \\ & \textbf{else } \textbf{false} \end{aligned}$$

The above helper functions all return either true or false, nevertheless QDDC notation makes use of summation hence requiring methods to return numbers.

Definition 4.22. *(*`nb_since`*(p, b).)* The first method required is to count the number of times a state variable is true within an interval. The counting is achieved as follows,

$$\begin{aligned} \texttt{nb\_since}(p, b) \quad = \quad & \textbf{if } \texttt{after}\,(b) \wedge p \\ & \textbf{then } \texttt{pre}(\texttt{nb\_since}(p, b)) + 1 \\ & \textbf{else } \quad \texttt{pre}(\texttt{nb\_since}(p, b)) \end{aligned}$$

Definition 4.23. *(*`age`*(p, b).)* The `age` method determines the number of consecutive clock cycles the state variable P is true.

The $\texttt{age}(p, b)$ operator is evaluated as,

$$\begin{aligned} \texttt{age}(p, b) \quad = \quad & \textbf{if } \texttt{after}(b) \wedge p \\ & \textbf{then } 0 \rightarrow \texttt{pre}(\texttt{age}(p, b)) + 1 \\ & \textbf{else } \quad 0 \end{aligned}$$

Definition 4.24. *(Rising Edge.)* The rising edge operator indicates when the state variable becomes **true**from **false**.

In Lustre syntax the function can be defined as,

$$\begin{aligned} \texttt{rising\_edge}(p) \quad = \quad & \textbf{if } \neg(\texttt{pre}(b)) \wedge p \\ & \textbf{then true} \\ & \textbf{else } \quad \textbf{false} \end{aligned}$$

Definition 4.25. Distance between identical events. Some safety requirements require that an event becomes **true** only if it has been **false** for a number of clock cycles.

The distance between identical events can be programmed as,

$$\begin{aligned} & \texttt{distance\_between\_events}(D, \delta, b) = \\ & \qquad \textbf{if } \texttt{rising\_edge}(D) \wedge \texttt{age}(\neg D, b) \geqslant \delta \\ & \qquad \textbf{then true} \\ & \qquad \textbf{else } \quad \textbf{if } \; D \wedge \neg(\texttt{after}(\neg D)) \\ & \qquad\qquad \textbf{then true} \\ & \qquad\qquad \textbf{else } \quad \textbf{false} \end{aligned}$$

4.4 Determinstic QDDC Operators Semantics

The logic tools provided by programming languages allows only evaluation of simple propositions. The following table shows how a proposition can be evaluated recursively in order to determine its truth value. The column $A_P(\mathcal{I})$ shows how the value of property P is obtained for interval $\mathcal{I}$, where in the case of propositions it is a point interval referring to the current clock cycle.

$\boldsymbol{P}$	$\mathbf{A}_P(\mathcal{I})$
p	`p`
$\neg P$	`not` $A_P(\mathcal{I})$
$P_1 \wedge P_2$	$A_{P_1}(\mathcal{I})$ `and` $A_{P_2}(\mathcal{I})$
$P_1 \vee P_2$	$A_{P_1}(\mathcal{I})$ `or` $A_{P_2}(\mathcal{I})$
$P_1 \Rightarrow P_2$	`not` $A_{P_1}(\mathcal{I})$ `or` $A_{P_2}(\mathcal{I})$
$P_1 \Leftrightarrow P_2$	(`not` $A_{P_1}(\mathcal{I})$ `and not` $A_{P_2}(\mathcal{I})$) `or` ($A_{P_1}(\mathcal{I})$ `and` $A_{P_2}(\mathcal{I})$)
$P_1 \otimes P_2$	(`not` $A_{P_1}(\mathcal{I})$ `and` $A_{P_2}(\mathcal{I})$) `or` ($A_{P_1}(\mathcal{I})$ `and not` $A_{P_2}(\mathcal{I})$)

Using propositions and the helper functions introduced in the previous section the transformation rules for the remaining deterministic QDDC notation can be defined.

Notation	$\mathbf{A}_{\text{Notation}}(\mathcal{I})$
$begin\,(P)$	$\texttt{after}(A_P(\mathcal{I})\ \texttt{and}\ \mathcal{I}_b)$
$\lceil P \rceil$	$\texttt{strict\_after}(\mathcal{I}_b)\ \texttt{and}\ \texttt{pre}(\texttt{always\_since}(A_P(\mathcal{I}), \mathcal{I}_b))$
$\lceil\!\lceil P \rceil\!\rceil$	$\texttt{always\_since}(P, \mathcal{I}_b)$
$\eta \leqslant c$	$\texttt{nb\_since}(\textbf{true}, \mathcal{I}_b) \leqslant \texttt{c}$
$\Sigma P \leqslant c$	$\texttt{nb\_since}(A_P(\mathcal{I}), \mathcal{I}_b) \leqslant \texttt{c}$
$age\,(P) \leqslant c$	$\texttt{age}(A_P(\mathcal{I}), \mathcal{I}_b) \leqslant \texttt{c}$
$end\,(P)$	$\texttt{after}(\mathcal{I}_b)\ \texttt{and}\ A_P(\mathcal{I})$
$D_1\ then\ D_2$	$A_{D_2}(\texttt{first}(\texttt{not}\ A_{D_1}(\mathcal{I}), \mathcal{I}_b))$
$D_1 \wedge D_2$	$A_{D_1}(\mathcal{I})\ \texttt{and}\ A_{D_2}(\mathcal{I})$
$D_1 \vee D_2$	$A_{D_1}(\mathcal{I})\ \texttt{or}\ A_{D_2}(\mathcal{I})$
$\neg D$	$\texttt{not}\ A_D(\mathcal{I})$
$\Diamond D$	**if** $\texttt{first}(D, \mathcal{I}_b)$ **then true** **else** $\texttt{pre}(\texttt{eventually}(D))$
$D_1 \overset{\delta}{\hookleftarrow} D_2$	**if** $\texttt{after}(\mathcal{I}_b)$ **and** $\texttt{age}(\textbf{true}, \mathcal{I}_b) < \delta$ **then** $\texttt{always\_since}(D_1 \wedge D_2, \delta)$ **else** $\texttt{pre}(\texttt{for\_least}(F_1, F_2, \delta))$
$D_1 \lessdot\!\!\sim D_2$	**if** $\texttt{first}(D_2, \mathcal{I}_b)$ **then if** $\texttt{pre}(D_1)$ **then true** **else false** **else** $\texttt{pre}(\texttt{hold\_before}(F_1, F_2))$

4.5 Conclusion

This chapter introduced a subset of QDDC notation restricted to deterministic operators, together with their execution semantics. The chapter does not provide any proofs related to the equivalence between our restricted notation and QDDC. The syntax of the synchronous data-flow programming language Lustre has been introduced as a background for better understanding of the notation semantics. Lustre notation is used extensively in the next chapters.

The next chapter describes how the execution semantics presented here are used to monitor systems during runtime.

Chapter 5

Validation Engine

This chapter builds on the deterministic QDDC introduced in the previous chapter by describing a validation engine over the notation. The purpose of the validation engine is to extract system states during runtime and determine their correctness. Correctness checking is performed on a state-by-state bases by comparing the state with the deterministic QDDC properties obtained from the specifications.

The validation engine is defined through three modules:

1. The initialisation of the deterministic QDDC formulae as symbolic automata;
2. The integration of properties inside the system code; and
3. The validation process during runtime.

5.1 Symbolic Automata Initialisation

Writing properties in mathematical notation is cleaner and easy when compared to nested function calls. To keep validation writing as simple as possible, the formal specifications are specified in a separate file using a suitable representation, for example using XML syntax.

The formal specifications are passed to the symbolic automata generator. The automata generator output consists in a symbol table and a collection of Abstract Syntax Trees (ASTs). The symbol table contains the variables used by the symbolic automata. While the ASTs store a representation of the properties.

The most practical way of evaluating formulae is by evaluating the simplest parts first followed by the application of operators over the results. This evaluation process continues to evolve until the result of the original formula is obtained. Figure 5.2 depicts the evaluation

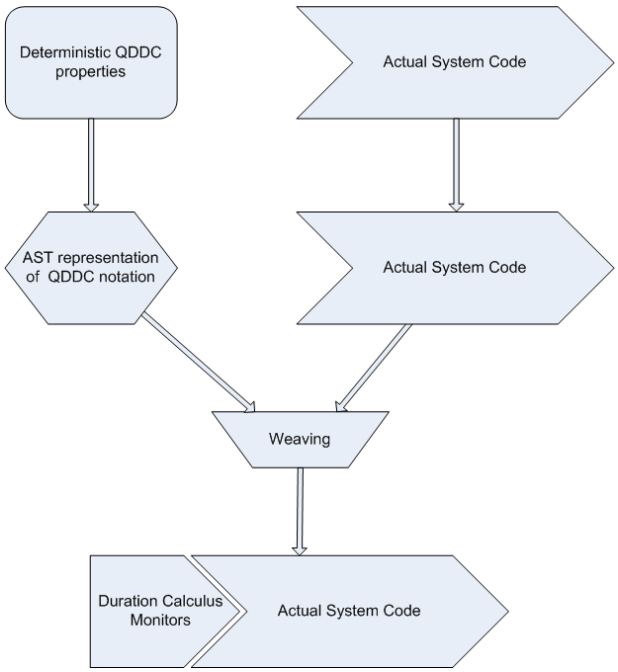


Figure 5.1: Abstracted view of the validation engine.

process of `begin`(P). From Figure 5.2 it is clear that ASTs are a suitable data structure for storing symbolic automata.

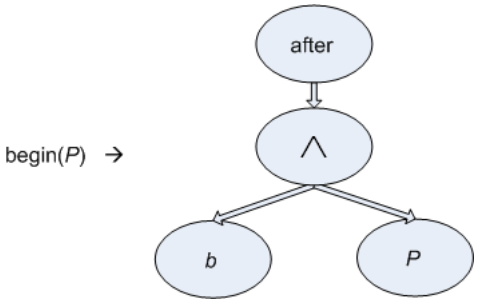


Figure 5.2: Evaluation decomposition of `begin`(P)

5.1.1 Space complexity

A formula represented in an AST consists in a number of operators, represented by non-leaf nodes, and propositions, the leaf nodes. Therefore, the amount of space required by the AST representation is $\mathcal{O}(n)$, where n is the number of nodes. The linearity in space complexity combined with the use of Lustre endows the validation engine with the capability of predetermining memory requirements.

Example 5.1. *Consider the simple property* `begin`(P)*. Using the transformation rules provided in Section §4.4, the property is transformed into* `after`($b \wedge P$)*. Thus the memory required is,*

$$
\begin{aligned}
\textit{space needed} &= \begin{cases} 1 & \textit{Boolean variable to store } \texttt{after} \textit{ result.} \\ 1 & \textit{To store the } b \wedge P \textit{ result.} \\ 2 & \textit{Boolean variables to store the propositions.} \end{cases} \\
&= 4 \quad \textit{Boolean variables}
\end{aligned}
$$

Boolean variables are typically represented using a byte[1]*, therefore, four bytes are required. Applying some simple optimisations, the space required can be reduced to three bytes by storing the* $b \wedge P$ *value directly in the* `after` *variable.*

Figure 5.3 illustrates the AST representing the formula.

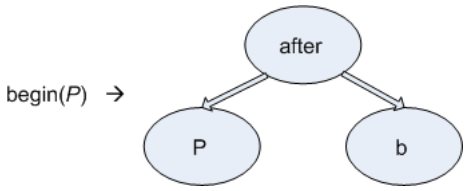


Figure 5.3: AST representation of `begin(`P`)`

5.2 Evaluation Process

The heart of validation is the process of verifying formal properties using the current system state. During the initialisation of the monitoring system the formal properties are converted into symbolic automata. The symbolic automata are stored as Abstract Syntax Trees (ASTs).

EVALUATE(*Symbolic Automaton*)
1 **for** each node starting from the leaf nodes
2 **do** expression variable ← evaluation node expression
3 Symbolic automaton validity result ← root node value

A practical way of evaluating a composite formula is by determining the values of atomic variables and then to apply operators over the obtained results. In the AST representation (Figure 5.2) the leaf nodes are the simplest to evaluate, as they are propositions. After evaluating the leaf nodes, the parent nodes of the leaves can be evaluated. Lines 1–2 perform the evaluation process starting from the simplest evaluations and move up along the tree hierarchy. The evaluation of the node expression is performed using the helper

[1]Some programming language defines boolean as bit, nevertheless, due to memory management techniques a byte gets reserved.

functions, of Section §4.3, over the Lustre environment, Section §4.2. The property validity is determined by the result of the root node evaluation.

