ITL Monitor: Compositional Runtime Analysis with Interval Temporal Logic

David Smallwood

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy, awarded by De Montfort University.

May 2019

To Mary Hunt

Abstract

Runtime verification has gained significant interest in recent years. It is a process in which the execution trace of a program is analysed while it is running. A popular language for specifying temporal requirements for runtime verification is Linear Temporal Logic (LTL), which is excellent for expressing properties such as safety and liveness.

Another formalism that is used is Interval Temporal Logic (ITL). This logic has constructs for specifying the behaviour of programs that can be decomposed into subintervals of activity [Mos83]. Traditionally, only a restricted subset of ITL has been used for runtime verification due to the limitations imposed by making the subset executable. In this thesis an alternative restriction of ITL was considered as the basis for constructing a library of runtime verification monitors (ITL-Monitor).

The thesis introduces a new first-occurrence operator ($\triangleright$) into ITL and explores its properties. This operator is the basis of the translation from runtime monitors to their corresponding ITL formulae. ITL-Monitor is then introduced formally, and the algebraic properties of its operators are analysed. An implementation of ITL-Monitor is given, based upon the construction of a Domain Specific Language using Scala. The architecture of the underlying system comprises a network of concurrent actors built on top of Akka - an industrial-strength distributed actor framework.

A number of example systems are constructed to evaluate ITL-Monitor's performance against alternative verification tools. ITL-Monitor is also subjected to a simulation that generates a very large quantity of state data. The monitors were observed to deliver consistent performance across execution traces of up to a million states, and to verify subintervals of up to 300 states against ITL formulae with evaluation complexity of $\mathcal{O}(n^3)$.

Declaration

I declare that the work in this thesis is original work undertaken by me between October 2010 and May 2019 for the degree of Doctor of Philosophy at the Software Technology Research Laboratory (STRL), De Montfort University, United Kingdom.

Acknowledgements

I would like to thank my original and long-standing supervisor Antonio Cau with whom I have had many discussions throughout these past years on runtime verification, Interval Temporal Logic, CCS, Tempura, Isabelle/HOL, and many other topics; and whose advice and guidance has been most gratefully appreciated. I would also like to thank Antonio for checking all of the mathematical proofs, and subsequently for encoding ITL into Isabelle/HOL thus enabling the mechanical proof checking of all of the laws developed within this thesis. I am also grateful to Antonio for reading and commenting upon several drafts of the thesis. I would like to thank my first supervisor, Helge Janicke, and members of the Faculty of Computing, Engineering and Media who have motivated and inspired me over the years including Hussein Zedan, Francois Siewe, Susan Bramer, Pam Watt, and Peter Messer. I would also like to record my gratitude to the University for supporting me with this postgraduate work. Finally, thank you to my friends and family who have accompanied me along the journey.

Mathematical laws and the software library

Mathematical laws

Throughout this thesis reference is made to laws in ITL. These are referenced by name and number such as $\mathit{FstFixFst}^{(C.261)}$. The number refers to the law's position in Appendix C. The name of each law follows the style used by Moszkowsi in (`http://antonio-cau.co.uk/ITL/itl-theorems/itl-theorems-home.pdf`).

All of the ITL laws have been proved mechanically using Isabelle/HOL [oCM18]. The Isabelle encoding for ITL was constructed by Antonio Cau as was the translation of Moszkowski's and Cau's earlier work into Isabelle. Cau also translated the LaTeX proofs that were constructed by the author of this thesis as part of the current work.

The complete document is available from the ITL homepage [CM16]. The work undertaken as part of this thesis is contained in Chapters 6 and 7. To access the document[1]

- Navigate to `http://antonio-cau.co.uk/ITL/index.html`
- Under Section 3 'Tools', select the link to the ITL library for Isabelle/HOL `http://antonio-cau.co.uk/ITL/itlhomepagesu13.html#x17-220003.3`
- Within the 'Deep embedding' section is a download (Version 1.9 (16/03/2019)) which is a zipped unix archive file

Software library

The Scala libraries developed as part of this thesis are distributed using an `sbt` archive. The libarary is currently available from the author, and will appear in the tools section of the ITL homepage [CM16].

[1] URL correct as of May 2019

Glossary of key terms

Acronyms

API Application Program Interface.

DSL Domain Specific Language 2.4.1.

ITL Interval Temporal Logic [CM16].

JVM Java Virtual Machine.

LTL Linear Temporal Logic [Pnu77].

RV Runtime Verification.

Terms

Actor A process, typically running in its own thread, that reacts to received messages by (possibly) updating internal state and sending messages to other actors.

Akka A framework and API for managing actors in JVM-based systems. [Akk17]

Compositionality The ability to reason about a system by combining analyses of its constituent parts.

Chop (;) The name of the sequential composition operator in ITL. It is used to split an interval non-deterministically into two subintervals, each of which satisfies its respective formula, connected by a shared state, e.g. $f \; ; \; g$.

Chopstar (*) The name of the repetition operator in ITL. It is used to specify a non-deterministic number of sequentially composed intervals, each satisfying the given formula, e.g. f^*.

First occurrence ($\triangleright$) The key ITL operator defined within this thesis. The purpose of $\triangleright f$ is to define an interval that satisfies f and has no strict prefix that satisfies f.

Interval A finite sequence of states – a subsequence of a trace.

ITL Monitor (ITL-Monitor) The name given to the monitor algebra developed in this thesis. It is also used to refer to its Scala implementation.

Monitor A software device for analysing the trace of a program to determine if it satisfies a given specification.

Runtime verification The use of monitors to determine whether or not a program is behaving according to its specification while it is running.

Scala A contemporary programming language combining object-oriented and functional programming paradigms [Sca17].

Strict initial interval or **strict prefix** – either an empty interval or a prefix that does not include the final state.

(Execution) trace – a finite sequence of program states generated by a running program.

Contents

Chapter 1

Introduction

1.1 Motivation

Program verification is a key activity in the software development lifecycle. Traditional software testing which includes many established approaches including functional and structural testing, equivalence partitioning, prime path analysis, etc. comprises an established body of knowledge within the industry, e.g., [AO08].

One approach that is particularly suitable for the analysis of critical systems is *model checking*. A formal specification of the required temporal properties is constructed, usually in a Linear Temporal Logic (LTL). This, and an abstract model of the system, are translated into (typically) Büchi automata – finite state automata that can accept infinite paths. The method tests every path through the automaton to ensure that it passes through at least one accepting state infinitely often. Such analysis is typically undertaken before a system is deployed because it verifies all possible execution traces that the system could generate.

By contrast, runtime verification is a lightweight approach that analyses a *single* execution trace generated by a program while it is running. It is particularly useful in situations when it is infeasible to conduct model checking, or when it is necessary to provide extra assurance that a specific program run does not violate its temporal requirements. Faults are discovered as they arise which leads to action being taken such as noting the fault; reacting and adapting the system's behaviour; or halting its execution.

Like model checking, runtime verification specifications are written in a formal language, and LTL is very widely used. Temporal properties such as 'whenever p occurs then q must follow' (liveness), or 'p and q must never hold at the same time' (safety) can be checked. However, runtime verification checks finite traces, and the interpretation of liveness over finite traces is different from its interpretation over infinte traces.[1]

Another temporal logic that has been used for runtime verification is Interval Temporal Logic (ITL) [Mos83]. ITL has constructs that are similar to those found in computer programs such as sequence, parallel composition, iteration, and variable assignment [MM84]. In [Mos86] an executable subset of ITL, called Tempura, was defined along with an outline algorithm for an interpreter. This provides an environment in which ITL formulae can be checked by executing them. Tempura has been applied to runtime verification using AnaTempura [Cau07], a tool that runs the program under test with Tempura concurrently executing the specification. The requirement for Tempura to be executable places a restriction on its design such that it requires the user to specify the values of program variables completely, and to state explicitly when program termination occurs.

The motivation for the work in this thesis was to investigate whether ITL could be used for runtime verification but without the executability constraints required by Tempura. It was immediately apparent that ITL could not be used without restriction because its non-deterministic chop (sequential composition) operator would lead to exponential performance growth as the number of chops increased. The approach was to specify a deterministic partitioning of the execution trace into a sequence of subintervals, each of which could be verified with an arbitrary ITL formula.

It soon became clear that the restriction necessary to specify the deterministic subintervals was not a restriction on chop but a restriction on its first operand. States are input to a runtime verification monitor sequentially and the subinterval ends as soon as the left-hand formula is satisfied. At this point the next subinterval begins and monitoring continues with the right-hand formula. This motivated the need to find an operator that restricted a formula in such a way that if an interval satisfied it, then no strict prefix did. This led to the definition of the new, derived, first-occurrence operator in ITL. As the underpinning ITL construct for the runtime monitors, it was necessary to explore the mathematical properties of the first occurrence operator in depth. This has added to the body of knowledge about ITL. Antonio Cau has encoded ITL in Isabelle[2] and documented a large number of ITL laws along with their mechanical proofs [CMS19]. These laws are drawn predominantly from work by Moszkowski and Cau. However, Chapters 6 and 7 of the document contain all of the laws developed as part of this thesis, along with their mechanical proofs.

A further motivation was to consider how such a language could be mapped directly into code thus enabling such restricted ITL formulae to be synonomous with the monitors that verify them. This eliminates the need for any specification preprocessing common to many existing runtime verification systems. Recent research in runtime verification has proposed that using

[1] The difference is discussed later on page 15.

[2] A generic proof assistant [oCM18].

Domain Specific Languages (DSLs) to support monitor construction is a very promising approach [BH11, FHR13] and that Scala is a suitable development language [AHKY15].

The research was undertaken firstly by constructing a mathematical model of the monitors, and exploring its properties. Then a software tool, written as a Scala DSL, was implemented and evaluated by constructing a number of case studies.

1.2 Contributions

The contributions of the thesis are listed below:

- The introduction of the operators $\boxed{s}$, $\diamondsuit_s$, and especially, first-occurrence ($\triangleright$) into ITL (see Chapter 3). These operators have been thoroughly investigated and, in particular, the properties of $\triangleright$ in relation to itself and other ITL operators including chop and chopstar.

  The theory includes the important law $\vdash \triangleright(\triangleright f \; ; \; g) \equiv \triangleright f \; ; \; \triangleright g$ which states that the sequential composition of first occurrences is itself a first occurrence. This is a significant result which has an important corollary $\vdash \triangleright \triangleright f \equiv \triangleright f$ which states that a first occurrence of f is itself a first occurrence.

  A comprehensive set of related theorems and fully worked proofs has been developed. These are presented in full in Chapter 6 of [CMS19].

- The construction of ITL-Monitor - a language whose operators combine $\triangleright$-restricted ITL formulae while preserving their first-occurrence properties. The operators satisfy a range of algebraic laws which have been discovered and categorised. All the theorems and their associated proofs have been fully worked out and are presented in full in Chapter 7 of [CMS19].

- Two Domain Specific Libraries written in Scala for implementing ITL expressions and formulae and ITL-Monitors: `ITL.scala` and `Monitor.scala`. The latter provides an API for constructing runtime monitors that verify the ITL formulae they represent.

- A demonstration that it is feasible to use ITL-Monitor to perform runtime verification on systems generating large numbers of states (an execution trace of c. a million states has been verified successfully). An execution trace that was partitioned into relatively short individual subintervals (around 300 states or fewer) was able to verify each of these subintervals against a formula with up to $\mathcal{O}(n^3)$ evaluation complexity in less than a tenth of a second.

1.3 Outline of thesis

Chapter 2 introduces the field of runtime verification. The syntax and semantics of two temporal logics are given. The first, Linear Temporal Logic (LTL), is used extensively in both model checking and runtime verification. The second, Interval Temporal Logic (ITL), is an extension of linear-time temporal logic and is the underlying logic used in this thesis. The semantics of LTL is defined over infinite paths and then extended to finite paths. The motivation for this is that runtime verification analyses an evolving execution trace which comprises a set of prefix-closed, finite intervals. The chapter discusses finite intervals in the context of runtime verification and argues for a partitioning of the execution trace into a sequence of finite intervals – a topic which is developed in the subsequent chapter. Traditional and contemporary approaches to runtime verification are discussed with particular emphasis on two software tools that will be used for comparison with the tool developed as part of this thesis.

Chapter 3 discusses two key distinguishing operators in ITL, chop (;) and chopstar (*), used for sequencing and repeating ITL formulae. Chop is a nondeterministic operator which requires that for some formula $f\ ;\ g$ a satisfying interval can be divided into a prefix and a suffix over which f and g hold respectively. The subintervals must share one chop (or fusion) point but this need not be determined uniquely. Chopstar can introduce multiple nondeterministic fusion points and the task of locating a set of such points in order to satisfy a formula has exponential complexity. This can be mitigated by defining a deterministic chop operator to specify a unique partitioning of the execution trace [BB08]. This thesis performs this task by defining a new first-occurrence operator $\triangleright$. This operator is independent of chop although combining it with chop to determine fusion points is its primary rôle. The concept of first-occurrence provides the basis for constructing runtime monitors. Consequently this operator can be combined with arbitrary ITL formulae and the chapter explores in depth the algebraic properties of $\triangleright$ and how it combines with other ITL operators.

Chapter 4 introduces ITL-Monitor, the compositional runtime verification language which is the subject of this thesis. ITL-Monitor is described in two ways. Firstly, ITL-Monitor is a language for constructing ITL formulae that preserve first-occurrence properties. A syntax is defined for a core language, and a translation function into ITL formulae is given. An informal discussion of the behaviour of the core operators is given in terms of ITL, and then a set of derived operators is introduced along with a justification for each. The algebraic properties of monitors are explored and a number of algebraic structures are presented. An ITL-Monitor also represents an executable, runtime monitor and the chapter describes the rôles of the various operators in this context. The application of Moszkowski's importable assumptions and exportable commitments [Mos96a, Mos98] to maintaining invariant properties over sequences of subintervals is presented. The chapter concludes with a demonstration of how the ITL-

Monitor operators can be combined to construct a runtime verification monitor for a small application.

Chapter 5 describes the implementation of ITL-Monitor as an API in Scala. Two basic data structures are introduced: one for representing ITL formulae, and the other for representing ITL-Monitor expressions. Details of their representation in Scala are provided with particular emphasis on how specific language features such as pattern matching, existential types, and infix method notation, have been exploited. Efficiency considerations are described in respect of the implementations of both libraries. Finally, the translation of monitors into a network of Akka [Akk17, Wya13] actors is given and the operation of this system is illustrated using two example monitors and a step-by-step animation.

Chapter 6 introduces two example systems. The first (latch) example provides a comparative analysis of ITL-Monitor with two other runtime verifcation tools: TraceContract and AnaTempura. Requirements specifications are presented using each of the tools' individual notations and these are compared with each other. The example is coded in Scala and executed multiple times with different combinations of the tools providing a runtime verification. The results and timing data are presented and analysed. The capability of each tool to provide feedback when the verification fails is also discussed. The second (checkout) example is used to generate a large quantity of state data to enable the performance of ITL-Monitor to be measured under stress. In particular, the lengths of subintervals that could be monitored effectively by formulae with $\mathcal{O}(n^3)$ and $\mathcal{O}(n^4)$ complexities are investigated. It is demonstrated that the system's performance scales linearly as the execution trace length is increased and this is shown for up to c. 1m states partitioned into 12000 subintervals.

Finally, Chapter 7 summarises the thesis and discusses future research potential.

Chapter 2

Runtime verification

Runtime verification is a method in which a computer program[1] is monitored while it is executing to determine whether or not it satisfies or violates certain correctness criteria. These are often safety properties expressed using a formal language such as, for example, temporal logic. Runtime verification also describes, more widely, a discipline of computer science whose "distinguishing research effort lies in synthesizing monitors from high level specifications" [Leu12].

An executing program generates a sequence of states which is analysed for the purpose of runtime verification.

Definition 2.1 Program state *The state of the program at a given point in time is the mapping of its variables to their current values.*

A program's behaviour may be understood in terms of how it modifies state. When an instruction causes a change to one or more of the state variables, then a new state is generated. In this way the program's execution trace can be represented as a sequence of states, $\langle s_0, s_1, \dots \rangle$.

Definition 2.2 Execution trace *An execution trace is a finite sequence of program states generated by a running program.*

While all of the variables which comprise the state are significant to the program's operation, only a subset of these may be relevant to the specification. Let a state containing only these monitored variables be denoted by σ_i. As the program is executed, an execution trace, σ, is generated: $\langle \sigma_0, \sigma_1, \dots \rangle$.

[1] The term *program* is used here to represent any unit of code whose behaviour is being verified. It could, for example, be a single subroutine, or a collection of concurrent processes, or any executable components that make up a system under scrutiny.

Definition 2.3 Runtime verification *A method by which an executing program is checked continuously for adherence to, or violation of, specified correctness properties.*

Runtime verification can be compared both to traditional software testing techniques and to model checking. However, there is a stronger relationship with the latter, due to the primacy of a formal specification, typically written in temporal logic. In contrast, traditional software testing techniques (e.g. see [AO08]) do not require formal methods, although formal methods and testing have been combined in, e.g., [BBC$^+$02, Hie02].

An important distinction between model checking and runtime verification is that the former performs an analysis on all possible runs of a program, whereas the latter only analyses one specific run at a time. For this reason runtime verification is considered to be a *lightweight* verification method. Runtime verification may be employed in conjunction with other verification techniques, sometimes as an important extra check, in order to maintain confidence in a particular program run.

Major developments in runtime verification appear in the *International Conference on Runtime Verification* which began as a workshop series in 2001 and became an annual international conference in 2010. Areas of particular interest include formal specification languages, temporal logics, runtime verification methods, and tool support.

Section 2.1 introduces Linear Temporal Logic (LTL) and Interval Temporal Logic (ITL), both of which are used to specify temporal behaviour for runtime verification, the latter being the logic used primarily in this thesis. Section 2.2 discusses intervals in the context of runtime verification. Section 2.3 considers two languages, MetateM, based on LTL, and Tempura, a deterministic subset of ITL, which are used to animate specifications.[2] Section 2.4 describes the principal architectures for runtime verification and discusses two runtime verification tools that are used for comparison with the current work.

2.1 Temporal logic

Temporal logic is relevant to both model checking and runtime verification. An exposition of temporal logics and how they are used in runtime verification is presented in [Fis11], and a discussion of the classification of temporal logics appears in [Eme90] (Chapter 16). This section covers Linear Temporal Logic (LTL), the primary temporal logic used in model checking and many runtime verification systems; and Interval Temporal Logic (ITL), which is the basis for the work in this thesis.

[2]Animation can be used to explore the behaviour of a specification interactively before it is included within a runtime verification.

2.1.1 Linear temporal logic

Linear Temporal Logic (LTL) [Pnu77, MP92] was first introduced in the context of program verification, as a language for expressing the temporal relationships between variables in a computer program.

Formulae in LTL are constructed from a finite set of propositional variables, P, which are normally written as lower-case alphanumeric symbols (possibly including underscores), e.g., p, q_2, is_on; the Boolean constants true and false; the propositional connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$; and the temporal connectives $\bigcirc$, $\Diamond$, $\Box$, $\mathcal{U}$, $\mathcal{W}$. Parentheses can be used to resolve ambiguity when combining operators.

The set of well-formed LTL formulae is defined inductively:

- Any propositional variable $p \in P$ is an LTL formula
- Either of the constants true and false is an LTL formula
- If φ and ψ are LTL formulae, then so are: $\neg\,\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\varphi \Leftrightarrow \psi$, $\varphi\,\mathcal{U}\,\psi$, $\varphi\,\mathcal{W}\,\psi$, $\Diamond\varphi$, $\Box\varphi$, $\bigcirc\varphi$, (φ)

The semantics of LTL is defined over infinite paths $\pi = \langle \pi_0, \pi_1, \pi_2, \dots \rangle$ in which each π_i is a state characterised by the set of propositions that are true at the i^{th} moment in time. The model of time is discrete and each state π_i has a successor, or *next* state, π_{i+1}. The initial state has index 0.

The semantics is defined using an interpretation function $\models$ which maps a path π, an index $i \geq 0$, and a well-formed formula φ, to a value in the set of Boolean values $\mathbb{B} = \{\top, \bot\}$. If a formula φ holds at index i then $((\pi, i) \models \varphi) = \top$, abbreviated to $(\pi, i) \models \varphi$; and if φ does not hold at index i then $((\pi, i) \models \varphi) = \bot$, abbreviated to $(\pi, i) \not\models \varphi$. The semantics of LTL formulae is given in Figure 2.1:

Constant
$(\pi, i) \models \mathsf{true}$

Propositions
$(\pi, i) \models p$ iff $p \in \pi_i$

Propositional operators
$(\pi, i) \models \neg\, \varphi$ iff $(\pi, i) \not\models \varphi$
$(\pi, i) \models \varphi_1 \wedge \varphi_2$ iff $(\pi, i) \models \varphi_1$ and $(\pi, i) \models \varphi_2$

Temporal operators
$(\pi, i) \models \bigcirc\varphi$ iff $(\pi, i+1) \models \varphi$
$(\pi, i) \models \varphi_1 \,\mathcal{U}\, \varphi_2$ iff there exists $j \geq i$ such that $(\pi, j) \models \varphi_2$ and for all $i \leq k < j$, $(\pi, k) \models \varphi_1$

Figure 2.1: LTL semantics

The *next* formula, $\bigcirc\varphi$, holds in state i if φ holds in state $i+1$. The existence of a next state is guaranteed because the semantics is defined over infinite paths. The *until* formula, $\varphi_1 \,\mathcal{U}\, \varphi_2$, holds in state i if φ_2 holds in some future state $j \geq i$, and φ_1 holds throughout all the states k where $i \leq k < j$. Importantly, φ_2 is *guaranteed* to be satisfied at some future state and for this reason $\mathcal{U}$ is referred to as the *strong until* operator. The semantics allows φ_1 to hold in the same state that φ_2 holds but does not require it. Furthermore, it is possible for φ_2 to hold 'immediately' in which case φ_1 holds vacuously over an empty interval. Below is a list of derived operators:

$$
\begin{array}{ll}
\mathsf{false} & \widehat{=}\ \neg\ \mathsf{true} \\
\varphi_1 \vee \varphi_2 & \widehat{=}\ \neg\,(\neg\,\varphi_1 \wedge \neg\,\varphi_2) \\
\varphi_1 \Rightarrow \varphi_2 & \widehat{=}\ \neg\,\varphi_1 \vee \varphi_2 \\
\varphi_1 \Leftrightarrow \varphi_2 & \widehat{=}\ (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1) \\
\Diamond\varphi & \widehat{=}\ \mathsf{true}\,\mathcal{U}\,\varphi \\
\Box\varphi & \widehat{=}\ \neg\,\Diamond\neg\,\varphi \\
\varphi_1 \,\mathcal{W}\, \varphi_2 & \widehat{=}\ (\varphi_1 \,\mathcal{U}\, \varphi_2) \vee (\Box\varphi_1)
\end{array}
$$

The main derived temporal operators are further described below.

Eventually $\Diamond\varphi$
Eventually φ will hold.
$(\pi, i) \models \Diamond\varphi$ iff there exists $j \geq i$ such that $(\pi, j) \models \varphi$

Always $\Box\varphi$
φ will always hold from this point.
$(\pi, i) \models \Box\varphi$ iff for all $j \geq i$, $(\pi, j) \models \varphi$

Weak until $\varphi_1 \mathcal{W} \varphi_2$

φ_1 must hold either until φ_2 holds, or forevermore if φ_2 never holds in the future.
$(\pi, i) \models \varphi_1 \mathcal{W} \varphi_2$ iff either $(\pi, i) \models \varphi_1 \mathcal{U} \varphi_2$ or $(\pi, i) \models \Box\varphi_1$
Note that $\vdash \varphi \mathcal{W} \mathsf{false} \Leftrightarrow \Box\varphi$.

A significant application area for LTL is model checking in which a model of a system is constructed that represents the set of infinite paths (sequences of states) that collectively encode all possible runs of the system. The goal of model checking is to establish that every one of these paths satisfies a given correctness property. The number of such paths increases combinatorially as the number of reachable states increases. Model checking, $M \models \varphi$, and validity, $\vdash \varphi$, in LTL are in the complexity class PSPACE [Fis11].

In contrast to model checking, runtime verification focuses on checking a single path – an execution trace. Consequently, as noted by [ZZC05], the issue of combinatorial complexity does not arise in this case. Runtime verification is not a substitute for model checking, but can be used as a complementary tool, or when model checking may be infeasible.

To introduce the comparison between model checking and runtime verification, an outline of the model checking process is presented below. It uses LTL with infinte path semantics. This is followed by a discussion of LTL with finite (truncated) paths which arise within the context of runtime verification.