Example 5.2. *Consider an evaluation tree representation of the property* $\lceil P \rceil$ `then end(`P`)`, *Figure 5.4.*

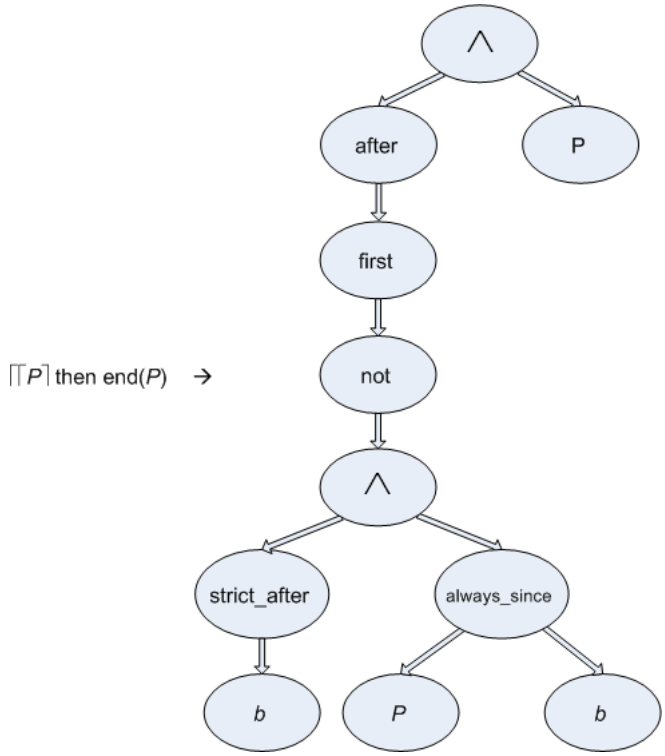


Figure 5.4: $\lceil P \rceil$ then end(P) – Evaluation tree representation

Starting the evaluation from the leaf nodes, the first evaluation to take place is to determine P's value. For the scope of the evaluation, the value of P is taken as being constantly ***true*** *and the evaluation is taking place after the beginning to the interval.*

Having determined the value of P as ***true****, the evaluation process moves to the parent nodes. The node at the first branch specifies that P must hold for the entire interval. Using the execution semantics of Section §4.4, the node must evaluate the expression*

$$\texttt{strict\_after}(b) \ \texttt{and pre}(\texttt{always\_since}(A_P(\mathcal{I}), b))$$

The function call $\texttt{strict\_after}(b)$ *returns* ***true*** *because it is not the first evaluation of the property.* $\texttt{always\_since}(A_P(\mathcal{I}), b)$ *also returns* ***true*** *as the value of P is assumed to be constant. By* **and***ing the obtained results, the satisfiability of property* $\lceil P \rceil$ *is determined. The next step is to evaluate the second branch. The evaluation of* `end(`P`)` *also results to be valid.*

Given the values of the two child branches, the evaluation of the root node can take place. The ***then*** *node is evaluated as* $A_{D_2}(\texttt{first}(\texttt{not } A_{D_1}(b, \mathcal{I})), \mathcal{I})$. *The expression* $A_{D_1}(b, \mathcal{I})$

is assigned the value of $\lceil P \rceil$*, which is* ***true****. Therefore,* `first(not` $A_{D_1}(b, \mathcal{I})$`)` *will return* ***false****. The* ***false*** *value indicates the second interval has not yet been started, therefore, the value of the root node is set to* ***indeterminate****. The value of* `end(P)` *associated with the* `then`*'s* A_{D_2} *parameter, is ignored. The result of the property satisfiability is* ***indeterminate*** *because the first interval is still being satisfied.*

5.2.1 Time complexity

The time complexity of the evaluation process consists on the number of nodes and the total number of functions called. On the assumption that each node visit consists in at least one function call then the time complexity is reducible to the number of function calls. This assumption is likely to hold because the leaf nodes are normally compensated by the indirect function calls.

$$\#(\text{node visits}) \leqslant \#(\text{function calls})$$

The system time complexity is effected by an $\mathcal{O}(f)$, where f is the number of function calls. Note that the evaluation time complexity also absorbs the cost of performing the function calls.

5.3 Validation Process

The validation process consists in two modules: the monitoring system, and a communication interface linking the validation engine and monitoring systems, Figure 5.5.

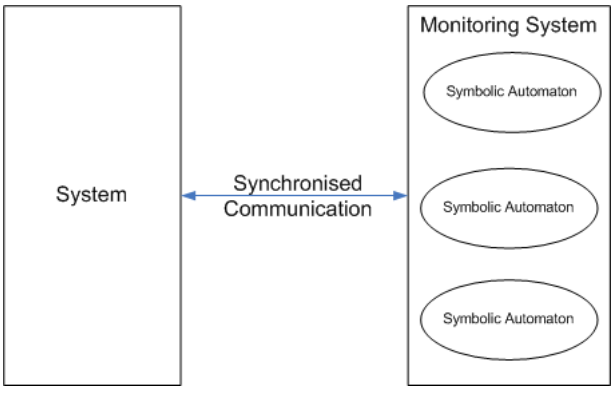


Figure 5.5: System composition diagram

5.3.1 Monitoring System

The monitoring system consists in two modules: a run-time checker and the Lustre environment. The run-time checker contains the list of the symbolic automata to be used by algorithm VALIDATE. The algorithm calls the evaluation process described earlier, which is executed on the Lustre environment.

```
VALIDATE
1  Stop system execution. // Required for variable integrity
2  Update non-expression variables
3  for each symbolic automaton
4        do Valid ← Evaluate(Symbolic automaton)
5           if Valid == false
6              then Error(Symbolic automaton)
7  Resume system execution. // On the assumption that the system
                                was not aborted due to errors.
```

The validation engine determines the current system correctness by analysing the current system state. The state analysis is achieved by having the non-expression variables, the leaf nodes, updated before computing the expression variables, the non-leaf nodes. During the update it is important that the constant variables get updated too as to reflect the current state with respect to the previous states, Figure 5.6. In algorithm VALIDATE the updating of variables is performed on line 2.

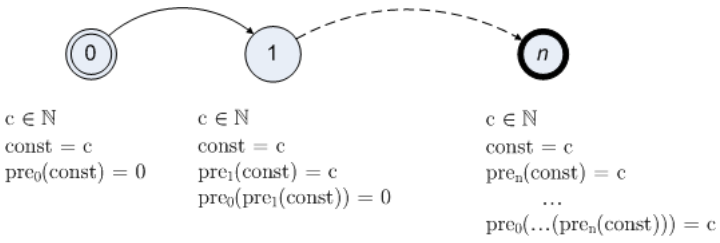


Figure 5.6: Lustre constant to system state relation

An important feature of validation is its ability to provide information about the system state violating the specifications. Lines 1 and 7 in VALIDATE ensures that if the system state is incorrect it is immediately trapped and reported before it propagates. The breaking of the system execution is also important to keep the validation variables integrity.

The core of validation is the evaluation of formal specifications over the current system state. The monitoring system performs validation by traversing the list of symbolic automata and for each automaton invokes the evaluation process, lines 3–6.

The monitoring system performs the above process for each new state encountered. The use of Lustre environment for performing the validation endows the monitoring system with the synchrony hypothesis. That is, assuming that the monitoring engine's time is based on the traversal of states than before performing the transition all properties has to be checked. Flowchart 5.7 shows the execution flow of the monitoring system.

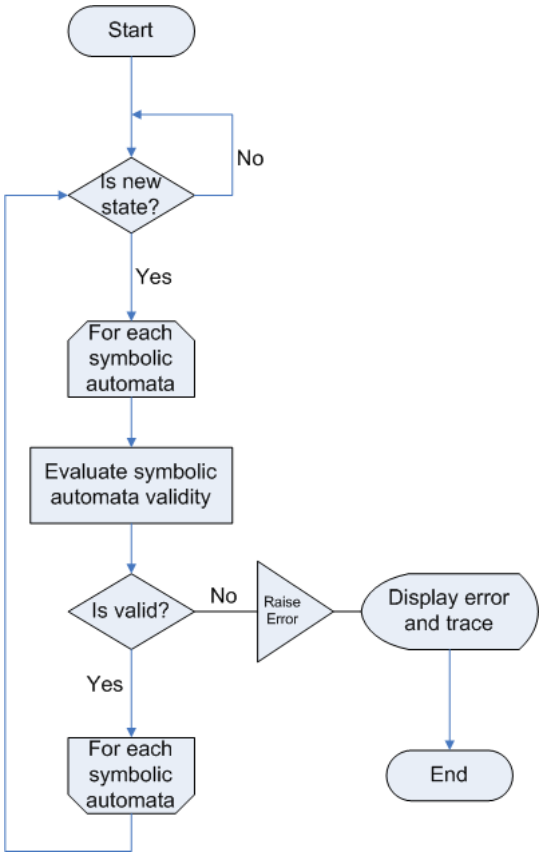


Figure 5.7: Validation process flowchart

5.3.2 Interface

The interface provides control over the monitoring system from the system under test. For an appropriate control over the monitored system, three control methods are required:

1. A synchronising call to the monitoring system to synchronise the state with the current system state;
2. Start/End the observation of a property; and
3. Update a variable to reflect new configuration.

The first control method, "synchronise", is the validation driving mechanism. Whenever the "synchronise" control is encountered during the system execution, the validation process is performed. The second control instructs the validation engine that it must start or stop

observing a property. The last control passes the system state information to the validation engine. The information passed is used during the variable update step, line 2 in algorithm VALIDATE.

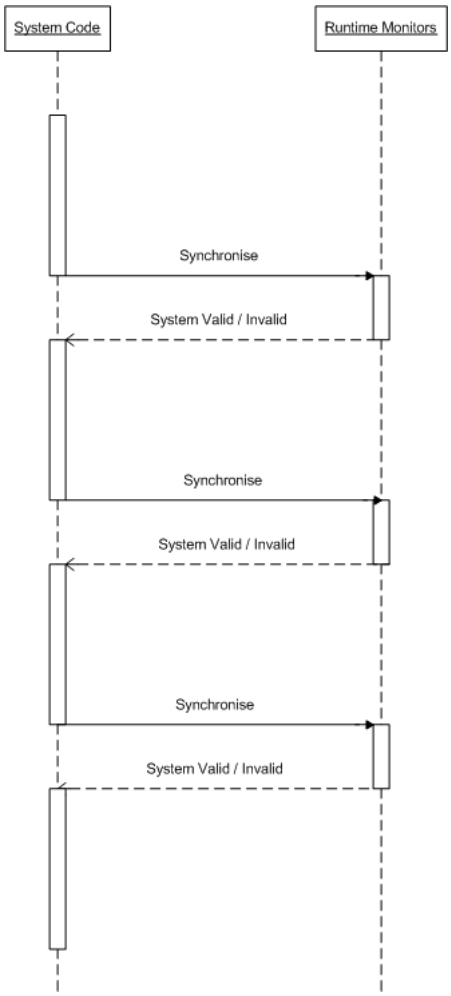


Figure 5.8: Sequence diagram illustrating flow control between the validation engine and the system.

5.4 Conclusion

This chapter detailed a validation engine for the discrete and deterministic subset of Duration Calculus. The validation engine has a space complexity of $\mathcal{O}(n)$ where n is the number of subformulae to be evaluated. The time complexity of the system consists in a number of iterations over the properties. Therefore, the time complexity is of $\mathcal{O}(n)$, where n is the number of function calls required to determine a property validity.

Although the logic notation used within this dissertation is specific for the synchronous evaluation, the validation engine presented is of generic nature.

Chapter 6

Prototypes of D^{3}CA

Chapters 4 and 5 provide a theoretical design for a validation engine whose properties are in discrete and deterministic subset of Duration Calculus. This chapter fills the gap between the theoretical part and the actual implementation of the validation system.

This chapter details two implementation approaches to the validation engine, a raw approach and a "three state" approach.

The first approach is implemented using the C++ language. The implementation highlights the difficulties that raise if the validation engine integration is performed as part of the system code writing. Another point highlighted in the implementation is the difficulties that raise when using the boolean value **false** to indicate that a property has not been *satisfied till now*.

After detailing the C++ implementation, the theoretical framework is reimplemented in C#. The new implementation uses 3-valued logic variables instead of the boolean variables. The C# implementation provides an additional tool, which abstracts the user from the validation engine function calls through the use of annotations by automating the weaving process.

Before starting detailing the implementations, the transformation of the mathematical formulae to the Abstract Syntax Trees (ASTs) data structure is provided in the next section.

6.0.1 Getting to AST

Abstract Syntax Trees (ASTs) are chosen because their structure perfectly reflects the way a property is evaluated. In ASTs the leaf nodes cannot be decomposed further, therefore, they are considered to be atomic, simple propositions. The relation between the AST data

structure and the way properties are evaluated is best described through examples. The ASTs of the examples are illustrated in Figure 6.1.

Example 6.1. *Consider the simple property* `begin`(P)*. The property is evaluated by first determining the truth of the propositions,* P *and* b*. The value of* P *is then passed to the function* `after`$(P \wedge b)$*, which will determine the property satisfiability.*

Example 6.2. *Now, consider the composite property* $\lceil P \rceil$ `then end`(P)*. The evaluation of the property is more complex due to the* `then` *operator. The* `then` *operator determines the satisfiability of the property by evaluating the RHS of the property, in our case* `end`(P)*. So the root of the AST has to be set to the return of* `end` *that according tot he transformation rules is the* `and` *of two functions. The RHS of the root node is simply evaluated by determining the value of state variable* P*. On the contrary the value of the LHS consists in the evaluation of a subtree. The root node of the subtree is the function call to* `after`$(b))$`, related to the initial (end`(P) *expression. Because the* `end`(P) *expression is called as a result of the* `then` *operator, the variable* b *is replaced by the call to the* `first` *function. The value of* `first` *is dependent on the value of expression* $\lceil P \rceil$*. Using the transformation rules expression* $\lceil P \rceil$ *is evaluated by* `and`*ing the result of two function calls –* `strict_after`(b) *and* `always_since`(P, b)*. Because expression* $\lceil P \rceil$ *is bound with the start of the interval, the variable* b *of the latter formulae indicates the start of the interval. Figure 6.1 shows the optimised AST generated for the property* $\lceil P \rceil$ `then end`(P)*.*

The above two examples provides the relation of the AST structure to the evaluation flow of the properties.

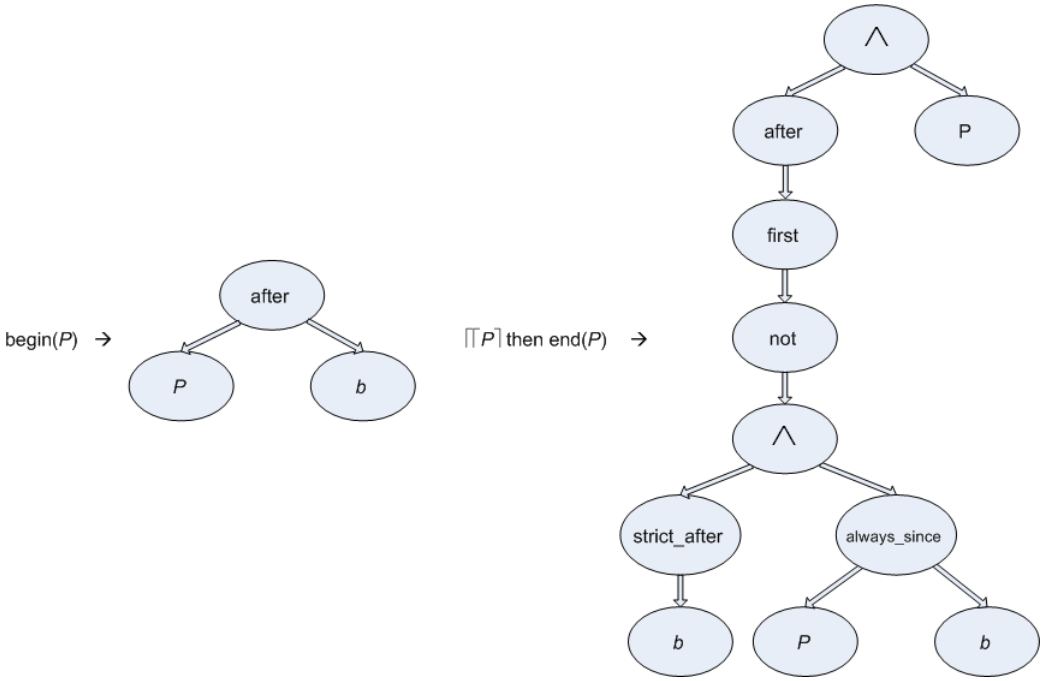


Figure 6.1: AST representation

The use of ASTs for property evaluation is more efficient in terms of performance. The conversion from the mathematical notation to ASTs is performed through the use of an external parser generator. The parser class is generated using the ANTLR Parser Generator [P+05]. The ANTLR Parser Generator is chosen because using the same grammar it is able to generate the parse code in different programming languages.

6.1 D^{3}CA – C++

The C++ version presented helps in understanding the drawbacks of a raw implementation. This prototype also intends to show places where optimisation can take place.

The next three sub-sections discuss the C++ implementation. The discussion commence with the description of the Lustre Environment implementation. It then proceeds by describing the validation engine and concludes with a brief analysis. The next section describes the second prototype.

6.1.1 Lustre Environment

The Lustre environment consists in a library providing the definition of basic datatypes and the two Lustre special operators, `pre` and `followed by`. To allow the use of `pre` over variables, the datatypes are extended with a linked list, where the head of the list refers to the previous value.

The three basic datatypes have the same structure, as the one depicted below. The data field "Value" is used to hold the current value associated with the variable. Accessing the data field to perform operations can be tedious, especially if the number of variables is high. The classes defining the Lustre datatypes override the C++ type operators to reduce the efforts required to perform operations over the "Value" field.

Data Structure "Lustre Integer":

int Value; {The value associated with the variable.}
Lustre Integer* Previous $\leftarrow$ ***nil***; {Pointer to the variable's previous value.}

The `followed by` operator is used to initialise a variable's value. In the C++ implementation the variables are initialised as classes, therefore, the `followed by` operator is hidden inside the datatype constructor. For example, the Lustre expression $1 \rightarrow X$ is programmed in Lustre as, `Lustre Integer` X `= new Lustre Integer`(1).

In Lustre the `pre` method is more like an operator, example `pre`(X). In our Lustre library

the operator is implemented as a method of the variable datatype, for example, $X \rightarrow$`pre`(). The result of the method call is an object of the same type as the variable. Using recursive calls, $X \rightarrow$`pre`() $\rightarrow$`pre`(), one can traverse the history of the variable.

The use of a linked list to hold the variable's history consumes space. To limit the space used for history records, the constructor of each datatype is extended by a parameter defining the number of previous values to be held. Example,

$$\texttt{Lustre Integer } X \texttt{ = new Lustre Integer}(1, 3)$$

states that the variable X is initialised to 1 and has a history log of up to 3 previous values.

6.1.2 Initialising

The initialisation of a monitoring property consists in converting the property to an AST, Section §6.0.1. During initialisation, the variables used by the evaluation process are stored into a symbol table. The symbol table consists in a vector holding a number of SYMBOL datatypes.

Data Structure "SYMBOL":

string Name; {The variable's reference name.}
short int Type; {The Lustre type of the variable.}
void* Data; {A pointer to the memory location holding the variable instance.}
void* Address; {A pointer to the memory address mapped to the variable.}

6.1.3 Validation Engine

The validation engine consists in an interface to the evaluation process. The symbolic automata evaluation is performed by traversing the AST structure in a bottom-up fashion. The ANTLR Parser Generator allows the tree traversal to be defined using the parser grammar. For the traversal class generation the parser file is modified to bind the logic function calls to the node visit algorithm.

The most important method call in the validation engine is `synchronise`. However, to compute the current state validity the state variables need to be updated. The user is supplied with a method that allows the variables to update their value. Before the `synchronise` method is called, the input and constant variables have to be updated by placing a call to the variable's `Update()` method.

The `synchronise` method is a direct implementation of the VALIDATE algorithm, described

in Section §5.3.1. In the evaluation loop the tree traversal method is called for every AST.

6.1.4 Analysis

The design of the validation engine and the C++ implementation are quite similar. The use of Abstract Syntax Trees (ASTs) for storing the properties resulted in an efficient way and easy way of performing validation.

Although the monitoring system is efficient there is still room for improvement. The first drawback in the implementation is the use of **false** to indicate undecidability. Although the drawback is not visible to the end user, the complexity of evaluating the properties validity effects the computation performance. A solution to surmount the problem is to use three-valued variables, as described in the design chapters.

Another major drawback in the approach is the manually insertion of variable updates before synchronising. In systems where the number of variables is high, the process of inserting the update method calls becomes intractable. The problem is easily avoided through the use of a pre-compiler, which allow annotations to be placed instead of the actual validation controls.

In the next section the same system is reimplemented using C#. The new implementation follows more strictly the design and implements the solutions presented above for the two drawbacks mentioned. Note that a new language is chosen as to evaluate the benefits and abstraction obtainable from using ASTs and Lustre as the validation implementation platform.

6.2 #D^{3}CA

The #D^{3}CA is a reimplementation of the C++ prototype, however, it is extended with a weaver and three-valued variables.

6.2.1 Weaver

The weaving process consists of two phases. The first phase generates a validation class based on the assertions, which are passed through an external XML file[1]. The validation class generated is then linked to the project during compilation. The second phase consists in parsing the system source code for annotations, defined later. The annotations found are replaced with the actual validation code. Figure 6.2 illustrates the D^{3}CA architecture.

[1]Refer to Appendix B for samples of the XML file format.

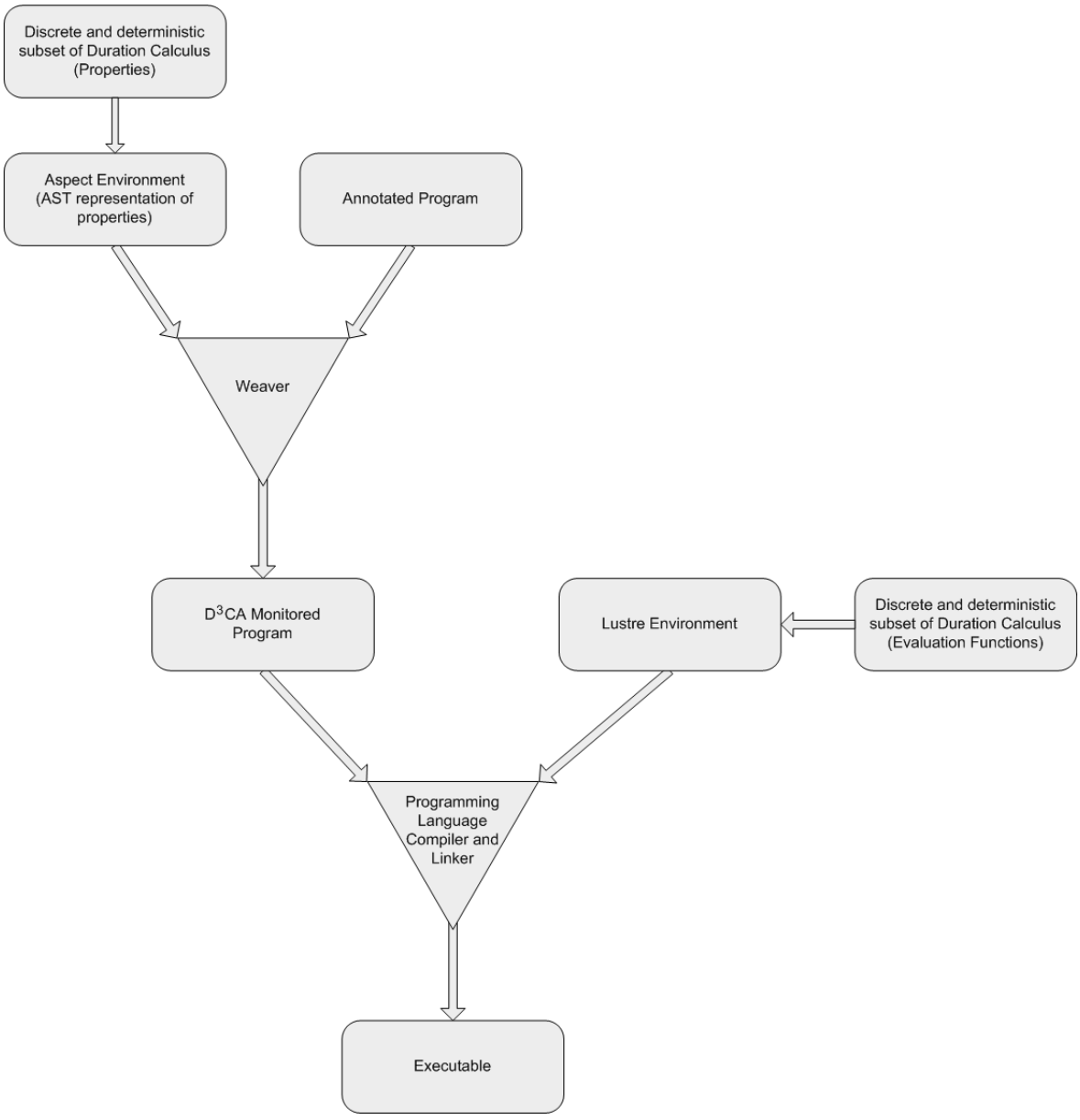


Figure 6.2: D^{3}CA Architecture Overview.

The generation of the validation class consists in copying the class template. However, the constructor is extended such that it contains the validation engine calls to convert the properties into ASTs. During the parsing of the XML file, the variables that are marked as constant or as effected from the system execution are listed into a text file. The generated text file is used during the second phase of compilation. Note that a text file is used because the first and second phase are performed using two separate programs.

The parsing of annotated files is performed using a simple algorithm, which reads the input on a line by line bases. When an annotation is read, it is replaced with the appropriate code. The most complex part of the weaver is to handle the variables update when the `synchronise` annotation is found. Using the text file generated during the first phase, the variables listed in the file are added for update.

6.2.2 Annotations

The use of a weaver allows the insertion of annotations to manage the validation code integration. The weaver is able to identify five different annotations that provide instructions about managing the assertion evaluation. Note that the assertions per sè are not integrated into the code, like the traditional assertions, but placed as part of the validation engine. The five annotations are:

1. <*validation_engine*: **bind** *variable_name variable_value*>
2. <*validation_engine*: **unbind** *variable_name*>
3. <*validation_engine*: **start** *assertion_name*>
4. <*validation_engine*: **stop** *assertion_name*>
5. <*validation_engine*: **synchronise** $\mathbb{B}$ >

The first parameter of all annotations identify the validation engine(s), which must receive the annotated instruction.

Localisation of variables in the system code is a problem when it comes to validation. The validation properties require to access the system variables values from different segments of code. A simple solution is to declare all the variables in the global scope but global scoping leads the software to loss maintainability and clearness. The solution adopted in the #D^{3}CA is to provide two annotations that allow validation variables to be bound to the system variable while it is in scope and to be unbound when the variable scope ends. The `bind` annotation takes two parameters, the name of the validation variable and the value to be assigned. The value can also refer to a system variable. The `unbind` annotation sets the variable to refer to its current value. Note that, the variable is still updated on every synchronisation to reflect the state transitions.

Interval Temporal Logic assertions require to define where the interval starts and ends. The user who adds the annotations is supplied with two annotations, a `start` and `stop`, to control when and where an assertion should be checked.

The most important annotation is `synchronise`. Whenever this annotation is encountered the weaver replaces it with the code to update the validation variables. The last code statement to be added is the call to the `synchronise` method defined by the validation engine. The annotation takes a boolean parameter which specifies whether the errors trapped should terminate the system execution.

6.2.3 Lustre environment

The Lustre environment consists in a library providing the basic datatypes definition and the two Lustre special operators, `pre` and `followed by`. The datatypes are inherited from a common interface, which allows basic operations to be performed without the need of type casting.

The three basic datatypes have the same structure, as the one depicted below. The data field "Value" is used to hold the current value associated with the variable and its type reflects the programming language related datatype, example **int** for **Lustre Integer**. Accessing the data field to perform operations can be tedious, especially if the number of variables is high. To lighten the use of the Lustre datatypes, the classes override the type operators.