Let Σ^ω represent the set of all possible, potentially infinite execution paths of a program S. Then model checking requires that $\forall \pi \in \Sigma^\omega.\ (\pi, 0) \models \varphi$. If, for some path π, $(\pi, 0) \models \varphi$ then π is a model of S that satisfies φ.

Model checking uses finite state automata to represent all (possibly infinite) paths of the program under test, and the temporal state transitions that satisfy the required temporal formula. To encode infinite paths it is necessary to use a class of automata called ω-*automata* which can accept infinite words. An infinite word is accepted if the word describes at least one run through the automaton that passes through at least one accepting state infinitely often. Büchi automata are a class of such ω-*automata* that are used in model checking.

A Büchi automaton, BA, is defined as $BA = \langle A, S, \delta, I, F \rangle$ where A is an alphabet; S is a finite set of states; $\delta : S \times A \times S$ is a transition relation; $I \in S$, is a set of initial states; and $F \in S$, is a set of final states. Each letter is interpreted as a set of propositions: thus A is the powerset of a given set of propositions. A word represents an execution sequence of states in which each state is a set of propositions that hold in that state.

LTL can express a range of temporal properties, many of which are classified in [MP87], and a selection of which is presented in Figure 2.2 for illustration.

Classification	Typical formula
Invariance (Safety)	$\Box\varphi$ (or $\Box\neg\,\varphi$)
Guarantee (Liveness)	$\Diamond\varphi$
Persistence (Stability)	$\Diamond\Box\varphi$
Recurrence (Progress)	$\Box\Diamond\varphi$
Obligation (Correlation)	$\Diamond\varphi_1 \Rightarrow \Diamond\varphi_2$
Response	$\varphi_1 \Rightarrow \Diamond\varphi_2$
Precedence	$\neg\,\varphi_1\,\mathcal{U}\,\varphi_2$

Figure 2.2: Some patterns of LTL temporal properties [MP87]

For example, suppose the alphabet is given by $A = \mathbb{P}\{a, b\}$, then the LTL formula $a \wedge \bigcirc\Diamond\Box b$ can be translated into the following Büchi automaton. (N.B., the shorthand notation a is used within the transition relation δ to represent *any* set of propositions containing a.)

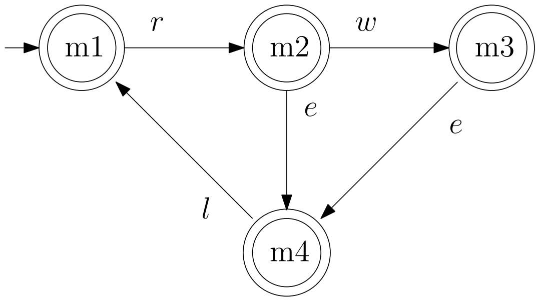
$$\begin{aligned}
A &= \mathbb{P}\{a, b\}\\
S &= \{s1, s2, s3\}\\
\delta &= \{(s1, a, s2), (s2, \mathsf{true}, s2), (s2, b, s3), (s3, b, s3)\}\\
I &= \{s1\}\\
F &= \{s3\}
\end{aligned}$$

The automata-based approach to model checking proceeds as follows. The system under test, S, is modelled as a Büchi automaton, BA_S. This represents all of the possible paths that could be generated by S. The temporal property that every run of S must satisfy is expressed as an LTL formula φ and its *negation* is translated into a Büchi automaton, $BA_{\neg\,\varphi}$. It is necessary to establish that the set of paths accepted by BA_S is a subset of the set of paths accepted by BA_φ. This condition can be established by checking that the intersection of the set of paths accepted by BA_S and the set of paths by $BA_{\neg\,\varphi}$ is empty. If every state in BA_S is made to be accepting, then $\forall\,\pi \in \Sigma^\omega.\ (\pi, 0) \models \varphi$ can be established by determining that the automaton $BA_S \times BA_{\neg\,\varphi}$ is empty. The accepting states of $BA_S \times BA_{\neg\,\varphi}$ will be precisely those containing the acceptance states of $BA_{\neg\,\varphi}$ and thus represent precisely those runs of S that satisfy $\neg\,\varphi$: i.e. 'bad states'. If the set of such paths is empty then the model checking succeeds.

Once constructed, the automaton $BA_S \times BA_{\neg\,\varphi}$ can be reduced using the following set of rules which are quoted from [Fis11] (page 34): "(i) remove transitions that contain contradictions (e.g. $a \wedge \neg\,a$); (ii) remove nodes that have no transitions emanating from them; (iii) remove terminal, non-accepting sets of nodes." These steps are applied repeatedly until none applies. If the resulting graph is empty then the temporal property was satisfied. To complete this discussion, a small example of the technique is provided below.

Example 2.1.1 Consider a program in which an agent can request to enter a particular state.

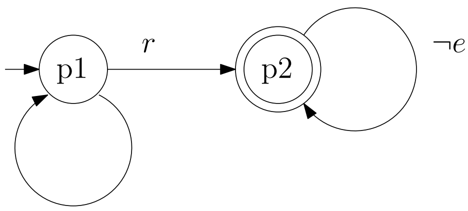
Following the request, the agent may enter the state or may have to wait before entering. An agent that has entered must subsequently leave and the process repeats. Using letters to represent the propositions (r = *requested*, w = *waiting*, e = *entered*, l = *left*), the behaviour is captured in the Büchi automaton shown in Figure 2.3.



$$
\begin{array}{lcl}
A_{BA_S} & = & \mathbb{P}\{r, w, e, l\} \\
S_{BA_S} & = & \{m1, m2, m3, m4\} \\
\delta_{BA_S} & = & \{(m1, r, m2), (m2, w, m3), (m2, e, m4), (m3, e, m4), (m4, l, m1)\} \\
I_{BA_S} & = & \{m1\} \\
F_{BA_S} & = & \{m1, m2, m3, m4\}
\end{array}
$$

Figure 2.3: The Büchi automaton BA_S.

It is required that this process satisfies the temporal property that whenever a request to enter is made then entry is guaranteed at some point thereafter. The temporal property can be expressed in LTL: $\varphi = \Box(r \Rightarrow \bigcirc\Diamond e)$. The negation of this formula is given by: $\neg\,\varphi = \Diamond(r \wedge \bigcirc\Box\neg\, e)$.[3] The Büchi automaton for $\neg\,\varphi$ is shown in Figure 2.4.



$$
\begin{array}{lcl}
A_{BA_{\neg\varphi}} & = & \mathbb{P}\{r, w, e, l\} \\
S_{BA_{\neg\varphi}} & = & \{p1, p2\} \\
\delta_{BA_{\neg\varphi}} & = & \{(p1, \mathsf{true}, p1), (p1, r, p2), (p2, \neg\, e, p2)\} \\
I_{BA_{\neg\varphi}} & = & \{p1\} \\
F_{BA_{\neg\varphi}} & = & \{p2\}
\end{array}
$$

Figure 2.4: The Büchi automaton $BA_{\neg\varphi}$.

Figure 2.5 shows the combined Büchi automaton $BA_S \times BA_{\neg\varphi}$. Unreachable states have been

[3] Negation can be moved inside next: $\neg\bigcirc\varphi \Leftrightarrow \bigcirc\neg\,\varphi$. $\Diamond$ and $\Box$ are duals: $\neg\,\Box\varphi \Leftrightarrow \Diamond\neg\,\varphi$, and $\neg\,\Diamond\varphi \Leftrightarrow \Box\neg\,\varphi$

removed. Notice that state $m3p2$ has no transitions emanating from it and therefore it is a candidate for removal. Consequently, its removal makes $m2p2$ the next candidate for removal. The resulting graph is a terminal non-accepting set of nodes which can all be deleted to leave an empty graph. Thus the model checking succeeds. ■

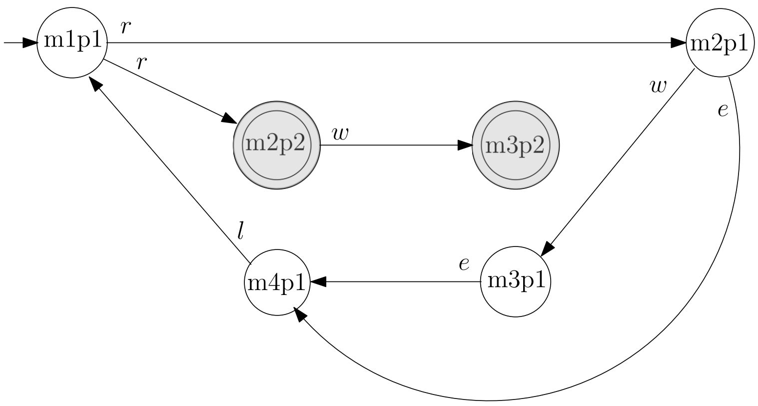


Figure 2.5: The Büchi automaton $BA_S \times BA_{\neg\varphi}$ with nodes marked for removal.

The process can be automated using a model checker such as SPIN [Spi17, Hol04].

2.1.2 LTL with finite paths

With infinite paths semantics $\bigcirc\varphi$ is always defined. However, for a finite path the meaning of $\bigcirc\varphi$ in the final state needs to be defined. Also, the meaning of $\varphi_1 \mathcal{U} \varphi_2$ needs to be defined for a finite interval. A *weak next* operator ⓦ is introduced in which ⓦ φ holds in the final state and has the same meaning as $\bigcirc\varphi$ in all the preceding states[4]. The semantics of these operators appears in Figure 2.6. Note how the definition of $\varphi_1 \mathcal{U} \varphi_2$ requires that φ_2 *must* hold in the final state, if not before.

$$
\begin{array}{ll}
(\pi, i) \models \bigcirc\varphi & \text{iff} \begin{cases} (\pi, i+1) \models \varphi & \text{if } i+1 < n \\ \bot & \text{otherwise} \end{cases} \\
(\pi, i) \models ⓦ\, \varphi & \text{iff} \begin{cases} (\pi, i+1) \models \varphi & \text{if } i+1 < n \\ \top & \text{otherwise} \end{cases} \\
(\pi, i) \models \varphi_1 \mathcal{U} \varphi_2 & \text{iff } (\pi, i) \models \varphi_2 \text{ or } ((\pi, i) \models \varphi_1 \text{ and } (\pi, i) \models \bigcirc(\varphi_1 \mathcal{U} \varphi_2))
\end{array}
$$

where n is the length of the path.

Figure 2.6: LTL finite semantics (overrides temporal operators in Figure 2.1)

[4] This presentation uses the symbol ◯ for strong next and ⓦ for weak next. Other symbols commonly used for these operators are X and $\bar{X}$.

Runtime verification analyses an execution path each time that a new state, π_i, is generated. Thus a sequence of prefix paths is produced $\pi^0 = \langle \pi_0 \rangle$, $\pi^1 = \langle \pi_0, \pi_1 \rangle$, $\ldots$. Suppose that π represents the (possibly infinite) execution path for a run of program S. Then each π^k, $0 \leq k$, is a finite prefix of π, and each prefix path π^{k+1} represents a 'better' approximation of π than its predecessor π^k. These finite, prefix paths are produced by executing programs as each new state is generated, and the set of paths $\{\pi^0, \pi^1, \ldots \pi^k\}$ is a prefix-closed set. Such finite prefixes constitute *truncated paths.*

Consider the evolving path as a program executes. Certain properties may be established globally on the basis of evidence provided by a finite, partial prefix. For example, the *safety* property, $\square(\neg\, b)$, is violated as soon as an instance of b is detected. In this case the prefix path has provided sufficient evidence to establish that the condition is violated – it may not subsequently be judged to have been satisfied. Conversely, the liveness property, $f \Rightarrow \Diamond g$, can be established if f has been observed and then, later, g holds. The prefix path (up to g) has provided sufficient evidence to establish that the condition is satisfied. By contrast, consider a liveness property such as $\square(f \Rightarrow \Diamond g)$. No finite prefix can determine the correctness of this claim.

Liveness in the context of finite path semantics is interpreted slightly differently from liveness in the context of infinite path semantics. Consider the LTL formula $\Diamond\varphi$. The infinite LTL semantics (Figure 2.1) requires that φ holds at some point in the future. However, in the case of a finite semantics (Figure 2.6), φ must hold at some point up to the final state. Liveness properties over infinite paths can be established by model checking. However, in the context of runtime verification, a liveness property can only be checked for a specific program run by observing φ before the end of the execution trace is reached.

Analysis of truncated paths has the potential to produce misleading judgements. For example, the formula $\Diamond\varphi$ may be false over paths $\pi^0, \pi^1 \ldots \pi^k$, but true over π^{k+1}. An analysis of a truncated path can only provide a judgement on the basis of the information available up to a certain point in time. [EFH+03, BLS07, LS09, BLS11] have proposed using three- and four-valued temporal logics to deal with the potentially misleading nature of premature judgements over truncated paths.

[LS09] introduces LTL_3, a three-valued logic, in which the satisfaction function returns one of $\{\top, \bot, ?\}$ where ? represents 'inconclusive'. Consider a finite word w, and its concatentation with an infinite word u, written $w \cdot u$, and using the relation $\models_3$ to represent satisfaction in LTL_3, then $w \models_3 \varphi$ is defined as $\top$ if for all u, $w \cdot u \models_3 \varphi$; $\bot$ if for all u, $w \cdot u \not\models_3 \varphi$; and ? if neither $\top$ nor $\bot$ can be established based on w. For example, if $\neg\, p$ holds throughout w then $w \models_3 \square\neg\, p = ?$ because p may hold in a future state. Conversely, if p holds at some point within w, then $w \models_3 \Diamond p = \top$, and $w \models_3 \square\neg\, p = \bot$.

In [BLS07] the authors develop a refinement of LTL_3 called *RV-LTL* in which a four-valued logic, B_4, is introduced: $B_4 = \{\bot, \bot^p, \top^p, \top\}$. The values represent *false*, *presumably false*, *presumably true*, and *true* respectively. The syntax of *RV-LTL* formulae is given inductively by:

$$\varphi ::= \mathsf{true} \mid p \mid \neg\, \varphi \mid \varphi \vee \varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \bigcirc\varphi \mid \circledw\, \varphi \qquad \text{(where } p \in P\text{)}$$

The semantics of *RV-LTL* is derived from LTL_3 and the following definition is taken from [BLS07]:

> Let $\pi \in \Sigma^*$ denote a finite path of length $n = |\pi|$. The truth value of an *RV-LTL* formula φ wrt π at position $i < n$, denoted by $(\pi, i) \models_{RV} \varphi$, is an element of B_4 and is defined as follows:
>
> $$(\pi, i) \models_{RV} \varphi = \begin{cases} \top & \text{if } (\pi, i) \models_3 \varphi' = \top \\ \bot & \text{if } (\pi, i) \models_3 \varphi' = \bot \\ \top^p & \text{if } (\pi, i) \models_3 \varphi' = ? \text{ and } (\pi, i) \models \varphi = \top \\ \bot^p & \text{if } (\pi, i) \models_3 \varphi' = ? \text{ and } (\pi, i) \models \varphi = \bot \end{cases}$$
>
> where φ' is obtained from φ by replacing each $\circledw$ operator with $\bigcirc$.

[BLS07] argue that logics for runtime verification should not evaluate to $\top$ or $\bot$ prematurely, but should evaluate to $\top$ or $\bot$ as soon as possible. These properties are referred to as *impartiality* and *anticipation* respectively.

ITL-Monitor, the runtime verification language which is the subject of this thesis, delivers three judgements: DONE, FAIL, and MORE. These correspond to the LTL_3 verdicts $\top$, $\bot$, and ?, respectively. In future work it would be possible to explore the efficacy of using a greater number of potential judgements. For example, one could consider *RV-LTL*'s $\top^p$ and $\bot^p$, or, relatedly, the five-valued judgements used by RuleR [BHRG09]: `{TRUE, STILL_TRUE, STILL_FALSE, FALSE, UNKNOWN}` in which `STILL_TRUE` relates to a (safety) property not yet falsified; `STILL_FALSE` relates to a (liveness) property not yet satisfied; and `UNKNOWN` relates to a condition that does not fit the other criteria such as a combination of `STILL_TRUE` and `STILL_FALSE`.

2.1.3 Interval temporal logic

Interval Temporal Logic (ITL) [Mos82, Mos83, CZCM96, CM16, CMS19] provides an alternative formalism for specifying runtime system behaviour. Propositional and first-order variants of ITL have been developed for finite and infinite path semantics. In this thesis first order, finite ITL is used. The significance of using a finite path temporal logic for runtime verification was discussed in Section 2.1.2. In first order ITL validity is not decidable. However, it is possible to check if a given model satisfies a first order ITL formula, and this is exactly the requirement for runtime verification. This same argument is made by [BRH07] in respect of another runtime verification logic, EAGLE, discussed later in Section 2.4.3.

ITL [CM16, CMS19] is defined over finite intervals (non-empty sequences of states), $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_{|\sigma|}$, in which each state is the union of the mapping from the set of integer variables $IntVar$ to $\mathbb{Z}$, and the mapping from propositional variables $PropVar$ to the Boolean values $Bool = \{\mathsf{tt}, \mathsf{ff}\}$. $IntVar \cup BoolVar$ is the set of alphanumeric identifiers, which may contain underscores and subscripts, beginning with an uppercase letter.

The syntax of ITL expressions, e, and ITL formulae, f is presented below. The definitions are taken from [CM16]:

Integer Expressions	ie	$::=$	$z \mid A \mid ig(ie_1, \ldots, ie_n) \mid \bigcirc A \mid \mathsf{fin}\ A$
Boolean Expressions	be	$::=$	$b \mid Q \mid bg(be_1, \ldots, be_n) \mid \bigcirc Q \mid \mathsf{fin}\ Q$
Formulae	f	$::=$	$\mathsf{true} \mid h(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid$
			$\mathsf{skip} \mid f_1 \ ; \ f_2 \mid f^*$

where	z	denotes an integer value
	A	denotes an integer variable (can change within an interval)
	b	denotes a Boolean value
	Q	denotes a propositional variable
	ig	denotes an integer function symbol (e.g $+$ and $\times$)
	bg	denotes a Boolean function symbol (e.g $\wedge$ and $\vee$)
	v	denotes a Boolean or integer variable
	e_i	denotes a Boolean or integer expression
	h	denotes a predicate symbol. (e.g. $\leq$ and $=$)

Temporal formulae are interpreted over a finite interval. Formulae can be composed sequentially using the *chop* operator, (e.g. $f \ ; \ g$), and iteratively using the *chopstar* operator, (e.g. f^*). In the semantics that follows:

$\mathcal{E}[\![\ldots]\!](\sigma)$ is the semantic function: $Expressions \times \Sigma^+ \rightarrow \mathbb{Z}$.

$\mathcal{F}[\![\ldots]\!](\sigma)$ is the semantic function: $Formulae \times \Sigma^+ \rightarrow Bool$.

$\sigma = \langle \sigma_0, \sigma_1 \ldots \rangle$ is an interval.

$\sigma_i(A)$ represents the value associated with the state variable A in state σ_i.

$\sigma \sim_v \sigma'$ means that the intervals σ and σ' are identical with the possible exception of their mappings for the variable v.

The semantics is given inductively over the structure of ITL expressions and formulae. This is also taken from [CM16]:

$$
\begin{array}{ll}
\mathcal{E}[\![z]\!](\sigma) & = z \\
\mathcal{E}[\![A]\!](\sigma) & = \sigma_0(A) \\
\mathcal{E}[\![ig(ie_1, \ldots, ie_n)]\!](\sigma) & = ig(\mathcal{E}[\![ie_1]\!](\sigma), \ldots, \mathcal{E}[\![ie_n]\!](\sigma)) \\
\mathcal{E}[\![\bigcirc A]\!](\sigma) & = \begin{cases} \sigma_1(A) & \text{if } |\sigma| > 0 \\ \text{any } x \text{ s.t. } x \in \mathbb{Z} & \text{otherwise} \end{cases} \\
\mathcal{E}[\![\mathsf{fin}\ A]\!](\sigma) & = \sigma_{|\sigma|}(A) \\
\mathcal{E}[\![b]\!](\sigma) & = b \\
\mathcal{E}[\![Q]\!](\sigma) & = \sigma_0(Q) \\
\mathcal{E}[\![bg(be_1, \ldots, be_n)]\!](\sigma) & = bg(\mathcal{E}[\![be_1]\!](\sigma), \ldots, \mathcal{E}[\![be_n]\!](\sigma)) \\
\mathcal{E}[\![\bigcirc Q]\!](\sigma) & = \begin{cases} \sigma_1(Q) & \text{if } |\sigma| > 0 \\ \text{any } x \text{ s.t. } x \in \mathit{Bool} & \text{otherwise} \end{cases} \\
\mathcal{E}[\![\mathsf{fin}\ Q]\!](\sigma) & = \sigma_{|\sigma|}(Q) \\
\mathcal{F}[\![\mathsf{true}]\!](\sigma) & = \mathsf{tt}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{F}[\![h(e_1, \ldots, e_n)]\!](\sigma) = \mathsf{tt} & \text{iff } h(\mathcal{E}[\![e_1]\!](\sigma), \ldots, \mathcal{E}[\![e_n]\!](\sigma)) \\
\mathcal{F}[\![\neg f]\!](\sigma) = \mathsf{tt} & \text{iff not}(\mathcal{F}[\![f]\!](\sigma) = \mathsf{tt}) \\
\mathcal{F}[\![f_1 \wedge f_2]\!](\sigma) = \mathsf{tt} & \text{iff } \mathcal{F}[\![f_1]\!](\sigma) = \mathsf{tt} \text{ and } \mathcal{F}[\![f_2]\!](\sigma) = \mathsf{tt} \\
\mathcal{F}[\![\mathsf{skip}]\!](\sigma) = \mathsf{tt} & \text{iff } |\sigma| = 1 \\
\mathcal{F}[\![\forall\, v \bullet f]\!](\sigma) = \mathsf{tt} & \text{iff for all } \sigma' \text{ s.t. } \sigma \sim_v \sigma', \mathcal{F}[\![f]\!](\sigma') = \mathsf{tt} \\
\mathcal{F}[\![f_1\ ;\ f_2]\!](\sigma) = \mathsf{tt} & \text{iff (exists } k, \text{ s.t. } \mathcal{F}[\![f_1]\!](\sigma_0 \ldots \sigma_k) = \mathsf{tt} \text{ and } \mathcal{F}[\![f_2]\!](\sigma_k \ldots \sigma_{|\sigma|}) = \mathsf{tt}) \\
\mathcal{F}[\![f^*]\!](\sigma) = \mathsf{tt} & \text{iff (exists } l_0, \ldots, l_n \text{ s.t. } l_0 = 0 \text{ and } l_n = |\sigma| \text{ and} \\
& \quad \text{for all } 0 \le i < n, l_i \le l_{i+1} \text{ and } \mathcal{F}[\![f]\!](\sigma_{l_i} \ldots \sigma_{l_{i+1}}) = \mathsf{tt})
\end{array}
$$

The length of an interval σ, denoted $|\sigma|$, is equal to the number of states minus one. Thus a one-state interval is defined to have a length of zero.[5] The meaning of a state variable is given by its value in the *first* state of the interval (σ_0). The temporal formula skip represents an interval of unit length (i.e. two states). The formula $f_1\ ;\ f_2$ holds over an interval if the interval can be split into two subintervals: a prefix over which f_1 holds and a suffix over which f_2 holds. The prefix and suffix intervals thus obtained must share one common state which is simultaneously the *final state of the prefix* and the *initial state of the suffix*. The formula f^* holds over an interval if it is possible to split the interval into a series of subintervals each of which satisfies f: i.e. $f\ ;\ f\ ;\ \ldots\ ;\ f$.[6] These fundamental temporal formulae are illustrated in Figure 2.7.

[5] This interpretation of a single state representing an empty interval has always been part of ITL.

[6] Note that ; is associative.

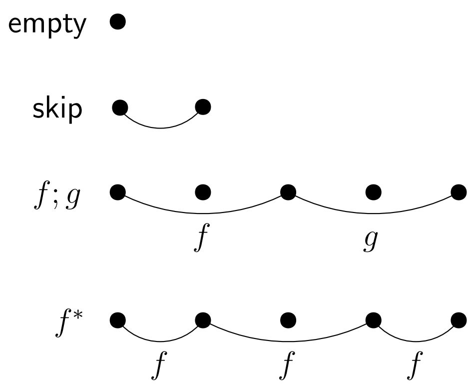


Figure 2.7: ITL operators empty, skip, chop, and chopstar

If $\mathcal{F}[\![f]\!](\sigma) = \mathsf{tt}$ then the formula f is *satisfied* by interval σ. This is written $\sigma \models f$. If the formula f is satisfied by all possible intervals then the formula is *valid*, written $\models f$.