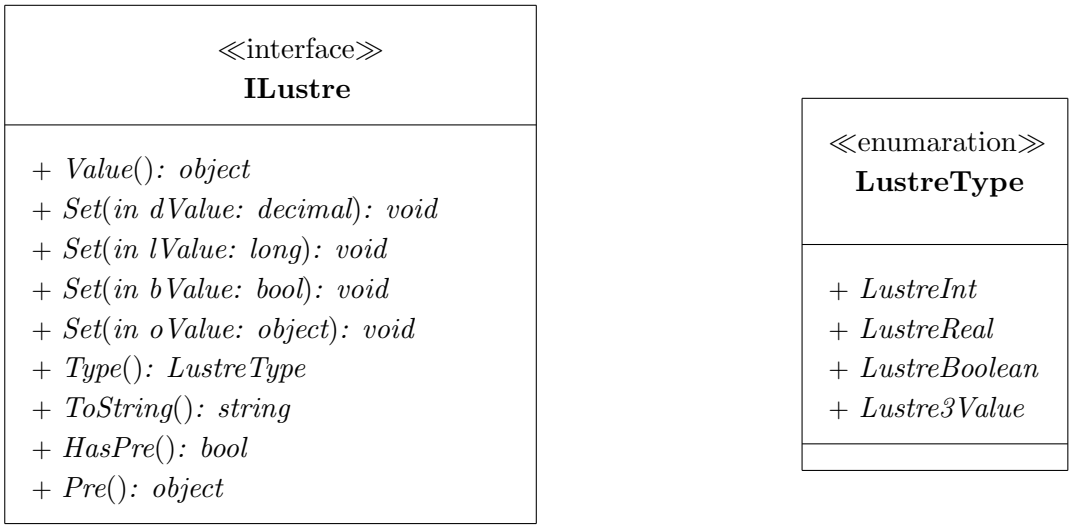


Figure 6.3: #D^{3}CA Lustre Interface and Enumeration UML Diagram

The Lustre environment is extended with a new datatype, 3-valued variables. The **ValueType** is an enumeration over the three possible values: **true**, **false** and **indeterminate**.

Data Structure "Lustre 3Valued":

ValueType Value; {The value associated with the variable.}
object Previous ← ***nil***; {Pointer to the variable's previous value.}

Like in the C++ implmentation the Lustre special operator, `pre`, is programmed as a method of the class. For example, to access the previous value of variable X the syntax is X`.Pre()`. The `followed by` is encoded as part of the variable type contructor.

6.2.4 Validation Engine

The validation engine consists in the combined implementation of the evaluation and validation processes described in Chapter 5. The evaluation of property validity is performed by executing the formula semantics on a Lustre environment. For efficiency reasons the formulae are stored as Abstract Syntax Trees (ASTs) which provide a clear flow structure to represent how the property is evaluated. As described in algorithm EVALUATE, on page 53, ASTs are traversed bottom-up because the leaf nodes consist of atomic propositions or numeric constants. On node traversing the Lustre environment is called to evaluate the semantics related to the operator node. The current validity of a property is determined by the result of the root node.

The validation process provides a loop over the ASTs and is responsible of reporting violated properties to the user. The `synchronise` boolean parameter is used by the validation method to control the action taken when an error is trapped.

6.3 Analysis

Through the use of ASTs as a data structure for storing formulae and Lustre as a platform for the evaluation of properties, D^{3}CA prototypes achieve the benefits of implementation independence from the logic point of view. The Lustre and AST methods need to be defined in the host programming language, nevertheless, new logic operators can be introduced without any need of further modification in the validation mechanism.

The AST creation and traversing mechanisms are generated using the ANTLR Parser Generator [P$^+$05], which through the use of a language definition tag is able to use the same grammar to generate classes in different languages. This leads the system to be more portable to different programming languages.

The Lustre environment provides a suitable abstraction from the programming language. By implementing the few Lustre methods in the desired programming language, the same

validation mechanism can be used without any need for modifications. The validation engine can also be less dependent on the programming language if it is defined as a shared library on Unix or a DLL in Windows.

The pre-compiler introduced in the C# prototype is the only programming language dependent module. The use of a pre-compiler and annotations for inserting the monitoring system provides a clean separation between the validation and the actual system code.

The two prototypes are far from perfect. One of the main drawbacks is in how the validation variables are created. When the system creates a new internal variable, it is labelled according to the information it is holding. This is a problem if the system requires to check different copies of the property because there is only one global symbol table and the variable will be created only once. Hence, the integrity of the validation states is lost.

Another drawback in the approach is that it lacks support for concurrent systems. When the validation starts its process only the execution branch calling the engine is stopped. This might lead to deadlocks if the branch being checked is holding a lock when an error is trapped.

Altough the implementations have some drawbacks, the design is more generic. The validation design provides a clean and easy solution for using formal specifications as runtime monitors over a system. An important feature provided by the validation system presented is that the space and time requirements for validation can be predetermined.

Chapter 7

Case Studies

Another aim of the dissertation is to show how systems can be observed through QDDC properties. The generic implementation of the D^3CA framework allows various scenarios to be observed.

This section presents two real-life scenarios, a mine pump alarm system and a simple answering machine. Throughout the dissertation, two simulation programs (one for each system) are used. Simulating a system gives various advantages such as the ability to introduce errors, pause the system to analyse the current state and control the environment.

The mine pump alarm system is common example [Pan00, GHR04, Jos01]. The purpose of the system is to test the state verification aspect of determinstic QDDC properties.

The answering machine system is used to show the expressiveness of the deterministic QDDC notation over state transitions which describe the assumptions on the system execution. The answering machine system shows that deterministic QDDC properties are expressive enough to monitor these assumptions.

7.1 Mine Pump

A widely used example in Duration Calculus literature is an automated alarm system for a mine pump. This case study shows the expressiveness of the discrete and deterministic Duration Calculus presented earlier. Figure 7.1 shows a graphical representation of the mine pump context as assumed in the system specifications[1].

A mine has a leakage of water (H_2O) and methane (CH_4) that in certain circumstances become a treat to the workers. The water leakage is collected in a sump on which a pump

[1]The figure is adopted from the work by Mathai [Jos01]

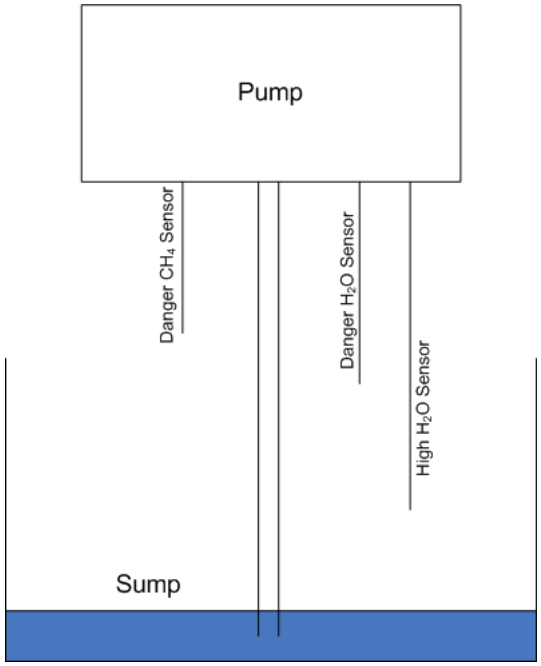


Figure 7.1: Mine Pump Diagram

is operated to lower the water level when it reaches critical levels. To operate the pump safely, the methane level has to be low otherwise it will be dangerous for the miners. In situations where the pump cannot be operated an alarm is activated to prepare the miners for evacuation.

7.2 QDDC Specifications

A number of properties can be defined for the mine pump operation. The pump is operated if the level of water is high and the amount of methane permits the operation. Let assume that the pump takes δ time to start pumping water. Then the condition to start the pump is,

$$\llbracket \text{High } H_2O \wedge \neg \text{High } CH_4 \rrbracket \xrightarrow{\delta} \llbracket \text{Pump On} \rrbracket$$

Informally, the condition says that given the water level has been high for the last δ time and the methane is under control then the pump should be pumping water out. The condition above is not enough on its own to guarantee that the pump is operating correctly as it does not check that the pump is really pumping water out. The pump takes at most ϵ time to lower the water level.

$$\llbracket \text{Pump On} \rrbracket \xrightarrow{\epsilon} \llbracket \text{Low } H_2O \rrbracket$$

Given the condition to operate the pump and the condition to verify the pump operation, another condition has to be placed to stop the pump. The pump is stopped in two situations:

either the level of water has lowered and there is no need to pump further water or the methane level has increased.

$$\llbracket \text{Low } H_2O \vee \text{High } CH_4 \rceil \xrightarrow{\delta} \llbracket \neg \text{Pump On} \rceil$$

Monitoring the pump is not sufficient since the environment provides no guarantees. To validate the system completely it is useful to place some checks on the environment. The first set of conditions define the assumptions related to the water levels. The pump takes some time to start and it is important to provide some time variation due to methane release. Therefore, the water level to start the pump has to be lower than the dangerous level.

$$\llbracket \text{High } H_2O \rceil \overset{\omega}{\hookleftarrow} \llbracket \neg \text{Dangerous } H_2O \rceil$$

The above property states that for the first ω time the high water level should not be dangerous for the workers. In relation to the last property another condition can be placed that verifies that when the water level is dangerous than it must also be high. This property provides a check on the water level sensor.

$$\Box(\llbracket \text{Dangerous } H_2O \Rightarrow \text{High } H_2O \rrbracket)$$

The miners work on three shifts and the objective of the mine pump system is to guarantee that only one shift is lost over a large number of shifts, say 10,000 shifts. This condition assumes that methane is released infrequently and that the distance between two consecutive methane releases is at least ζ time.

$$\Box(\lceil -\text{High } CH_4 \rceil^0 \widehat{} \llbracket \neg \text{High } CH_4 \rceil \widehat{} \lceil \text{High } CH_4 \rceil^0 \Rightarrow \eta > \zeta)$$

The methane release is taken to be of short duration and that it takes a short time to disperse in air.

$$\Box(\llbracket \text{High } CH_4 \rrbracket \Rightarrow \eta < \kappa)$$

The pump and the environment are under control. However, the pump configuration has an alarm that alerts the miners in case of danger. The alarm is activated when the water level is dangerous or the methane level is high for some time.

$$\llbracket \text{Dangerous } H_2O \rrbracket \xrightarrow{\delta} \llbracket \text{Alarm On} \rrbracket$$

$$\llbracket \text{High } CH_4 \rrbracket \xrightarrow{\delta} \llbracket \text{Alarm On} \rrbracket$$

Like with the pump, the alarm must be checked that it turns off when the danger has

passed. The alarm is turned off once the water level and the methane level are no more dangerous.

$$[\![\neg \text{Dangerous } H_2O \wedge \neg \text{High } CH_4]\!] \xrightarrow{\delta} [\![\neg \text{Alarm On}]\!]$$

7.2.1 Creating the validation engine

The validation engine is described by rewriting the properties and constraints in deterministic QDDC using XML syntax. Appendix B lists the properties as supplied to the #D^{3}CA. The XML file is passed through the AST generator module of the validation engine that creates the respective AST representations inside a C# class.

The next step is to insert the `synchronise` points inside the mine pump code. The artefact produced is a simulation, thus, the user is left to define a `synchronise` by forcing the system into a new state. Referring to Figure 7.2, a set of buttons are provided in the lower left part of the window to allow the user to instruct the simulation to advance a number of steps. For each step the `synchronise` method is invoked.

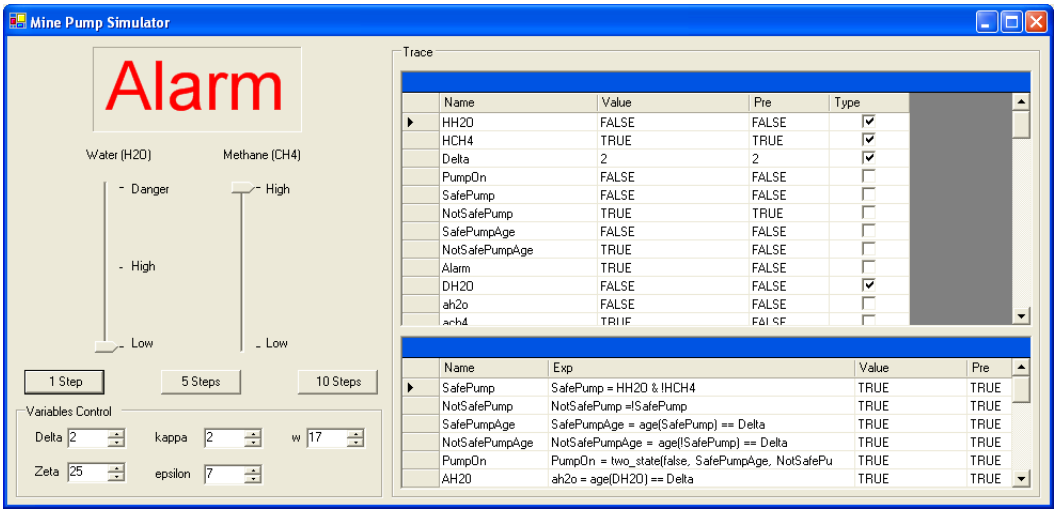


Figure 7.2: Mine Pump Simulation

The annotated source code and the generated C# class are compiled and link together. The final result is a simulation to test the mine pump properties.

7.2.2 Simple Test Scenario

The test scenario presented forces the mine pump methane environment assumption to be violated. This shows that the mine pump properties are enough powerful to capture changes in the environment. This case study also shows that the validation engine captures error on occurrence.

The test environment variables are initialised to:

Variable	Value
δ	2
ϵ	7
κ	2
ω	17
ζ	25

The mine has been operating fine until methane level has raised high. On normal circumstances the level of methane settles back to safe after 2 steps. For the purpose of simulation, we decided that the methane level should stay high for 5 steps. On the 3rd iteration over the properties the environment assumption about the methane release is violated. The D^{3}CA validation engine traps the error as soon the property returns **false**.

On trapping the error the validation engine generates a report, which displays the property that evaluated to **false**. In the scenario above the property reported is

$$\Box(\llbracket \text{High } CH_4 \rrbracket \Rightarrow \eta < \kappa)$$

Together with the property, the values of each subexpression are returned to allow the user to trace the origin of the error.

Expression	Value
High CH_4	True
$\llbracket$High $CH_4\rrbracket$	True
η	3
κ	2
$\eta < \kappa$	False

Analysing the reported results, it is clear that the property returned **false** because the expression $\eta < \kappa$ wasn't satisfied. Using the results of η and κ, one can determine that the length of the interval was longer than expected.

7.3 A Simple Answering Machine

The mine pump case study covers a lot of the D^{3}CA features. However, a less complex case study is useful to further understand the power of the validation engine. The operation of a simple answering machine can easily be represented as an automaton, Figure 7.3. In this case study the observation properties are placed over the state's transitions.

A characteristic of the answering machine operation is that the time intervals are measured

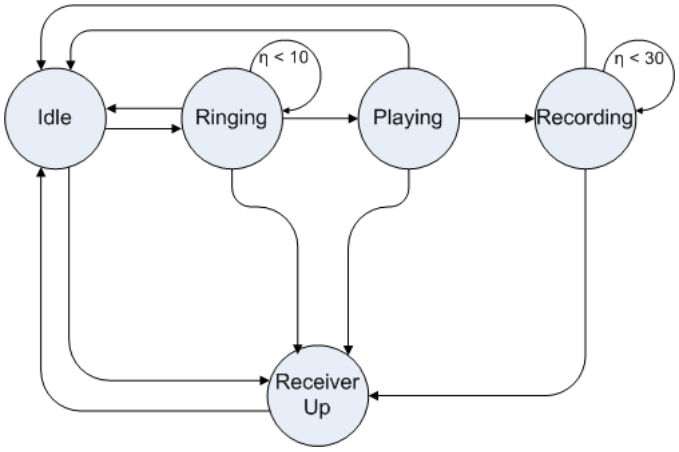


Figure 7.3: Answering Machine State Diagram

using different metrics. For example, the interval of the "ringing" state is determined by the number of tone rings. While the "recording" interval is measured in seconds. A feature of D^{3}CA is that metrics are irrelevant, since it all depends on the `synchronise` annotation. Note that this feature is only applied because the answering machine flow is sequential, that is, there is no concurrency in the execution.

7.3.1 QDDC Specifications

The simplest operation in the answering machine is that it is "idle". In other words, it is not performing any tasks.

$$\text{idle} = \neg\text{ringing} \wedge \neg\text{playing} \wedge \neg\text{recording}$$

When an incoming call arrives, the telephone starts ringing, therefore the machine enters the "ringing" state. There are two possible situations that can arise: either the call is answered by picking the receiver or the telephone continues to ring. In the later case, the length of the interval will determine when the answering machine should start operating by playing the recorded message. The transition from the "ringing" state to the "playing" state is defined as,

$$[\![\text{ringing}]\!] \Rightarrow \eta = 9^\frown[\![\text{playing}]\!]$$

When the recorded message is played the next task of the answering machine is to start recording the caller message. The message left by the caller is expected to be of at most 30 seconds.

$$[\![\text{playing}]\!]^\frown[\![\text{recording}]\!] \Rightarrow \eta \leqslant 30$$

Combining the operating states of the answering machine, a single property can be placed

over the operation flow of the answering machine.

$$\text{operating} = (\llbracket \text{idle} \rrbracket \frown \llbracket \text{ringing} \rrbracket \Rightarrow \eta = 9 \frown \llbracket \text{playing} \rrbracket \frown \llbracket \text{recording} \rrbracket \Rightarrow \eta \leqslant 30)^{+}$$

The property states that the answering machine can be considered as working well if it loops indeterminately over the operation state. The above formula doesn't capture constraints in the transitions. Therefore, three additional constrains are introduced:

1. The "ringing state" can only be entered from the "idle" state.

$$\text{idle} \lessdot \text{ringing}$$

2. The recorded message is only played if the telephone has been rang 9 times.

$$\text{ringing} \wedge \eta = 9 \lessdot \text{playing}$$

3. The caller message is recorded only after the answering machine message has finished playing.

$$\text{playing} \lessdot \text{recording}$$

The last assumption is that when the receiver is up the answering machine is in the "idle" state, in other words inoperative.

$$\llbracket \text{receiver up} \rrbracket \Rightarrow \llbracket \text{idle} \rrbracket$$

The pulling up of the receiver at any time interrupts the operation. Therefore, the answering machine is said to be working correct according to the specifications if it is either operating or the receiver is up.

$$\llbracket \text{operating} \vee \text{receiver up} \rrbracket$$

7.4 Creating the validation engine

The validation engine is described by rewriting the properties and constraints in deterministic QDDC using XML syntax. Appendix B lists the properties as supplied to the #D^{3}CA. The XML file is passed through the AST generator module of the validation engine that creates the respective AST representations inside a C# class.

We need to trap errors as soon they occur. Since each call to the `step()` method associated with the step buttons determine a clock tick, the `synchronise` method is attached to end of the `step()` method call. The answering machine code and the generated C# class are compiled and linked together to generate the simulation system, Figure 7.5 (Page 78).

7.5 Simple Test Scenarios

The properties that check the transitions can easily be violated, especially those containing the "holds before" operator. For example, the property

$$\text{ringing} \wedge \eta = 9 \mathrel{\prec\!\!\!\!\sim} \text{playing}$$

is violated when a step is performed with the "playing" option ticked and the current system is "idle" or "recording". The error report generated for this scenario is shown in Figure 7.6. The error states that the answering machine has entered the "playing" state when the telephone was not ringing or before the "ringing" interval has elapsed.

The scenario illustrated by the sequence diagram, Figure 7.4, show the answering machine operation.

The answering machine is "idle", that is, it is waiting for an incoming call. After a while a new call arrives, thus the phone starts ringing. On the first ring the `synchronise` method is called. The properties are evaluated and system state is updated to "ringing". Eight other `synchronise` methods are called and the system has not changed from "ringing", therefore the deterministic QDDC property

$$\texttt{age}(\text{ringing}) < 10$$

related to the QDDC property

$$[\![\text{ringing}]\!] \Rightarrow \eta = 9$$

is still valid. On the next ring the answering machine starts operating and its state changes to "playing". A `synchronise` call is called and this time the state has changed to the "playing" so the properties are still valid. At the end of playing the message a new `synchronise` is signalled to the validation engine, which checks that the answering machine started to record the caller message. A number of `synchronise` method calls are performed as seconds elapse. When the line drops or the 30 seconds of recording elapse the system state returns to "idle".

This time the answering machine owner decides to perform a call and pulls up the receiver. The event of lifting the receiver invokes the validation engine to check that the system is "idle" and the receiver is not in place. When the owner hangs up the validation engine is invoked to verify that the answering machine has remained "idle" and that the line has been dropped.

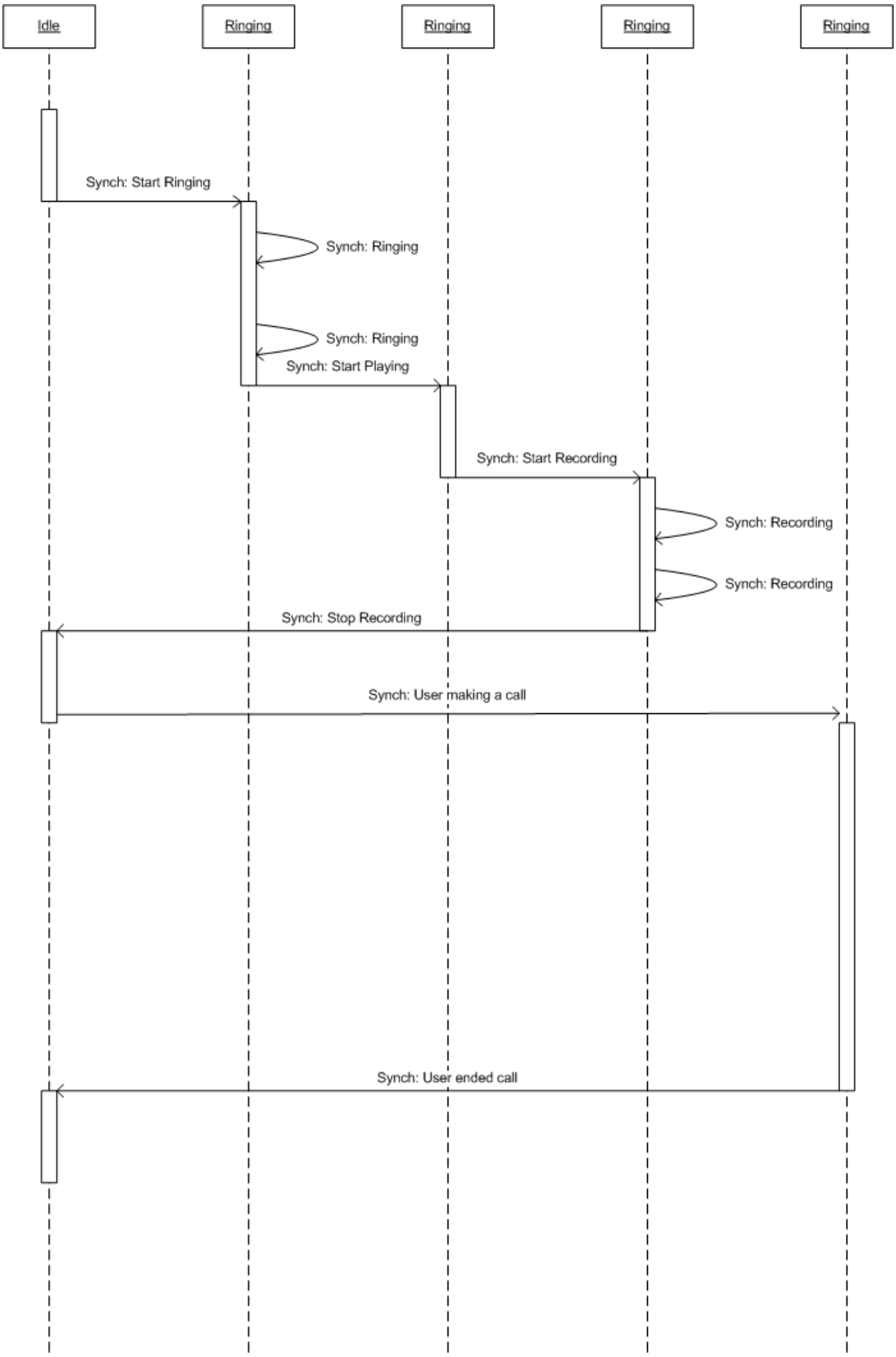


Figure 7.4: A Simple Answering Machine – Sequence Diagram

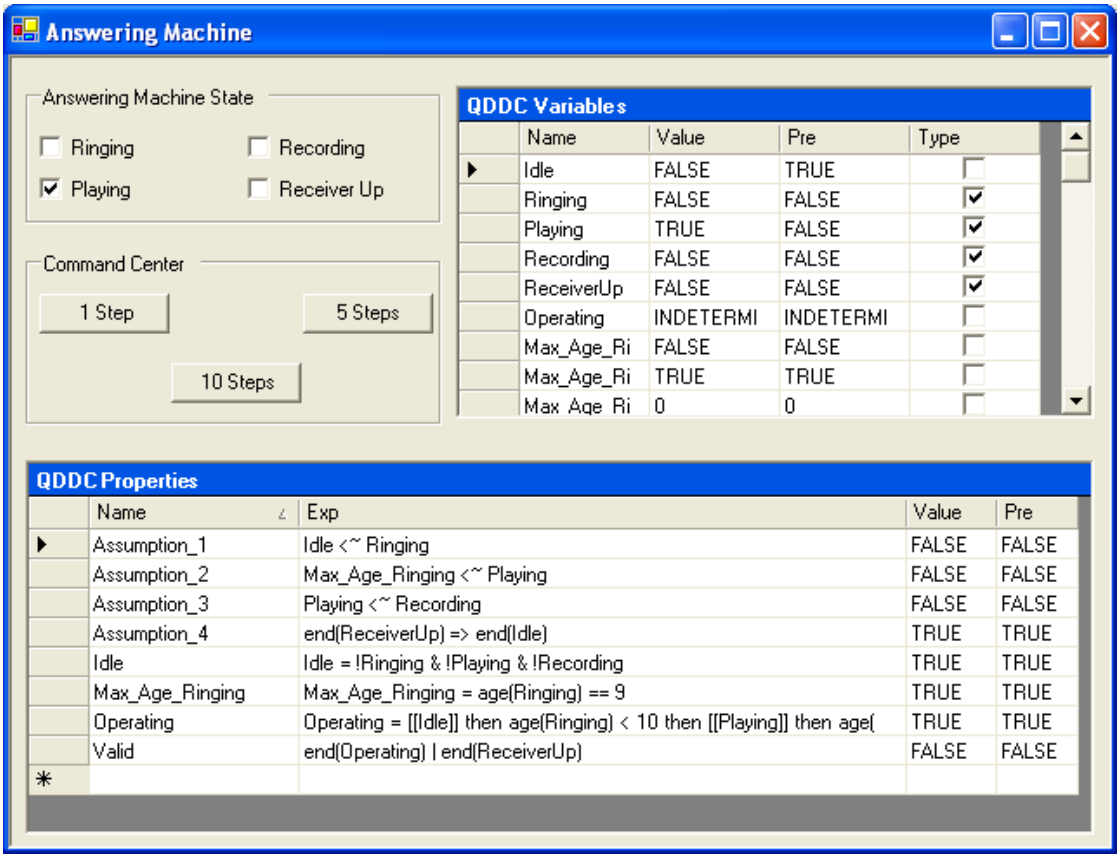


Figure 7.5: Answering Machine Simulation

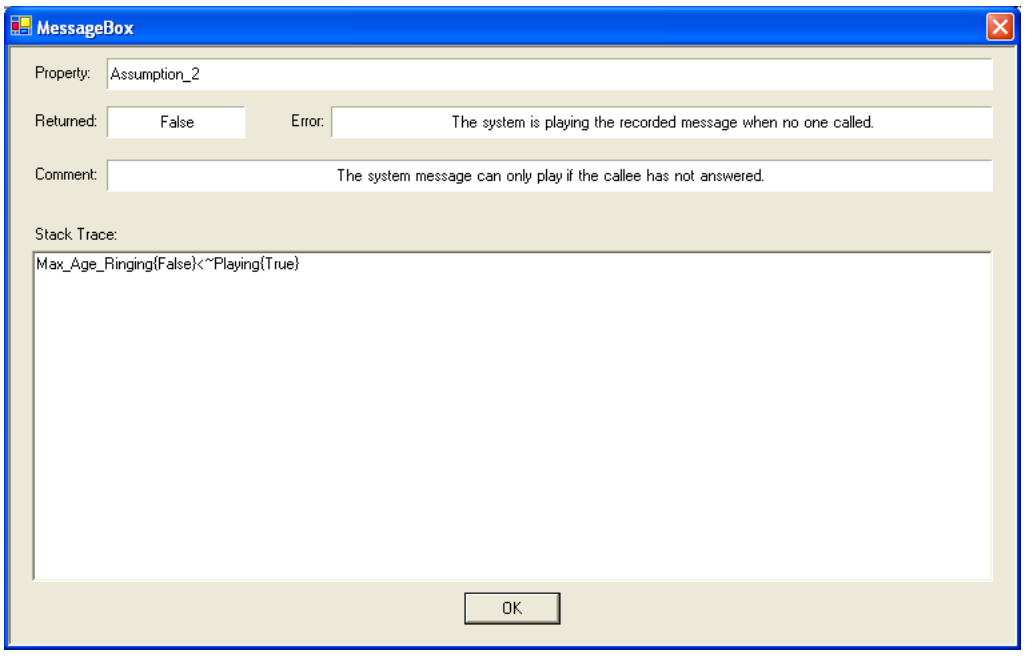


Figure 7.6: Answering Machine Simulation – Error Reporting

7.6 Conclusion

The case studies show that deterministic QDDC is powerful enough to capture a large number of real-life properties. The studies are based on real applications, however, due to time limitation a simulation had to be created.