2.1.3.1 Derived operators

The following operators are derived:[7]

$$
\begin{array}{lcll}
\mathsf{false} & \widehat{=} & \neg\, \mathsf{true} & \qquad \textit{FalseDef}^{(C.2)} \\
f_1 \vee f_2 & \widehat{=} & \neg\,(\neg\, f_1 \wedge \neg\, f_2) & \qquad \textit{OrDef}^{(C.3)} \\
f_1 \supset f_2 & \widehat{=} & \neg\, f_1 \vee f_2 & \qquad \textit{ImpDef}^{(C.4)} \\
f_1 \equiv f_2 & \widehat{=} & (f_1 \supset f_2) \wedge (f_2 \supset f_1) & \qquad \textit{EqvDef}^{(C.5)} \\
\exists\, v \bullet f & \widehat{=} & \neg\, \forall\, v \bullet \neg\, f & \qquad \textit{ExistsDef}^{(C.6)}
\end{array}
$$

Figure 2.8 presents a table of operator precedences and associativity.

[7]The laws are listed in Appendix C and the associated number indicates the law's position in that list.

Precedence	*Operator*	*Example*
1	$*$	$f^{**} \equiv (f^*)^*$
2	$\neg$	$\neg \neg x \equiv \neg (\neg x)$
2	$\bigcirc$	$\neg \bigcirc x \equiv \neg (\bigcirc x)$
3	$;$	$f_0 ; f_1 ; f_2 \equiv (f_0 ; f_1) ; f_2$
4	$\wedge$	$f_0 \wedge f_1 ; f_2 \wedge f_3 \equiv (f_0 \wedge (f_1 ; f_2)) \wedge f_3$
5	$\vee$	$f_0 \vee f_1 \vee f_2 \wedge f_3 \equiv (f_0 \vee f_1) \vee (f_2 \wedge f_3)$
6	$\supset$	$f_0 \supset f_1 \supset f_2 \equiv f_0 \supset (f_1 \supset f_2)$
7	$\equiv$	$(f_0 \equiv f_1 \equiv f_2) \equiv (f_0 \equiv (f_1 \equiv f_2))$

All derived prefix operators are $R - L$ *with precedence* 2.
Thus, $\neg \boxed{\text{i}} \neg \Diamond f \ ; \ g \wedge h \equiv (\neg (\boxed{\text{i}} (\neg (\Diamond f))) \ ; \ g) \wedge h$

Figure 2.8: ITL Operator Precedence and Associativity Table

$\bigcirc f \ \widehat{=} \ \mathsf{skip} \ ; \ f$ *StrongNextDef*(C.7)

$\mathsf{more} \ \widehat{=} \ \bigcirc \mathsf{true}$ *MoreDef*(C.8)

$\mathsf{empty} \ \widehat{=} \ \neg \, \mathsf{more}$ *EmptyDef*(C.9)

$\Diamond f \ \widehat{=} \ \mathsf{true} \ ; \ f$ *DiamondDef*(C.10)

$\Box f \ \widehat{=} \ \neg \Diamond \neg f$ *BoxDef*(C.11)

$\text{ⓦ} \, f \ \widehat{=} \ \neg \bigcirc \neg f$ *WeakNextDef*(C.12)

$\Diamond_{\text{i}} f \ \widehat{=} \ f \ ; \ \mathsf{true}$ *DiDef*(C.13)

$\boxed{\text{i}} f \ \widehat{=} \ \neg \Diamond_{\text{i}} \neg f$ *BiDef*(C.14)

$\Diamond_{\text{a}} f \ \widehat{=} \ \mathsf{true} \ ; \ f \ ; \ \mathsf{true}$ *DaDef*(C.15)

$\boxed{\text{a}} f \ \widehat{=} \ \neg \Diamond_{\text{a}} \neg f$ *BaDef*(C.16)

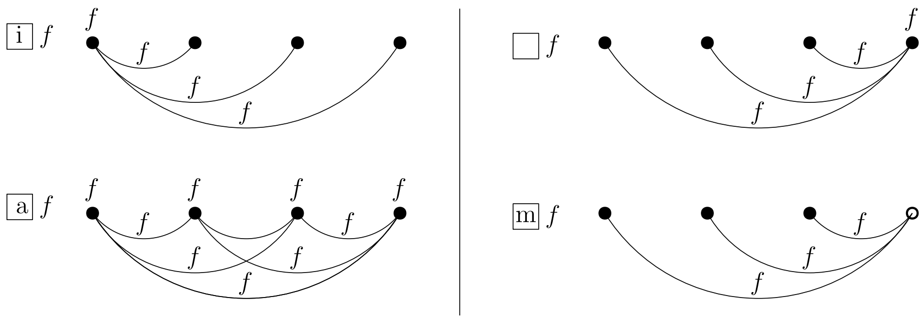
$\Diamond_{\text{m}} f \ \widehat{=} \ \neg \boxed{\text{m}} \neg f$ *DmDef*(C.71)

$\boxed{\text{m}} f \ \widehat{=} \ \Box(\mathsf{more} \supset f)$ *BmDef*(C.70)

Compared to LTL, the number of standard, derived operators in ITL is significantly greater. For example, the operators $\Box$ and $\Diamond$, defined over finite *suffix* intervals, have counterparts, $\boxed{\text{i}}$ and $\Diamond_{\text{i}}$, defined over finite initial (*prefix*) intervals. $\boxed{\text{i}} f$ specifies that f holds over *all* prefix intervals including the empty initial interval (i.e. the first state), and over the interval itself. The formula $\Diamond_{\text{i}} f$ means that f holds for *at least one* initial interval.

These operators can be combined. For example, $\Box \boxed{\text{i}} f$, which is equivalent to $\boxed{\text{i}} \Box f$, means that f holds over *all* subintervals. This case has its own derived operator, $\boxed{\text{a}} f$. In a similar way, $\Diamond_{\text{a}} f$ represents *at least one* subinterval and is equivalent to $\Diamond \Diamond_{\text{i}} f$ or $\Diamond_{\text{i}} \Diamond f$.

Whereas $\Box$ and $\Diamond$ include the empty suffix (i.e. the last state), there is a variation on each of these which covers all suffixes *except* the last state. These are $\boxed{\text{m}}$ and $\Diamond_{\text{m}}$ – the 'm' is intended to read "mostly".[8] The similarities and differences between various 'box' operators are illustrated in Figure 2.9.



(The empty disk, $\circ$, indicates that f does not necessarily hold at this empty interval)

Figure 2.9: ITL operators $\boxed{\text{i}}$, $\Box$, $\boxed{\text{a}}$, $\boxed{\text{m}}$

Because intervals are finite, it is possible to specify an empty interval (i.e., in ITL an interval with a single state) and a non-empty interval (at least two states). This is achieved using the formulae empty and more respectively. For example, the formula $\Box(\mathsf{more} \supset f)$ specifies that all non-empty suffixes satisfy f: i.e. f is not required to hold in the final state.

In Section 2.1.2 it was observed that a finite path semantics in LTL required a weak version of the *next* operator. Likewise, ITL has both *strong next* $\bigcirc$[9] and *weak next* $\textcircled{w}$ operators. Weak next is the dual of strong next, i.e. $\textcircled{w}\, f \equiv \neg \bigcirc \neg\, f$. The laws of ITL can be used to show that $\neg \bigcirc \neg\, f$ is equivalent to $\mathsf{empty} \vee (\mathsf{skip}\ ;\ f)$, which captures the semantics of weak next more directly.

Further temporal operators can be derived which simulate imperative programming language constructs such as $\mathsf{if}\ldots\mathsf{then}\ldots\mathsf{else}$ and $\mathsf{while}\ldots\mathsf{do}$ etc. These are useful within the context of Tempura – an executable subset of ITL – which is discussed in Section 2.3.2.

$$\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2 \ \widehat{=}\ (f_0 \wedge f_1) \vee (\neg\, f_0 \wedge f_2) \qquad \textit{IfThenElseDef}^{(C.17)}$$

$$\mathsf{if}\ f_0\ \mathsf{then}\ f_1 \ \widehat{=}\ \mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else\ empty} \qquad \textit{IfThenDef}^{(C.18)}$$

$$\mathsf{fin}\, f \ \widehat{=}\ \Box(\mathsf{empty} \supset f) \qquad \textit{FinDef}^{(C.19)}$$

$$\mathsf{halt}\, f \ \widehat{=}\ \Box(\mathsf{empty} \equiv f) \qquad \textit{HaltDef}^{(C.20)}$$

$$\mathsf{keep}\, f \ \widehat{=}\ \boxed{\text{a}}\,(\mathsf{skip} \supset f) \qquad \textit{KeepDef}^{(C.21)}$$

$$f^0 \ \widehat{=}\ \mathsf{empty} \qquad \textit{IterZeroDef}^{(C.23)}$$

$$f^{n+1} \ \widehat{=}\ f\ ;\ f^n, \quad [n \geq 0] \qquad \textit{IterDef}^{(C.24)}$$

[8] $\boxed{\text{m}}$ and $\Diamond_{\text{m}}$ are discussed in [Mos96a].

[9] Note that the $\bigcirc$ operator is overloaded in ITL and is defined for expressions and formulae.

$$\mathsf{for}\ n\ \mathsf{do}\ f \quad \widehat{=} \quad f^n \qquad ForDef^{(C.25)}$$
$$\mathsf{while}\ f_0\ \mathsf{do}\ f_1 \quad \widehat{=} \quad (f_0 \wedge f_1)^* \wedge \mathsf{fin}(\neg\, f_0) \qquad WhileDef^{(C.26)}$$
$$\mathsf{repeat}\ f_0\ \mathsf{until}\ f_1 \quad \widehat{=} \quad f_0\ ;\ \mathsf{while}(\neg\, f_1)\ \mathsf{do}\ f_0 \qquad RepeatDef^{(C.27)}$$

The formula $\mathsf{fin}\, f$[10] holds when f is true in the final state of the interval. f may hold for any suffix interval but it *must* hold in the final suffix interval. This formula contrasts with $\mathsf{halt}\, f$ which requires that f holds in the final state and that no other suffix interval satisfies f. Thus $\mathsf{halt}\, f$ uniquely determines an interval. For example, the formula $\mathsf{halt}(X = Y)$ holds over an interval in which $X = Y$ only in the final state. A discussion relating halt to the work of this thesis is presented in Section 3.8.6.

$\mathsf{keep}\, f$ requires that f holds over every two-state interval. Thus, for example, to specify that X must increase by one in each subsequent state: $\mathsf{keep}(\bigcirc X = X + 1)$.

Finally, there are standard, derived operators related to expressions:

$$A := e \quad \widehat{=} \quad (\bigcirc A) = e \qquad AssignDef^{(C.28)}$$
$$A \approx e \quad \widehat{=} \quad \Box(A = e) \qquad TemporalEqualityDef^{(C.29)}$$
$$A \leftarrow e \quad \widehat{=} \quad \mathsf{fin}\ A = e \qquad TemporalAssignDef^{(C.30)}$$
$$A\ \mathsf{gets}\ e \quad \widehat{=} \quad \mathsf{keep}(A \leftarrow e) \qquad GetsDef^{(C.31)}$$
$$\mathsf{stable}\ A \quad \widehat{=} \quad A\ \mathsf{gets}\ A \qquad StableDef^{(C.32)}$$
$$\mathsf{padded}\ A \quad \widehat{=} \quad (\mathsf{stable}\ A\ ;\ \mathsf{skip}) \vee \mathsf{empty} \qquad PaddedDef^{(C.33)}$$
$$A \lessdot\!\sim e \quad \widehat{=} \quad (A \leftarrow e) \wedge \mathsf{padded}\ A \qquad PaddedTemporalAssignDef^{(C.34)}$$
$$\mathsf{len}(n) \quad \widehat{=} \quad \mathsf{skip}^n \qquad LenDef^{(C.35)}$$

The assignment operators define values in *next*, *all*, and *final* states respectively; and gets and stable provide convenient shorthand notations. For example, $\mathsf{stable}\, A$ means that A's value does not change throughout the interval. The operator padded specifies stability up to but not including the final state. This is useful when used with the chop operator to specify stability up to but not including the shared state: for example, $\mathsf{padded}\, A\ ;\ \mathsf{stable}\, \neg\, A$. Padded temporal assignment $A \lessdot\!\sim e$ specifies that A remains unchanged throughout the interval until, possibly, the final state, at which time it gets the value e. The formula $\mathsf{len}(n)$ specifies that the length of the interval is n. Working with fixed-length intervals is a key aspect of the work in this thesis and properties of interval length are explored in more detail in Section 3.5.