The test scenario in the case study provides a small overview of how and when the validation engine determines that a property is violated. The case studies shows that properties described in QDDC can be translated in their respective deterministic formulae. The validation engine is enough powerful to trap behavioural errors according to the specifications.

Part III

Validation and Aspect-Oriented Programming

Chapter 8

Interval Temporal Logic Validation as an Aspect

Aspect-Oriented Programming (AOP) originated in 1997 at Xerox PARC as a solution for handling design concerns that cannot be clearly encapsulated using programming languages, both procedural and object-oriented [KLM+97]. These types of concerns are referred to as *aspects*. Aspects are different from normal procedures because they have *crosscutting* semantics. In simpler words, their execution depends on different parts of the software architecture.

The concept behind the AOP framework can be summarised as "In program P, whenever condition C is encountered, perform action A" [FF05]. The "condition C" refers to annotation marks, also known as *point-cuts* in AspectJ [Asp], while "action A" is the execution of aspect A. The application of the concept is performed using five components [KLM+97]: (i) a component language (procedural or object-oriented programming language), (ii) an aspect language, (iii) an aspect weaver, (iv) a component program (the system source code without any aspects – traditional way of programming), and (v) one or more aspect programs (code for aspects). The *aspect weaver* is a pre-compiler which given an annotated component program and a collection of aspect programs, replaces the annotated conditions with calls to the respective aspect program. The AOP framework flow is depicted in Figure 8.1.

The definition of an aspect as given in the original paper [KLM+97] is vast and ambiguous, which can lead normal software composition to be defined as aspects. Filman and Friedman [FF05] extend the definition of an aspect with two properties:

1. obliviousness; and
2. it must not be applicable to a single place.

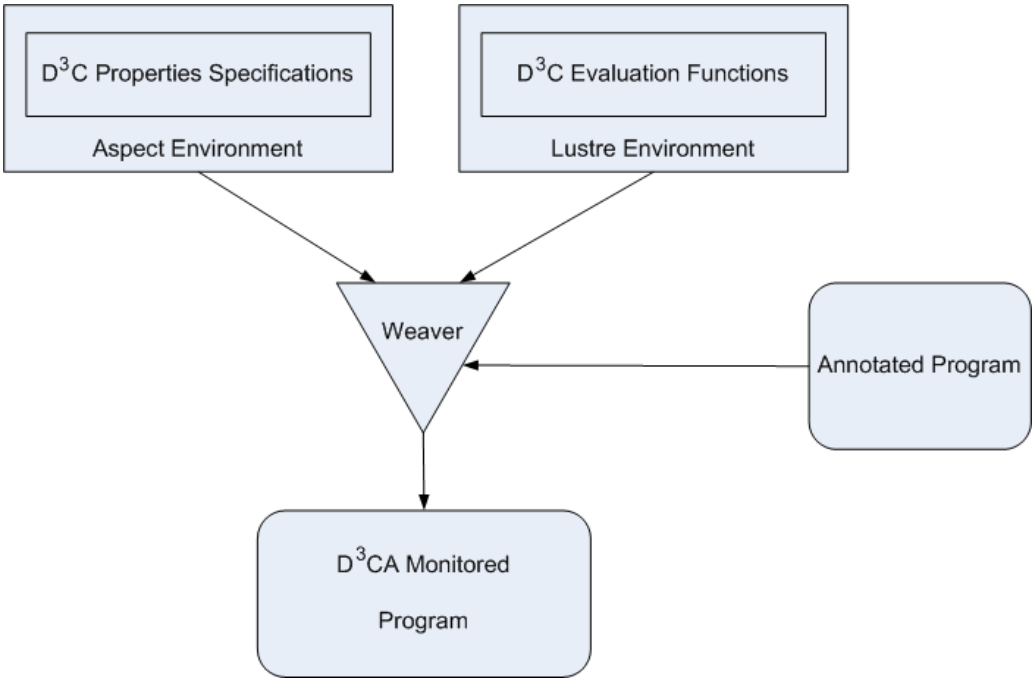


Figure 8.1: Aspect-Oriented Programming Architecture Overview.

The obliviousness property states that an aspect must not affect the system composition. For example, a program is extended with a log file. The logging of method calls or of events can be performed using the same logging code. Logging crosscuts the system so it is an aspect. The logger is oblivious because the logging insertion has no effect on the system code.

A number of AOP implementations for different programming languages have been developed. Some of the frameworks are: Aspect .NET [SSW02], AspectJ [Asp], Hyper/J [IBM], and AspectC++ [SLU05]. Given that validation is an aspect then one can introduce formal checking within any programming language using AOP tools.

8.1 Validation as an Aspect

The definition of an aspect is vague. Therefore we restrict the definition of an aspect as **a task whose *execution semantics* crosscut the program's decomposition and are oblivious to the developer**. In other words, an aspect is any task whose execution depends on different components and does not localise to any part of the system.

Interval temporal logics (ITL) require a time metric. This chapter uses the `synchronise` call to the validation engine as the basic clock. The `synchronise` call provides a global mechanism to enforce the properties to be checked over the new state of the system.

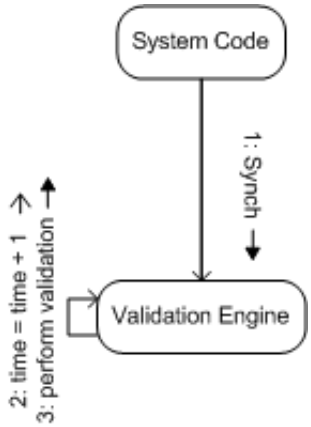


Figure 8.2: `Synchronise` communication diagram

The fundamental property for aspects is their crosscutting behaviour. The crosscutting behaviour of ITL properties arises from their requirement to capture information from different parts of the system.

The insertion of the `synchronise` call throughout the system code and the distributivity of ITL properties satisfy the crosscutting requirement of aspects. Figure 8.3 illustrates the crosscutting property over the execution states.

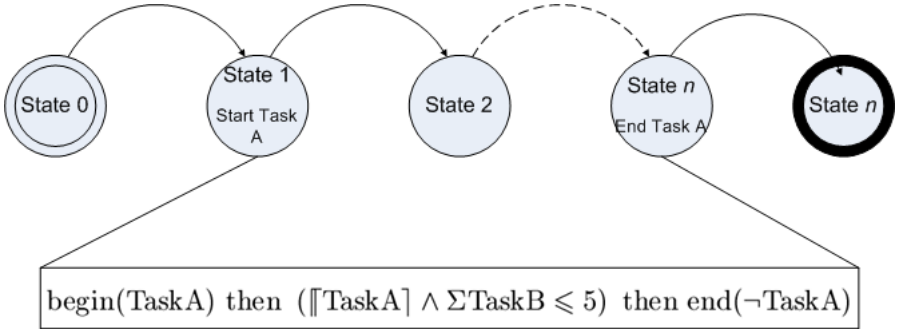


Figure 8.3: States representing an execution path.

Consider the property,

$$\text{begin(TaskA) then } (\lceil\text{TaskA}\rceil \wedge \Sigma\text{TaskB} \leqslant 5) \text{ then end}(\neg\text{TaskA})$$

The property states that from the start of TaskA, TaskB cannot be executed more than five times both directly and indirectly until TaskA finishes executing. A direct call to TaskB occurs when TaskA requires the service of TaskB whereas an indirect call happens when TaskB is called from any other task and not TaskA.

The last requirement for validation to be an aspect is the obliviousness property. Obliviousness is achieved by abstracting as much as possible the dependencies between the code and the formal specifications. In D^3CA described earlier obliviousness is achieved by using

annotations to control the weaving of ITL properties with the system code.

8.2 Defining D^{3}CA weaver in AOP terms

The D^{3}CA weaver module consists in transforming annotated control instructions to the actual validation code. The annotated control instructions can be of five types:

1. {*validation_engine*: **bind** *variable_name variable_value*}
2. {*validation_engine*: **unbind** *variable_name*}
3. {*validation_engine*: **start** *assertion_name*}
4. {*validation_engine*: **stop** *assertion_name*}
5. {*validation_engine*: **synchronise** $\mathbb{B}$}

In validation the state variables are mapped to the system variables. In software, variables are typically placed in their local scope and are inaccessible from the outside code. This gives rise to a problem with Interval Temporal Logic assertions since they have crosscutting semantics. This problem is surmounted by providing two variable annotations - `bind` and `unbind`. A variable can either be bound to a system variable while the variable is in scope or else bound to a numeric value. When the variable is bound to a numeric value it is treated as a constant variable.

The `unbind` annotation is used when a system variable scope is about to be lost. The `unbind` annotation instructs the weaver that the value of the variable must be kept constant.

An important characteristic of the D^{3}CA is the use of intervals. An interval have to be defined for every assertion. This is done by inserting one of the two interval assertions – `start` and `stop`. The `start` annotation takes an assertion name as a parameter. When the weaver encounters the notation it sets the assertion interval to zero and starts the observation interval.

Assertions might not require to be checked for the entire program execution. The `stop` annotation is used to indicate the end of an assertion's interval. As with the `start` annotation, the `stop` annotation requires an assertion name to be passed as a parameter. When the `stop` annotation is encountered, the weaver instructs the assertion that its interval has elapsed. An important feature of the `stop` annotation is that it calls the validation engine to check the system state. When the stopped assertion is checked, its return value is expected to be **true**. Therefore, if the return value is `indeterminate`, the validation engine treats it as **false** because it has not been completely satisfied during the interval.

The last and most important annotation is `synchronise`. This annotation instructs the weaver that the properties have to be checked for consistency. Before performing the runtime checking the variables are updated. The update also includes the reassignment of constant variables to reflect their values according to the current state, as shown in Figure 8.4.

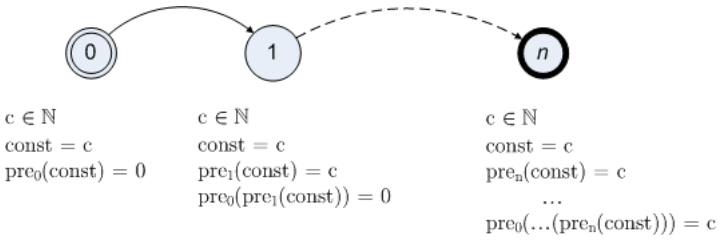


Figure 8.4: Lustre constant to system state relation

The `synchronise` annotation takes a Boolean value. This value indicates whether pn the trapping an error the synchronisation method should abandon execution or not. However, the runtime checker still reports the errors that have been encountered, regardless of the value passed.

From the description of the D^{3}CA weaver, one can notice that it is only a small instantiation of the AOP concept.

8.3 Conclusion

The Aspect-Oriented Programming (AOP) technique has emerged as a suitable programming style for separating different concerns – *aspects*. AOP defines an aspect as any task whose execution semantics depends on data not local to a single task. Using the work in the previous chapters, this chapter argues that validation is an aspect. Therefore, AOP tools can be used to cleanly weave validation into the system code.

Chapter 9

Conclusions

> "One way that you can fortify your software's exception–handling ability is to harness formal specification statements. Intended for run-time verification of an application's design, formal specifications can be translated by code generator into C, C++, or Java statements to be deployed for catching exceptions in the final product"
>
> Doron Drusinsky [Dru01]

Formal verification has always been an important tool for guaranteeing correctness of software. The problem with the method of formal verification is that it requires the monitoring system to construct an automaton to model all possible execution paths in the software. In large and/or complex systems the number of system states grows exponential. This may lead the system to become unresponsive due to resource exhaustion by test runs.

Throughout the years, validation has been proposed as a mechanism to surmount *the state explosion problem*. Validation proposes a monitoring solution for performing formal verification during a program execution. This problem is overcome by performing validation *on a single* execution path. One must note that the state explosion problem is still present in validation; however, it is reduced because states along paths never visited don't need to be created.

Projects in validation have concentrated on the use of different logic notation, mainly propositional point logic and temporal logic. Nevertheless, some reactive and real-time properties might be best captured or less messy if represented in an interval temporal logic.

9.1 Research Overview

Part I provides the necessary background to the dissertation and a review of related projects. The main research carried out throughout the dissertation is put together in Part II.

Section 4.1 identifies a suitable subset of Duration Calculus for performing runtime monitoring. The subset consists of operators that are lifted to discrete time and whose semantics produce deterministic automata. The operators are restricted to discrete time because the operators in the original Duration Calculus, presented by Chaochen [CHR91], work on real-time intervals, which makes it impossible to measure during execution time. The Duration Calculus operators are all based on the Reimann Integral, which can easily be calculated using frequent sampling points.

Some of the Duration Calculus operators, such as the chop operator, have a non-deterministic behaviour. Checking the validity of a state during execution can only be performed on the current and previous states. Thus non-deterministic behaviour rising from left-branching has to be avoided. Following the work of Gonnord *et. al.* [GHR04] the subset obtained by the discretisation of the operators is further restricted to operators that produce a deterministic automaton.

After having identified the set of operators that are useful for monitoring systems, the operators execution semantics in terms of the synchronous data-flow programming language Lustre [HCRP91] have been obtained. The programming language Lustre is used as it allows the computation of a new state to be performed in no time, under the synchrony hypothesis. Lustre has also been adopted as it allows memory and the time requirements for computing the validity of a state to be predetermined.

9.1.1 Monitoring System

One of the objectives of thus dissertation is to provide a generic implementable framework to monitor properties using the identified subset of Duration Calculus. The framework proposed consists of three modules:

1. The Duration Calculus mathematical notation with its semantics in Lustre;
2. A weaver for integrating the mathematical notation inside the system code; and
3. The host language in which the system code is written.

The monitoring engine is executed over a simulated Lustre environment provided as a library to the host language, which is linked statically during compilation. The library consists of three datatypes representing the Lustre basic datatypes: integer, real and Boolean. During

the evaluation of the logic the ambiguous meaning of the Boolean value **false** leads to complex implementation, so the Lustre Boolean operator is replaced with the 3-valued logic operator.

A key feature of the framework provides the monitoring system with a weaver. The weaver allows annotations to be used inside the system code to control and communicate with the validation engine. The end result of weaving the annotations is a duplicate of the original code with the annotations converted into the respective tangled code.

Executing the mathematical semantics over Lustre allows the framework to be implementation-independent. This endows the system with the ability to be enhanced without the need to modify the validation engine. The use of annotations together with the automated process of converting properties to a monitoring library results in a flexible and easy to use environment.

9.2 Possible future enhancements

The solution provided in this dissertation is far from optimal and lacks in providing useful features. This section succinctly highlights some of the possible enhancements or modifications that can be undertaken.

9.2.1 Multiple Validation Interfaces

The solution implemented allows the scope for using more than one validation engine. This is achieved by having the validation engine provide its own symbol table for storing variables and a list to hold the assertions.

The system lacks a global clock over the validation instances since each engine keeps its own clock. Another drawback of the current approach is that it becomes unmanageable when the number of engines increases.

In graphical user interfaces a similar problem exists when the number of open windows related to the same application is large, therefore making it impossible for the user to keep track of his work. A solution that is adopted in the field of computer-human interface is the Multiple Document Interface (MDI), whereby multiple windows appear as part of the main parent window. Following the concepts of MDI and plug-ins, this section describes a solution for having one single validation engine but multiple instances of validation classes.

As shown in Figure 9.1, the validation has to be defined using two separate modules. The first module provides the validation engine, a list of validation instances and the global

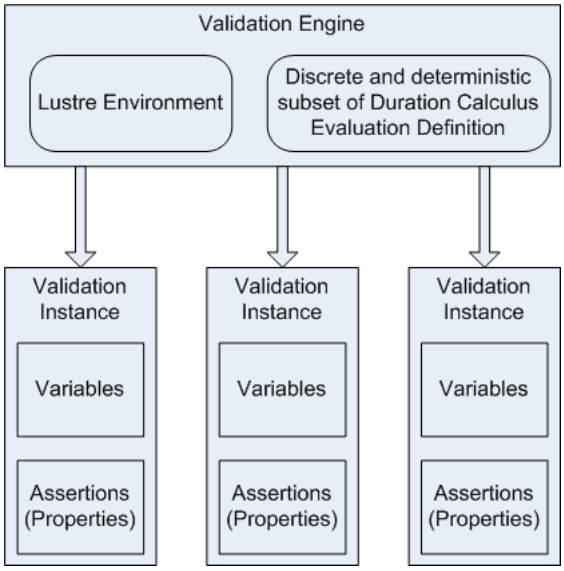


Figure 9.1: Multiple Validation Interfaces Architecture.

clock. The engine extends the annotations set with two new tuples to add and remove validation instances.

The validation instance provides the necessary information to the engine, such as the assertions to be checked and the variables values related to the assertions. The validation instance can also be extended with a clock which allows it to run asynchronously from other validation instances. The use of different validation instances makes it easier to handle multithreaded systems as each class can be instructed to add its own validation to the engine, without leading to ambiguity when using similar names for variables.

The validation engine implements the synchronise mechanism as described in previous chapters, with a difference in how assertions and variables are accessed. However the `synchronise` mechanism can be extended to allow validations to run asynchronously to each other, similar to a locally asynchronous global synchronous (LAGS) concept, which allows validation instances to be checked without checking other instances.

9.2.2 Logic Domains

Duration Calculus is one of the logics currently being used. Different logics provide notation for different properties in systems [BMN00]. By defining the logic semantics in the Lustre environment and extending the annotation tuples with an identifier for the logic, as done in Eagle, the same mechanism for monitoring the systems can be used.

9.3 Summary

This dissertation proposes a solution that allows formal specifications using deterministic and discrete Duration Calculus to be integrated in a system. The D^3CA described is generic to any logic notation as long as the notation can be translated into Lustre programs.

The case studies presented are real-life scenarios and, with minimal changes for hardware adaptation, can be installed on the real equipment. The case studies showed that the D^3CA is capable of detecting most of the errors instantaneously.

As an addition to the research, we have showed that Interval Temporal Logic validation is an aspect in Aspect-Oriented Programming (AOP). Thus the tools already available for AOP can be used to integrate the core part of the D^3CA in any programming language.

Appendix A

"leads to" operator in terms of age()

The "leads to" operator cannot be directly described in terms of the deterministic QDDC notation as it is non-deterministic. However, analytically the operator can be represented using deterministic QDDC notation. It is important to note that the reducibility of the QDDC operator to the deterministic QDDC expression is not proved anywhere in this report as it was not formally proven.

Relation. The "leads to" ($\xrightarrow{\delta}$) can be expressed as an expression in terms of `age(`P`)` and "then" operators.[1]

$$P \xrightarrow{\delta} Q \triangleright \texttt{age(}P\texttt{)} < \delta \text{ then } P \wedge Q$$

Analytical Proof. The "leads to" part states that if the state variable P has been **true** for δ clock cycles and is still **true** then while P is still **true**, Q must also be **true**.

The second part of the relation states that the interval is divided in two subintervals. The first subinterval ends when the state variable, P, has been true for δ clock cycles. Then from the δ clock cycle onward, the state variable, Q, must be **true** , when P is **true**. The result of the `age()..then` expression is similar to that of the "leads to" expression without the "always" ($\Box$) operator. $\Box$

[1] The symbol $\triangleright$ means that the RHS expression is reducible to the LHS expression.

Appendix B

Case Studies Properties in XML format

B.1 Mine Pump

```
<?xml version="1.0" encoding="UTF-8" ?>

<qddc>
```

Pump Properties

```
  <variable name="HH2O" type="bool" inputVar="true" />
  <variable name="HCH4" type="bool" inputVar="true" />
  <variable name="Delta" type="int" inputVar="true" />
  <variable name="PumpOn" type="bool" inputVar="false" />
  <variable name="SafePump" type="bool" inputVar="false" />
  <variable name="NotSafePump" type="bool" inputVar="false" />
  <variable name="SafePumpAge" type="bool" inputVar="false" />
  <variable name="NotSafePumpAge" type="bool" inputVar="false" />

  <property name="SafePump">
    <expression> SafePump = HH2O &amp;amp; !HCH4 </expression>
  </property>

  <property name="NotSafePump">
    <expression> NotSafePump = !SafePump </expression>
  </property>
```

```
<property name="SafePumpAge">
  <expression> SafePumpAge = age(SafePump) == Delta </expression>
</property>

<property name="NotSafePumpAge">
  <expression> NotSafePumpAge = age(!SafePump) == Delta </expression>
</property>

<property name="PumpOn">
  <expression>
    PumpOn = two_state(false, SafePumpAge, NotSafePumpAge)
  </expression>
</property>
```

Alarm Properties

```
<variable name="Alarm" type="bool" inputVar="false" />
<variable name="DH2O" type="bool" inputVar="true" />
<variable name="ah2o" type="bool" inputVar="false" />
<variable name="ach4" type="bool" inputVar="false" />
<variable name="na" type="bool" inputVar="false" />
<variable name="na_int" type="bool" inputVar="false" />
<variable name="alarm_int" type="bool" inputVar="false" />

<property name="AH2O">
  <expression> ah2o = age(DH2O) == Delta </expression>
</property>

<property name="ACH4">
  <expression> ach4 = age(HCH4) == Delta </expression>
</property>

<property name="Not_DH2O_and_Not_HCH4">
  <expression> na_int = !DH2O &amp;amp; !HCH4</expression>
</property>

<property name="NA">
  <expression> na = age(na_int) == Delta</expression>
</property>
```

```
<property name="H2O_or_HC4">
  <expression> alarm_int = ah2o | ach4</expression>
</property>

<property name="Alarm">
  <expression> Alarm = two_state(false, alarm_int, na) </expression>
</property>
```

Assertion Properties

```
<variable name="Assertion" type="bool" inputVar="false" />
<variable name="w" type="int" inputVar="true" />
<variable name="eps" type="int" inputVar="true" />
<variable name="zeta" type="int" inputVar="true" />
<variable name="kappa" type="int" inputVar="true" />
<variable name="ws1" type="bool" inputVar="false" />
<variable name="ws2_int" type="bool" inputVar="false" />
<variable name="ws" type="bool" inputVar="false" />
<variable name="pc" type="bool" inputVar="false" />
<variable name="LCH4" type="bool" inputVar="false" />
<variable name="mr1_int" type="bool" inputVar="false" />
<variable name="mr1" type="bool" inputVar="false" />
<variable name="mr2" type="bool" inputVar="false" />

<property name="Water_Seepage_1">
  <expression> ws1 = DH2O => HH2O </expression>
</property>

<property name="Water_Seepage_2_int">
  <expression> ws2_int = age(HH2O) &lt; w </expression>
</property>

<property name="Water_Seepage_2">
  <expression> ws2 = edge(DH2O) =&gt; !ws2_int</expression>
</property>

<property name="Pump_Capacity">
  <expression> pc = HH2O =&gt; age(PumpOn) &lt;= eps </expression>
</property>
```

```
  <property name="Low_Methane">
    <expression> LCH4 = !HCH4 </expression>
  </property>

  <property name="Methane_Release_1_int">
    <expression> mr1_int = age(LCH4) &lt; zeta </expression>
  </property>

  <property name="Methane_Release_1">
    <expression> mr1 = [[HCH4]] then [[HCH4]] =&gt; !mr1_int </expression>
  </property>

  <property name="Methane_Release_2">
    <expression> mr2 = age(HCH4) &lt; kappa </expression>
  </property>

  <property name="Assertion">
    <expression>
      Assertion = ws1 &amp; ws2 &amp; pc &amp; mr1 &amp; mr2
    </expression>
  </property>
```

Validation Property

```
  <property name="Valid">
    <expression> end(Assertion) | !end(DH2O) </expression>
  </property>

</qddc>
```

B.2 Answering Machine

```
<?xml version="1.0" encoding="UTF-8" ?>
<qddc>
  <variable name="Idle" type="bool" inputVar="false" />
  <variable name="Ringing" type="bool" inputVar="true" />
  <variable name="Playing" type="bool" inputVar="true" />
  <variable name="Recording" type="bool" inputVar="true" />
  <variable name="ReceiverUp" type="bool" inputVar="true" />
```

```
<variable name="Operating" type="bool" inputVar="false" />
<variable name="Max_Age_Ringing" type="bool" inputVar="false" />

<property name="Idle">
  <expression>
    Idle = !Ringing &amp;amp; !Playing &amp;amp; !Recording
  </expression>
  <error>
    It is not possible to have the system idle while it is
    busy doing something.
  </error>
  <comment>Attempting to set the system idleness.</comment>
</property>

<property name="Max_Age_Ringing">
  <expression>Max_Age_Ringing = age(Ringing) == 9 </expression>
</property>

<property name="Assumption_1">
  <expression>Idle &amp;lt;~ Ringing</expression>
  <error>The system is ringing when it was already busy.</error>
  <comment>
    The system can only receive calls when it is idle.
  </comment>
</property>

<property name="Assumption_2">
  <expression>Max_Age_Ringing &amp;lt;~ Playing</expression>
  <error>
    The system is playing the recorded message when no one
    called.
  </error>
  <comment>
    The system message can only play if the callee has not
    answered.
  </comment>
</property>

<property name="Assumption_3">
  <expression>Playing &amp;lt;~ Recording</expression>
  <error>Ooops the system is wasting space on tape.</error>
  <comment>The caller doesn't know that the system is recording
```

```
      his message.
    </comment>
  </property>

  <property name="Assumption_4">
    <expression>end(ReceiverUp) =&gt; end(Idle)</expression>
    <error>
      The answering machine and the owner are using the phone
      line simultaneously.
    </error>
    <comment>
      If the caller is talking on the phone the answering machine
      must be switched off.
    </comment>
  </property>

  <property name="Operating">
    <expression>
      Operating = ([[Idle]] then age(Ringing) &lt; 10 then [[Playing]]
                  then age(Recording) &lt; 30)+
    </expression>
    <error>The answering machine is not acting as expected.</error>
    <comment>
      Stop playing around with the answering machine settings.
    </comment>
  </property>

  <property name="Valid">
    <expression>end(Operating) | end(ReceiverUp)</expression>
  </property>
</qddc>
```

Bibliography

[ABG+04] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Roşu, K. Sen, W. Visser, and R. Washington. Combining Test Case Generation with Run-time Verification. *ASM issue of Theoretical Computer Science*, 2004. To appear.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[Ale03] Andrei Alexandrescu. Generic<programming>: Assertions. *C/C++ Users Journal (Advanced Solutions for Professional Developers)*, April 2003.