[10] fin is also an overloaded operator in ITL, defined for expressions and formulae.

2.1.3.2 State formulae

Conventionally in ITL, the variable w is used to denote a state (non-temporal) formula. Such formulae do not include skip, ; , or *, or any operators derived from them. As such, a formula w is equivalent to $\mathsf{init}\ f$ where $\mathsf{init}\ f \mathrel{\widehat{=}} (f \land \mathsf{empty})\ ;\ \mathsf{true}$. It is used to express a property that must hold in a *single state* and, specifically, the first state of an interval. For example, the following equivalences capture properties that hold for state formulae:

$\vdash\ \boxdot\, w \equiv w$ *StateEqvBi*$^{(C.93)}$ [MOS]

$\vdash\ \diamondplus\, w \equiv w$ *DiState*$^{(C.112)}$ [MOS]

2.2 Intervals and runtime verification

Runtime verification of program behaviour requires checking that certain propositions occur in some temporal order. For example, in LTL a pattern describing such sequencing of events uses nested *until* operators [MP95]:

$$q_m\,\mathcal{U}\,q_{m-1}\ldots q_1\,\mathcal{U}\,q_0$$

The *until* operator is not associative and, in the absence of parentheses, is understood to associate to the right. This formula specifies a chain of intervals starting with a q_m interval.[11] $q_1\,\mathcal{U}\,q_0$ specifies that a finite q_1 interval holds at every position until q_0 holds. Note that q_0 may hold immediately in which case the q_1 interval contains no states.

Consider the requirement that q_1 must hold at the current position, and p must hold anywhere between the current position and the next position at which q_0 holds. This specifies a temporal ordering of these three propositions that is *not* captured by the LTL formula $q_1\,\mathcal{U}\,p\,\mathcal{U}\,q_0$ which neither guarantees p nor q_1. For example, if q_0 holds at the current state then the formula is satisfied. The formula needs to be strengthened:

$$q_1 \land (\neg\, q_0\,\mathcal{U}\,(p \land (\neg\, q_0\,\mathcal{U}\,q_0)))$$

This specifies the endpoints of a finite interval within which p must hold. Note that the formula is satisfied if q_1, p, and q_0 hold in the current position. It is also possible for p to hold either at the same state as q_1 (at the beginning) or at the same state as q_0 (at the end).

[11] A q interval is an interval in which q holds at every position.

The requirement can be expressed more straightforwardly in ITL:[12]

$$q_1 \wedge \mathsf{halt}(q_0) \wedge \Diamond p$$

The difference is that the semantics of LTL is defined with reference to a single point whereas ITL semantics interprets formulae with reference to two points – the start and end of an interval. In the ITL formula, q_1 must hold in the first state; $\mathsf{halt}(q_0)$ requires that the final state (and no other) satisfies q_0; and $\Diamond p$ requires p to hold at some point *within* the interval (defined by $\mathsf{halt}(q_0)$) including at the beginning or at the end. The formula is also satisfied by an empty (one state) interval in which all three propositions hold.

The approach to runtime monitoring advocated in this thesis requires partitioning the execution trace into a sequence of finite intervals and specifiying temporal formulae that the individual subintervals must satisfy. The interval semantics of ITL directly supports this approach to specification.

[Wol81] provides an example of a property that is not expressible in LTL, namely that "a property p has to be true in every even state of a sequence". For example, the formula $p \wedge \Box(p \Rightarrow \neg\, p) \wedge \Box(\neg\, p \Rightarrow p)$ does *not* express this property because it requires p not to hold in every odd state which is not what the specification says. [Wol81] proves more generally that for $i = km$ where $i \geq 0$, and $k \geq 2$, it is not possible in LTL to express the property "p is true in every state i". In ITL, which has the Kleene star operator, these examples can be written: $(p \wedge \mathsf{len}(i))^*$.

2.3 Direct execution of specifications

Runtime verification entails the dynamic analysis of an executing program against a formal specification. Direct execution of the specification represents a special case in which the program and the specification are the same. In this section two examples are described: MetateM and Tempura.

Both of these tools operate according to rules which define how to transition from one state in the execution trace to the next. In the case of MetateM there may be a choice of rules to apply at each step, admitting the possibility of future backtracking. In the case of Tempura, each step transition is uniquely determined. The following subsections provide a brief overview of each system.

[12] The propositions should be written with capital letters in ITL, but have been left in lower case here to aid comparison.

2.3.1 MetateM

METATEM [Fis06] transforms an LTL formula into separated normal form (SNF). This representation of a formula comprises a set of transition rules in which the current (and future) states are defined as a progression from the previous state.

Separated Normal Form

An LTL formula is translated into the following form: $\Box \bigwedge_{i}^{n} R_i$ where each R_i is one of the following rules (l_j, l_k, and l represent literals):

$$\begin{array}{lcll} start & \Rightarrow & \bigvee_{k=1}^{r} l_k & \text{(an } \textit{initial} \text{ rule)} \\ \bigwedge_{j=1}^{m} l_j & \Rightarrow & \bigcirc \bigvee_{k=1}^{r} l_k & \text{(a } \textit{step} \text{ rule)} \\ \bigwedge_{j=1}^{m} l_j & \Rightarrow & \Diamond l & \text{(a } \textit{sometime} \text{ rule)} \end{array}$$

[Fis11] (Chapter 4) shows how an arbitrary LTL formula can be translated into SNF. Each rule maps a formula relating to the 'current' state to a formula about the current and future states. Thus, each rule defines how the execution may progress from one state to the next. The rules whose antecedants are true are 'triggered' and values are produced to make the consequents true. (Rules whose antecedants are false are vacuously satisfied). Where the antecedants provide a selection of alternatives ($\bigvee_{k=1}^{r}$) then one of these may be selected at random. The *sometime* rule also involves a potential choice: to satisfy $\Diamond l$ immediately or to postpone. Meta rules in METATEM govern how these choices are made in order to optimse for efficiency. If a selected alternative leads to a future inconsistency, then backtracking is used to return to the last decision and select an alternative. If the backtracking returns to the initial state then the formula is deemed to be inconsistent because no model can be found to satisfy it.

Executing an LTL formula provides an alternative way to understand a specification. Its behaviour can be analysed using step by step animation and this can help to establish that the specification itself is fit for purpose. METATEM was inspired by Tempura, a tool that performs a similar analysis for ITL, and which is discussed below.

2.3.2 Tempura

Tempura [Mos86] is a tool for executing temporal logic specifications written using a subset of ITL. It can be used both for specifying required behaviour and also for validating the specification by animation. Tempura is also the basis of the AnaTempura runtime verification system which is described in Section 2.4.5.

Tempura statements correspond to certain ITL formulae that enable deterministic progression from one state to the next. Consequently, Tempura does not contain statements that require backtracking. This decision was taken deliberately to facilitate the "efficiency and simplicity of the interpreter" [Mos86]. Thus, the following compound statements are available: $f_1 \wedge f_2$ and $f_1 \supset f_2$, but neither $\neg\ f$ nor $f_1 \vee f_2$ are allowed. Furthermore, all variables need to be completely specified by the user in each state and termination conditions must also be provided.

Similar to METATEM, each Tempura statement is separated into a conjunction of the form $(current\ state) \wedge ⓦ\ (future\ states)$. The *weak next* operator is necessary because the interpreter may be in the final state, in which case the conjunction simply reduces to *current state*. [Mos86] (Chapter 8) discusses a possible implementation of Tempura and shows how each Tempura statement can be translated into the 'current and future states' formulation.

The following sequence of examples highlights the restrictions imposed by the deterministic requirements in Tempura. It also serves to illustrate the operation of the interpreter.

Consider the tempura statement: $\bigcirc I = I + 1$ (I's value increases by one in the next state). This statement can not be executed because I's value is not determined in the initial state, and a termination condition has not been provided. Figure 2.10 shows the translation of $\bigcirc I = I + 1$ into Tempura[13] and the corresponding failure when an attempt is made to run the program.

```
/* run */ define non_exec1() =
{
    exists I:
    {
       next I = I + 1
    }
}.
```

The Tempura program representing the ITL formula $\bigcirc I = I + 1$ generates the following output when run:

```
***Tempura error:  state #0 (pass #2) is not completely defined.
   Evaluating:  (next(I) = (I + ...))
   Undefined variables:
     Exists level  3: { I }
     Exists level  2: { }
     Global level  1: { }

   Fail
```

Figure 2.10: A non-executable Tempura program

[13] The existential quantification is used to declare the variable I.

To address the problem of incompletely defined variables, one could provide an initial value for I and write: $(I = 0) \wedge (\bigcirc I = I + 1)$. Thus Tempura would be able to determine the value of I in the *next* state. The updated program is shown in Figure 2.11. However, this is still insufficient information for Tempura to be able to execute the specification. The issue is that the specification would be true of *any* interval in which the first two states contained $I = 0$ and $I = 1$ respectively. Tempura needs to be able to determine the length of the interval over which to generate the required states. In the example, specifying an interval length of one (i.e. two states) is sufficient to enable the execution of the specification. This is shown in Figure 2.12.

```
/* run */ define non_exec1() =
{
    exists I:
    {
       I = 0 and next I = I + 1
    }
}.
```

The Tempura program representing the ITL formula $I = 0 \wedge \bigcirc I = I + 1$ generates the following output when run:

```
run non_exec2().

***Tempura error:  the interval length is undefined.
   Evaluating:  run non_exec2(?)
   Fail
```

Figure 2.11: A second, non-executable Tempura program

```
/* run */ define can_exec1() =
{
    exists I:
    {
       len 1 and I=0 and next I = I + 1
    }
}.
```

The Tempura program representing the ITL formula $\mathsf{len}\,1 \wedge I = 0 \wedge \bigcirc I = I + 1$ generates the following output when run:

```
run can_exec1().

Done!  Computation length:  1.  Total Passes:  2.
Total reductions:  18  (18 successful).  Maximum reduction depth:  7.
Time elapsed: 0.000020
```

Figure 2.12: An executable Tempura program

A final enhancement can be achieved by interacting with the user so that the initial value of I can be input. It is possible to print out the values of I in *all* states so that the animation can be inspected visually. In Figure 2.13 the addition of input and output statements to achieve this purpose is demonstrated.

```
/* run */ define can_exec2() =
{
    exists I:
    {
       len 1 and
       input(I) and
       next I = I + 1 and
       always output(I)
    }
}.
```

The Tempura program representing the ITL formula $\mathsf{len}\,1 \wedge \bigcirc I = I+1$ generates the following output when run with an initial value of 2 input for I.

```
run can_exec2().
State   0: % I=?
2.
State   0: I=2
State   1: I=3

Done!  Computation length:  1.  Total Passes:  2.
Total reductions:  27  (27 successful).  Maximum reduction depth:  7.
Time elapsed: 3.439013
```

Figure 2.13: An executable Tempura program with I/O

The final example in this short discussion will demonstrate how contradictory behaviour can be identified and reported by Tempura. The running example will be adapted by introducing a constraint on the *final* value of I: $\mathsf{len}\,1 \wedge \bigcirc I = I + 1 \wedge \mathsf{fin}(I = 2)$. This restricts the initial value of I to be 1. Tempura cannot run the specification “backwards” in time to deduce this. However, if the value in the first state is not equal to 1 then a contradiction in the second state will be apparent. Figure 2.14 illustrates this behaviour.

```
/* run */ define can_exec3() =
{
    exists I:
    {
       len 1 and
       input(I) and
       next I = I + 1 and
       always output(I) and
       fin (I = 2)
    }
}.
```

The Tempura program representing the ITL formula $\mathsf{len}\,1 \wedge \bigcirc I = I + 1 \wedge \mathsf{fin}\, I = 2$ generates the following output when run with a value of 2 input for I.

```
run can_exec3().
State   0: % I=?
2.
State   0: I=2
State   1: I=3

***Tempura error:  attempt to overwrite variable.
   Evaluating:  (I = 2)

   The variable has currently the value 3.

   Fail
```

However, when run with a value of 1 input for I no contradiction occurs.

```
run can_exec3().
State   0: % I=?
1.
State   0: I=1
State   1: I=2

Done!  Computation length:  1.  Total Passes:  2.
Total reductions:  31  (31 successful).  Maximum reduction depth:  7.
Time elapsed: 3.429490
```

Figure 2.14: An executable Tempura program with I/O and a final constraint

2.4 Architectures

Correctness properties are specified using a verification logic: for example, an LTL formula φ. To enable a program to be checked against φ the formula must be transformed into an executable *monitor* that can be run alongside the program.

Definition 2.4 Monitor *A monitor is a process that analyses an execution trace and tries to determine whether or not it satisfies a specification.*

A runtime monitor continuously analyses an evolving execution trace of a running program. The process of adapting a program so that it emits significant event and/or state data to a monitor as these events occur is called *instrumentation*.

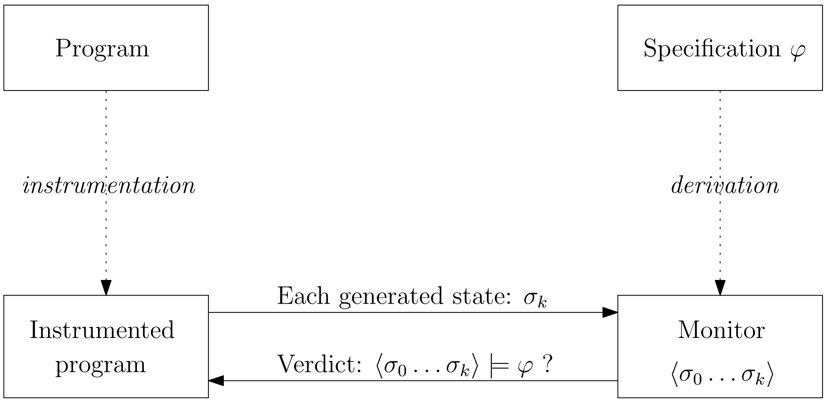
Definition 2.5 Instrumentation *The adaptation of a computer program to insert code that captures and transmits to a runtime monitor any event relevent to the runtime verification.*

[RH16] observe that the two most common instrumentation techniques are capturing method calls and using variable updates. The first approach is used in a range of Java-based runtime verification systems in which events can be triggered upon entering or leaving specified methods. A popular technique uses AspectJ [Asp17], an extension to Java that enables 'aspects' of a system – such as those required for instrumentation – to be separated from the main program logic. The nature of such aspects is that they are interwoven with the main program and the insertion of specialised code to deal with these is performed automatically. AspectJ is a language in which certain events within a program execution such as method calls, so-called 'pointcuts', can be specified together with code to be run at each of these pointcuts. This method is used by JavaMOP [JMLR12]

The second approach requires assertion points to be placed directly into the program at the points when a change is made to any of the monitored state variables. This is the method used by AnaTempura [CZCM96, ZZC05] (see Section 2.4.5) in which the assertions comprise formatted print statement which transmits event data to the standard output channel which, in turn, is read by the monitor. The use of bespoke assertion points is also required by TraceContract [Hav19, BH11] and ITL-Monitor, the subject of this thesis. Both TraceContract and ITL-Monitor are constructed as DSLs within Scala (cf. 2.4.1) and, as such, become part of the host program under test via an API. An example program and specification are presented in Section 6.2 which form the subject of a comparative analysis of each of these runtime verification tools.

Figure 2.15 provides a high-level view of the relationship between a program and a monitor.

The figure shows three relationships:



The program to be verified is instrumented so that it can generate states when significant events occur. A monitor is derived from the specification φ. Each new state σ_k is passed to the monitor which maintains the trace history $\langle \sigma_0 \dots \sigma_k \rangle$, and, upon receipt of each new state, checks $\langle \sigma_0 \dots \sigma_k \rangle \models \varphi$, and returns a verdict.

Figure 2.15: Runtime verification

1. Between the program and the instrumented program. The program is adapted to include code that emits significant events (states) to the monitor when they occur. (see Definition 2.5)

2. Between the instrumented program and the monitor. Significant events that create a new state for analysis are sent to the monitor which, in turn, delivers a verdict.

3. Between the specification and the monitor. This can be achieved using automatic translation (compilation) from a specification language into a programming language. Alternatively, the specification and the monitor may coincide either by using a specification language that is directly executable, or by using an API in the executable language that encodes the specification.

The following arrangements of programs and monitors describe standard architectural patterns [Leu12].[14]

Outline monitoring: the monitor is separate from the program under test. This architecture is based upon a loose coupling between the components and relies upon data being communicated using a channel. AnaTempura (2.4.5) is an example of such an outline monitor system. It is possible using such an arrangement that an outline monitor can use a separate processor and not affect the running performance of the program itself.

[14] These are not mutually exclusive: ITL-Monitor is both inline and online.

Inline monitoring: the monitor is part of the program itself and shares its computational resources. This facilitates efficient cooperation between the program and the monitor. Although this can increase the potential coupling between the components, mitigation can be achieved by design. For example, ITL-Monitor is implemented using an *actor* model in which messages are passed between autonomous processes which maintain their own encapsulated state.

Offline monitoring: the analysis takes place after the program has run. This requires that a log of the program run is constructed while the program executes, and is stored for subsequent analysis. This can be achieved by any runtime verification system since it is possible to 'execute' a log file by traversal. There are verification systems that are designed to be run offline. One specific example is LogScope [BHRG09] which used offline monitoring because the specific application for which it was designed (NASA's Mars Science Laboratory, a planetary rover) was unable to provide runtime data in a reliable order. Another (unpublished) example is ITL-Tracer [Jan10], a Java monitor for analysing completed program traces with respect to an ITL specification. Offline monitoring provides a post hoc analysis of a program's behaviour and can utilise algorithms that take considerably more time than would be acceptable for an interactive diagnosis.

Online monitoring: the analysis takes place while the program is running. [LS09] points out that making the monitor part of the system itself allows the monitor to analyse faults and modify subsequent behaviour. In particular, online monitoring facilitates *runtime reflection* in which fault detection, identification and recovery can take place. ITL-Monitor has been implemented as a monitoring process that runs concurrently with the program, and which can raise user-defined exceptions when the verification fails. This mechanism can be used by the main program to react at runtime, for example by defining recovery behaviour within *catch* clauses.

2.4.1 Domain specific languages

Current research in runtime verification is increasingly considering the use of *Domain Specific Languages* (DSLs, see Pattern 21 in [BL13]) for monitor construction. Indeed, DSLs, particularly in Scala [OSV16, Sca17], have been the subject of active research in recent years including, e.g. [Hav11, Hav13, Hav14, YAH+16]. Björner and Havelund argue in [BH14] that specification, verification and programming may be converging with such contemporary programming language developments. A classification of DSLs with Scala is given in [AHKY15].

DSLs can be categorised into external and internal, and the latter can be implemented using either a deep embedding or a shallow embedding. External DSLs are separate languages whose syntax is not constrained by any host programming language. A specification written using an external DSL can either be compiled using a bespoke compiler into code that can be executed by the host language, or it may be parsed into an internal data structure and then interpreted within a progam. For example, in the host language Scala, the Scala combinator parser library could be used.[15] JavaMOP [JMLR12] and RuleR [BHRG09] are examples of runtime verification systems developed as external DSLs. Both of these languages have compilers that translate specifications into AspectJ aspects (see Section 2.4.2) which are used to instrument Java programs for monitoring.

Internal DSLs extend the host language itself and thus benefit from total integration with its constructs. A deeply embedded DSL comprises a language, represented as an abstract syntax tree, which is interpreted from within the program. Within this thesis, the ITL library for use with ITL-Monitor, namely `ITL.scala`, is a deeply embedded DSL. ITL expressions and formulae are instances of abstract syntax trees which can be transformed to perform certain optimisations, and interpreted for evaluation. A shallow embedded DSL uses the host language's features predominantly for its representation. This is facilitated by programming languages whose features support this approach. Such features include partial functions, generic types, pattern matching, and higher order functions.

TraceContract [BH11] is another internal DSL written in Scala for runtime verification. It supports specification using state machines and temporal logic. TraceContract is in current use and is actively maintained [Hav19]. In Chapter 6 TraceContract is selected as one of the contemporary runtime verification tools used for comparison with ITL-Monitor.

2.4.2 Aspect oriented approaches

Aspect-oriented programming (AOP) is designed specifically to enable the separation of concerns facilitated by using a compiler to interweave 'cross-cutting' monitoring code into an application. This approach facilitates monitoring by triggering verification activities at certain programmer-defined pointcuts specified using AspectJ (cf. page 31), for example, and a runtime monitoring system has to integrate with the AspectJ API. The syntax does provide significant flexibility for capturing classes of events. For example, a join point may be attached to the execution of an instance method associated with any object of a specific type. Events may be triggered, for example, before, after, or around such method invocations. Such instrumentation separates the concerns of the monitor code and the system under scrutiny.

Monitoring-oriented programming (MOP) [CR07, CR03] is a framework supporting the development and analysis of software systems that permits a variety of formal languages to be

[15] This is maintained as a community project at [Sca].

used in the specification of monitors for runtime verification. Eschewing the idea that a single formalism is appropriate, MOP supports a variety of user-defined plug-ins to enable particular properties to be expressed using a formalism that is best suited for the task. These plug-ins are monitor synthesizers that translate formulae into runtime monitors. An implementation of MOP for the Java language, JavaMOP [JMLR12, Jav17], has been developed in which monitoring code is woven into the program using AspectJ. The architecture permits user-defined Java code to be included ('user-defined actions') for execution when monitors report either success or failure. Thus, code can be inserted to perform dynamic recovery when specific errors are caught.

JavaMOP is a parametric monitoring framework. This means that formulae may contain parameters that become bound to actual object instances in the program. When a property must be monitored for a class of objects in a program, then every instance of the class has an associated bespoke monitor instance. To facilitate such monitoring, the input trace must be sliced such that each slice contains events only specific to a particular monitor instance. Efficient indexing of monitor instances has been shown to be computationally very efficient [Jin12].

RV-Monitor [DGH+16] is an evolution of the JavaMOP framework and is used for enforcing safety and security policies at runtime. A version of RV-Monitor for Android is described in [DFM+15]. RV-Monitor supports both manual and automated instrumentation, the latter via a tool such as AspectJ.

Related current research has focused on AOP and AspectJ in particular and in [JZR+16] the authors discuss the limitations of AspectJ's join point mechanism and propose a domain-specific aspect language, DiSL, which they demonstrate leads to extended code coverage. The authors provide a compiler for translating existing AspectJ *aspects* into DiSL.

2.4.3 Rule-based approaches

Rule-based runtime verification approaches comprise systems in which a specification is written as a set of rules, each of which is separated into its antecedants - sets of facts about the current state – and a set of consequents – future time formulae that must hold in (current and) future states. MetateM utilised this method splitting LTL formulae into present and future formulae using separated normal form (2.3.1).

MetateM influenced a range of rule-based, runtime verification logics developed significantly by Barringer, Havelund, Rydeheard et al. In [BGHS04a, BGHS04b] the authors introduced Eagle, a temporal fixed-point logic defined over finite traces. The full syntax and semantics of Eagle is presented in [BGHS04a]. Eagle was designed as a general purpose, rule-based temporal logic for runtime verfiication which included support for interval logic, LTL with future and past time, and regular expressions. In EAGLE temporal operators are expressed in terms of minimal and maximal fixpoints:

$$\begin{aligned}
&\underline{\mathsf{max}}\,\mathsf{Always}(\underline{\mathsf{Form}}\ F) &&= F \wedge \bigcirc\mathsf{Always}(F)\\
&\underline{\mathsf{min}}\,\mathsf{Eventually}(\underline{\mathsf{Form}}\ F) &&= F \vee \bigcirc\mathsf{Eventually}(F)\\
&\underline{\mathsf{min}}\,\mathsf{Until}(\underline{\mathsf{Form}}\ F_1, \underline{\mathsf{Form}}\ F_2) &&= F_2 \vee (F_1 \wedge \bigcirc\mathsf{Until}(F_1, F2))
\end{aligned}$$

where $\underline{\mathsf{max}}$ and $\underline{\mathsf{min}}$ denote maximum and minimum fixpoints respectively; $\underline{\mathsf{Form}}$ is the type of formulae; and $\bigcirc$ is the strong-next operator from LTL.[16] In EAGLE the semantics of formulae are defined over finite intervals, σ, whose states are indexed from 1 to $|\sigma|$. The indexes 0 and $|\sigma| + 1$ define the *boundary* of the interval. Rules defined using max evaluate to True at the interval boundaries, thus the following rule [BB08] holds only when the formula is observed at one of the boundaries:

$$\underline{\mathsf{max}}\,\mathsf{Limit}() = \mathsf{False}$$

Safety properties (e.g. $\Box p$) use a maximal fixpoint interpretation and, as such, are considered to be satisfied throughout the trace once the end is reached. If this were not the case then a contradiction would have been discovered earlier. Alternatively liveness properties (e.g. $\Diamond p$) use a minimal fixpoint interpretation. This means that if the end of the trace is reached then the property is not satisfied, otherwise it would have been discharged earlier. This interpretation of liveness is based upon finite path semantics (cf. page 15).