[Alu99] Rajeev Alur. Timed automata. In *Computer Aided Verification*, pages 8–22, 1999.

[AS01] Karine Arnout and Raphal Simon. The .net contract wizard: Adding design by contract to languages other than eiffel. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, page 14. IEEE Computer Society, 2001.

[Asp] AspectJ. `http://www.eclipse.org/aspectj/`.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, Reading, MA, USA, January 1985.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicholas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. In *Proceedings of the IEEE*, volume 91, pages 64–83, January 2003.

[BFMW01] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - java with assertions. In *Workshop on Runtime Verification, 2001 in conjunction with the 13th Conference on Computer Aided Verification, CAV'01*, 2001.

[BG92] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGHS03] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Eagle monitors by collecting facts and generating obligations. Technical Report Pre-Print CSPP-26, University of Manchester, Department of Computer Science, University of Manchester, October 2003.

[BGHS04a] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification, 2004.

[BGHS04b] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Program monitoring with LTL in Eagle. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 16*, April 2004.

[BJHL96] M. Brockmeyer, F. Jahanian, C. Heitmeyer, and B. Labaw. An approach to monitoring and assertionchecking distributed real-time systems, 1996.

[BM02] Mark Brörkens and Michael Möller. jassda Trace Assertions, runtime checking the dynamic of java programs. In *Trends in Testing Communicating Systems, International Conference on Testing of Communicating Systems*, pages 39–48, Berlin, March 2002.

[BMN00] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.

[BRLS04] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An Overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362, pages 49–69, Marseille, France, March 2004. Springer-Verlag GmbH.

[CCO02] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1):140–165, 2002.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[CE04] T. Cottenier and T. Elrad. Validation of context-dependent aspect-oriented adaptations to components. In *Ninth International Workshop on Component-Oriented Programming (ECOOP 2004)*, WS18 at ECOOP 2004, Oslo, Norway, June 2004.

[CES97] W. Canfield, E. Emerson, and A. Saha. Checking formal specifications under simulation. In *Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, page 455. IEEE Computer Society, 1997.

[CGN99] Z. Chaochen, D. Guelev, and Z. Naijun. A higher-order duration calculus, 1999.

[CGP02] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 4th edition, 2002.

[CHR91] Z. Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.

[CHX95] Zhou Chaochen, Dang Van Hung, and Li Xiaoshan. A duration calculus with infinite intervals. In *Fundamentals of Computation Theory*, pages 16–41, 1995.

[CP02] Gaurav Chakravorty and Paritosh K. Pandya. Digitizing interval duration calculus. Technical report, Tata Institute of Fundamental Research, 2002.

[CR03] F. Chen and G. Roşu. Towards Monitoring–Oriented Programming: A paradigm combined specification and implementation. In *Proceedings of RV'03: the Third International Workshop on Runtime Verification.*, volume 89, Boulder, Colorado, USA., 2003. Electronic Notes in Theoretical Computer Science, Elsevier Science.

[D'E06] Karlston D'Emanuele. Runtime monitoring of duration calculus assertions for real-time applications. Master's thesis, Computer Science and A.I. Department, University of Malta, 2006. To be submitted.

[DF04] Doron Drusinsky and J.L. Fobes. Executable specifications: Language and applications. *CrossTalk - The Journal of Defense Software Engineering*, 17(9):15–18, September 2004.

[DH03] Doron Drusinsky and Klaus Havelund. Execution-based model checking of interrupt-based systems. In *Workshop on Model Checking for Dependable Software-Intensive Systems. Affiliated with DSN'03, The International Conference on Dependable Systems and Networks*, pages 22–25, 2003.

[Dru00] Doron Drusinsky. The temporal rover and the ATG rover. In *SPIN*, pages 323–330, 2000.

[Dru01] Doron Drusinsky. Formal specs can handle exceptions. *Embedded Developers Journal*, pages 10–15, November 2001.

[DT02] Deepak D'Souza and P. S. Thiagarajan. Product interval automata, 2002.

[Eme90] E. A. Emerson. The role of büchi's automata in computing science. In S. MacLane and D. Siefkes, editors, *The Collected Works of J. R. Büchi*. Springer, Berlin, 1990.

[FF05] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[Gei01] M. Geilen. On the construction of monitors for temporal logic properties. In Klaus Havelund and Grigore Roşu, editors, *RV'01 - First Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 78–96, Information and Communication Systems Group, Faculty of Electric Enginnering, Eindhoven University of Technology, P.O.Box 513, 5600 MB Eindhoven, The Netherlands, July 2001. Elsevier Science Publishers.

[GH01] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. Technical Report 01.21, Research Institute for Advanced Computer Science, August 2001. Presented at the 16th IEEE International Conference on Automated Software Engineering, San Diego, 2001.

[GHM05] Allen Goldberg, Klaus Havelund, and Conor McGann. Runtime verification for autonomous spacecraft software. In *IEEE Aerospace Conference*, 2005.

[GHR04] L. Gonnord, N. Halbwachs, and P. Raymond. From discrete duration calculus to symbolic automata. In *3rd International Workshop on Synchronous Languages, Applications, and Programs, SLAP'04*, Barcelona, Spain, March 2004.

[GM03] Tanton H. Gibbs and Brian A. Malloy. Weaving aspects into c++ applications for validation of temporal invariants. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 249, Washington, DC, USA, 2003. IEEE Computer Society.

[GME04] Philip J. Guo, Stephen McCamant, and Michael D. Ernst. Safe runtime examination of data structures in C programs, September 2004.

[Gue00] Pedro Guerreiro. Another mediocre assertion mechanism for c++. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 226. IEEE Computer Society, 2000.

[Gui01] Kristian Guillaumier. CSM201 Compiling Techniques – Course notes, January 2001.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16, 1998.

[Han03] Michael R. Hansen. Duration calculus (extended abstract). In *15th European Summer School in Logic Language and Information*, pages 66–84. Kurt Gödel Society, Norbert Preining, August 2003.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HG96] D. V. Hung and P. H. Giang. Sampling semantics of duration calculus. *Lecture Notes in Computer Science*, 1135:188–??, 1996.

[HG99] Dang Van Hung and Dimitar P. Guelev. Completeness and decidability of a fragment of duration calculus with iteration. In *Asian Computing Science Conference*, pages 139–150, 1999.

[HHR94] C. A. R. Hoare, He Jifeng, and A. P. Ravn. Specification and implementation of a flashlight. ProCoS II document [OU CARH 2/2], Oxford University, UK, June 1994.

[Hil00] M. Hiller. Executable assertions for detecting data errors in embedded control systems, 2000.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[Hoa01] Tony Hoare. Assertions: a personal perspective. `http://research.microsoft.com`, June 2001. Draft.

[Hou02] Bernard Houssais. The synchronous programming language SIGNAL: A tutorial. SIGNAL online documentation– `http://www.irisa.fr/espresso/Polychrony/`, April 2002.

[HR04] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool java pathexplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[HS99] Klaus Havelund and Jens U. Skakkebæk. Applying model checking in java verification. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 216–231. Springer-Verlag, 1999.

[IBM] IBM. Hyper/J ™: Multi-dimensional separation of concerns for Java ™. `http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm`.

[Jos01] Mathai Joseph, editor. *Real-time systems specification, verification and analysis*. Tata Research Development and Design Centre, Jume 2001.

[Kel95] Den Kelley. *Automata and Formal Languages*. Prentice Hall, 1995.

[KKL+01] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for Java prgrams. In *1st Workshop on Runtime Verification (RV'01), volume 55 of ENTCS*, 2001.

[KKLS02] József Kovács, Gábor Kusper, Róbert Lovas, and Wolgang Schreiner. Integrating temporal assertions into a parallel debugger. In *Proceedings of the 8th International Euro-Par Conference*, pages 113–120, 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Kra98] R. Kramer. iContract - the Java(tm) design by contract(tm) tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.

[KSL02] Gábor Kusper, Wolgang Schreiner, and Róbert Lovas. Integrating temporal assertions into parallel debugging tools. Project report, RISC-Linz, March 2002.

[KV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[LA03] Ramnivas Laddad and Roger T. Alexander. Aspect-oriented programming will improve quality / aspect-oriented programming: the real costs? *IEEE Software*, 20(6):90–93, 2003.

[LG91] P. Le Guernic and T. Gautier. Data-flow to von neumann: the signal approach. In J.L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 413–438. 1991.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall Resources Series. Prentice-Hall, 2nd edition, Febrauary 1997.

[Mid98] C. A. Middelburg. Truth of duration calculus formulae in timed frames. In *1517*, page 22. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 31 1998.

[Mos84] Ben Moszkowski. Executing temporal logic programs. Technical Report UCAM-CL-TR-55, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, August 1984.

[MV04] Brian A. Malloy and Jeffrey M. Voas. Programming with assertions: A prospectus. *IT Professional*, 6(5):53–59, 2004.

[P+05] Terence J. Parr et al. ANTLR reference manual. `http://www.antlr.org`, January 2005. ANTLR Version 2.7.5.

[Pac94] Gordon J. Pace. Duration calculus: From parallel specifications to clocked circuits. M.Sc. Dissertation, Computing Laboratory, University of Oxford, 1994.

[Pac98] Gordon J. Pace. *Hardware Design Based on Verilog HDL*. PhD thesis, Computing Laboratory, University of Oxford, 1998. Chapter 3 §3.2, Pages 21–27.

[Pan] Paritosh Pandya. Model checking CTL[DC] properties of SMV, verilog and esterel designs.

[Pan00] P. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID. Technical Report TCS00-PKP-1, Tata Institute of Fundamental Research, 2000.

[Pan01] Paritosh K. Pandya. Model checking ctl*[dc]. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 559–573. Springer-Verlag, 2001.

[Pan02a] P. Pandya. The saga of synchronous bus arbiter: On model checking quantitative timing properties of synchronous programs. In Florence Maraninchi, Alain Girault, and ric Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[Pan02b] Parithosh Pandya. Interval duration logic: Expressiveness and Decidability. In *Proc. workshop on Theory and Practice of Timed Systems (TPTS'2002)*, Grenoble, France, April 2002. Electronic Notes in Theoretical Computer Science, Elsevier Science B.V., ENTCS 65.6.

[Pen99] Roger Penrose. *The Emperor's New Mind*, pages 86–92. Oxford University Press, March 1999.

[Pet99] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Department of Computer Systems, Uppsala University, February 1999. Chapter 1 §1.3, Pages 8–10.

[PH98] Paritosh K. Pandya and Dang Van Hung. Duration calculus of weakly monotonic time. *Lecture Notes in Computer Science*, 1486:55–??, 1998.

[PP99] Reinhold Plösch and Josef Pichler. Contracts: From analysis to C++ implementation. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 248, Washington, DC, USA, 1999. IEEE Computer Society.

[Rav94] Anders P. Ravn. *Design of embedded real-time computing systems.* PhD thesis, Department of Computer Science, Technical University of Denmark, September 1994.

[Ray96] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)*, Paderborn, Germany, July 1996. LNCS 1099, Springer Verlag.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[RS] Rafael Ramirez and Andrew E. Santosa. An aspect-oriented framework for concurrent applications.

[SH04] Volker Stolz and Frank Huch. Runtime verification of concurrent haskell programs. *Electr. Notes Theor. Comput. Sci.*, 113:201–216, 2004.

[Sim99] Charles Simonyi. Hungarian notation. `http://msdn.microsoft.com`, November 1999.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[SLS05] Usa Sammapun, Insup Lee, and Oleg Sokolsky. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA'05*, Hong Kong, August 2005.

[SLU05] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: an AOP extension for C++, May 2005.

[SS94] Jens Ulrik Skakkebæk and Natarajan Shankar. Towards a duration calculus proof assistant in PVS. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 660–679, Lübeck, Germany, sep 1994. Springer-Verlag.

[SSW02] Mario Schüpany, Christa Schwanninger, and Egon Wuchner. Aspect-oriented programming for .NET. In *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, March 2002.

[Str00] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley, Special edition, 2000.

[Tau03] Heikki Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2003.

[Tha05] Sahil Thaker. Runtime monitoring temporal property specification through code assertions. Department of Computer Science, University of Texas at Austin, 2005.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. pages 389–455, 1997.

[TR04] P. Thati and G. Roşu. Monitoring algorithms for metric temporal logic specifications. In *Proceeding of the 4th International Workshop on Runtime Verification (RV 2004)*, 2004.

[Var97] Moshe Y. Vardi. Alternating automata: Unifying truth and validity checking for temporal logics. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249, pages 191–206, Berlin, 13–17 1997. Springer.

[Voa97] Jeffrey Voas. How assertions can increase test effectiveness. *IEEE Software*, 14(2):118–119,122, 1997.

[WD96] Jim Woodcock and Jim Davis. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., 1996.

[Wei05] Eric W. Weisstein. "Zeno's Paradoxes.". From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/ZenosParadoxes.html, 2005.

[Wol02] Pierre Wolper. Constructing automata from temporal logic formulas: a tutorial. pages 261–277, 2002.

[Ziv03] Avi Ziv. Cross-product functional coverage measurement with temporal properties-based assertions. In *Proceedings of the Design Automation and Test in Europe*, pages 834–839. IEEE Computer Society Press, 2003.