EAGLE includes a non-deterministic, sequential composition operator. The formula F_1 ; F_2 is satisfied by a (finite) interval σ provided that the interval can be split into a prefix σ^p and suffix σ^s such that $\sigma^p(|\,\sigma^p\,|) = \sigma^s(1)$, i.e. the final state of σ^p coincides with the first state of σ^s, and F_1 holds on σ^p (observed from some position i), and F_2 holds on σ^s. Importantly, future operators within F_1 are limited to the scope of σ^p and, conversely, past-time operators within F_2 are limited to the scope of σ^s. The full semantics of EAGLE logic is presented in [DH05]. The following excerpt defines the behaviour of sequential composition:[17]

$$\sigma, i \models F_1 \ ;\ F_2 \text{ iff exists } j \text{ s.t. } i < j \leq |\sigma| + 1 \text{ and } \sigma_{1..j-1}, i \models F_1 \text{ and } \sigma_{j-1..|\sigma|}, 1 \models F_2$$

EAGLE also supports a related concatenation construct: $F_1 \cdot F_2$. In this case the prefix and suffix intervals do not overlap: thus $\sigma = \sigma^p\sigma^s$ (where juxtaposition here represents sequence concatenation).

[16] A brief discussion of fixpoint logics is found in Section 8.4 of [Eme90] and Section 2.8.4 of [Fis11].

[17] In EAGLE semantics, the first state is indicated by 1, not zero.

$$\sigma, i \models F_1 \cdot F_2 \text{ iff exists } j \text{ s.t. } i \leq j \leq |\sigma| + 1 \text{ and } \sigma_{1..j-1}, i \models F_1 \text{ and } \sigma_{j..|\sigma|}, 1 \models F_2$$

In [BB08] the authors show that sequential composition can be represented using concatenation, and vice versa, and therefore that each is equally expressive. However, it is observed that the non-determinism in these operators can be computationally expensive searching for a suitable set of cut points that satisfy a formula. Deterministic variants of the concatenation and sequential composition operators are introduced. The authors show that these deterministic variants do not add any new expressive power to Eagle but, by identifying them as bespoke operators, it is possible to provide more efficient implementations of runtime monitors that use them based upon their deterministic semantics. Specifically, eight variations are defined: these are the left-minimal, left-maximal, right-minimal, and right-maximal operators for both sequential composition and concatenation. $\lfloor F_1 \rfloor \circ F_2$, $\lceil F_1 \rceil \circ F_2$, $F_1 \circ \lfloor F_2 \rfloor$, and $F_1 \circ \lceil F_2 \rceil$, where $\circ$ can be either $\cdot$ or $;$. The semantics for $\lfloor F_1 \rfloor \; ; \; F_2$ is given below:

$$\begin{aligned}\sigma, i \models \lfloor F_1 \rfloor \; ; \; F_2 \text{ iff exists } j \text{ s.t. } i < j \leq |\sigma| + 1 \; & \text{ and } \sigma_1 \ldots \sigma_{j-1}, i \models F_1 \\ & \text{ and } \sigma_{j-1} \ldots \sigma_{|\sigma|}, 1 \models F_2 \\ \text{and not exists } k \text{ s.t. } i \leq k < j-1 \; & \text{ and } \sigma_1 \ldots \sigma_k, i \models F_1\end{aligned}$$

Eagle itself is not specific to any particular programming language. This means that it has no means of being instrumented (see Definition 2.5). Jeagle [DH05] is a runtime verification tool that extends Eagle and is written for the Java programming language. Jeagle incorporates a compiler that parses specifications written in a specification file and emits automatic instrumentation code that can be processed by AspectJ (cf. 2.4.2).

In [BRH07] the authors, while acknowledging the richness of Eagle and its appropriateness for specifiying complex temporal behaviours, also discuss how the non-deterministic concatenation operator leads to a high computational cost. The authors note that there are some subsets of Eagle that could be executed efficiently for runtime monitoring – in particular, the LTL subset of the language. The paper signals a change of research direction towards a lower level, rule-based logic for runtime verification.

In [BHRG09] RuleR, an online trace analysis tool, is described. In RuleR a set of rules is defined, each of which has an antecedant and a consequent. Both of these must be state expressions (i.e. no temporal formulae). The rules of the core system do not remain active between events – they are so called one-shot rules. As each event is passed to the system, the antecedants of the active rules are tested thereby conducting a breadth-first search of the possible traces that satisfy the rules. The only rules that become active in the subsequent state are the consequents of those whose antecedants were triggered in the current state. The process continues until either no rule applies, or the trace is terminated. On top of these single-state persistence (**step**) rules, RuleR has also built two others: **state** persistence and **always** persistence. The former defines rules that remain active until they are activated successfully, and the latter rules that remain activated throughout the verification. [BHRG09] observes that the **state** rules were used more often by users writing RuleR specifications, thus indicating a preference for the state machine approach. It is interesting to note that these three categories of rules are also reflected in the behaviours of the state functions provided in the subsequent TraceContract runtime verification tool (Section 2.4.4).

Research into RuleR, whose rule-based system is based on MetateM, led to consideration of an alternative, yet established, algorithm used extensively in AI rule-based systems: the Rete algorithm [For82]. [Hav15] discusses the adapation of the Rete algorithm for rule-based, runtime verification and its realisation as LogFire. The paper summarises the performances of RuleR and LogFire against each other and five other tools over seven experiments designed to stress the systems in terms of memory requirements and monitor indexing. The survey concludes that for low memory experiments RuleR performed better than LogFire, but that the situation was reversed for high memory experiments. Interestingly, (unoptimised) TraceContract performed comparably to RuleR. However, the MOP system outperformed all of the competition by an order of magnitude – the authors suggest this is due to MOP's indexing system being significantly faster than algorithms such as Rete for runtime verification.

2.4.4 TraceContract

TraceContract [BH11, Hav19] is a runtime verification tool implemented as a shallow embedded DSL (cf. 2.4.1) in Scala. It provides an API that supports specification using state machines and linear temporal logic. It also provides a persistent 'database' in which facts can be stored for future reference. Such a database provides support for temporal formulae involving previous events. However, the persistent nature of the facts makes it necessary for addition and deletion to be performed explicitly as the runtime verification proceeds. Alternatively, given that the monitors are written as standard Scala code, it is possible to encode previous events using the language itself rather than using the built-in database.

Fundamental to TraceContract is the class `Monitor[Event]` – which can be instantiated to create a monitor capable of processing a list of events `List[Event]`. `Event` is a generic type parameter which is substituted by the actual type of events to be monitored. Each monitor maintains a private (possibly empty) list of sub-monitors forming a hierarchical composition in which the sub-monitors at each level are effectively conjoined. As each new event is processed by a monitor it is, in turn, recursively passed on to each of its sub-monitors. This approach is utilised in the example given in Section 6.2.5.3.

A number of `Formula`e can be defined within each monitor that specify the required behaviour.

Formulae are separated into two types: one representing state logic, and another representing future time temporal logic. The two types can be used together within the same monitor if required. At the heart of the state logic formulae is the `Block` type:

```
type Block = PartialFunction[Event, Formula]
```

which is used in the definition of the state functions, e.g.

```
def state(block: Block): Formula
```

The use of `PartialFunction` allows Scala's pattern matching notation to be used to capture specific `Event` instances as shown in the following example, again taken from 6.2.5.3:

```
def S0: Formula = state {
                    case Event(true ,false,false) => S4
                    case Event(false,false,false) => S0
                    case _ => error
                  }
```

When this state formula (`S0`) is active then a matched event evolves the monitor into a subsequent state formula: for either of the valid events shown this is either `S4` or `S0`. The default case '_' catches any invalid event and evolves the monitor to an error state formula. This illustrates how state transitions can be defined based upon events. Once the formula has evolved to its new state formula then it is this that is matched against the next incoming event. A range of state functions is defined representing different types of state evolution. In each case, the parameter is a `Block` and the semantics of each function determines how the evolved formula is determined based upon whether or not the incoming event matches one of the cases. These are:

`state(block: Block): Formula` This formula remains in the monitor's list of active formula until a matching event occurs. Then the formula evolves according to `block`.

`step(block: Block): Formula` If the incoming event is matching then the formula evolves according to `block`. Otherwise the formula evolves to the special formula `True` representing success.

`hot(block: Block): Formula` This is the same as `state` with the exception that it is an error to be in a 'hot' state at the end of the trace. This is used to represent liveness properties with respect to finite path semantics (cf. page 15).

`strong(block: Block): Formula` A matching event must occur in the next state. Then the formula evolves according to `block`. If the match does not occur in the next state, or if there is no next state, then the formula evolves to `False` indicating failure.

`weak(block: Block): Formula` This is the same as `strong` except that no error occurs if there is no next step.

`always(block: Block): Formula` This is different from the other state functions in that it is always active. Whenever a matching event occurs then `always(block)` remains in the list of active formulae, and the formula that is produced by the match is also added to the list.

TraceContract also provides functions representing future time LTL formulae. These are

`matches(p: PartialFunction[Event, Boolean]): Formula` This equals `True` if and only if the current event satisfies `p`; otherwise `False`.

`not(f: Formula): Formula` This negates `f`.

`globally(f: Formula): Formula` This formula must hold for the current event, and all future events. At the end of the trace `globally(f)` equals `True`.

`eventually(f: Formula): Formula` This formula must hold either for the current event, of for some future event. At the end of the trace `eventually(f)` equals `False`.

`never(f: Formula): Formula` This formula must be false for the current event and for all future events. At the end of the trace `never(f)` equals `True`.

`strongnext(f: Formula): Formula` This formula must be true for the next event and there must be a next event. In this case `strongnext(f)` equals `True`; otherwise `False`. At the end of the trace `strongnext(f)` equals `False`.

`weaknext(f: Formula): Formula` This formula must be true for the next event if there is a next event. In this case, or if there is no next event, then `weaknext(f)` equals `True`; otherwise `False`. At the end of the trace `weaknext(f)` equals `True`.

All formulae, i.e. state formulae and future time LTL formulae, can be combined with the infix methods `and`, `or`, `implies`, `until`, and `unless`. An example of the use of LTL formulae is shown in Listing 6.2 (page 130).

TraceContract is designed to analyse an event trace. Monitoring can either take place offline, by processing the complete trace of a previously-run program, or it can be performed online by passing the events to the monitor as they are encountered. Instrumentation is not automated and the method, `verify(event: Event)`, must be called explicitly from within the program under test.

TraceContract is similar to ITL-Monitor in a number of ways: both are implemented as internal DSLs in Scala; both require manual instrumentation; and both are designed as experimental runtime verification tools for their respective formalisms. Both systems make obvious efficiency gains by analysing their internal abstract syntax trees to simplify evaluations, although neither has been subject to extensive optimisation. Both systems support monitor composition. This makes TraceContract an excellent choice for direct comparison (see Chapter 6).

However, there are also some fundamental differences. TraceContract does not have the facility to report judgements back to the program under test. Rather, they are output onto a transcript in a similar way to AnaTempura (see below Section 2.4.5). ITL-Monitor can report judgements, or throw exceptions, in response to each new event, thus providing the option to react at runtime. TraceContract is based upon a hybrid formalism that incorporates state machines and LTL, whereas ITL-Monitor is based upon ITL. Notwithstanding timed formulae, TraceContract monitors do not exploit multithreading, whereas every monitor in ITL-Monitor is an Akka actor. The Akka scheduler can allocate actors to run in parallel on separate system cores.

2.4.5 AnaTempura

AnaTempura [CZCM96, ZZC05] is an established runtime verification system that uses Tempura as its basis. The system consists of a Tempura specification, a Tempura interpreter, and a program to be verified. AnaTempura executes the specification and the program in parallel checking that the program satisfies the specification at each state.

The program under scrutiny is instrumented by *assertion points* embedded within the code. These transmit states to AnaTempura whenever a monitored state variable is modified. AnaTempura computes the expected trace and compares it with the actual trace supplied by the program under test, reporting continually whether these traces are in agreement. AnaTempura permits a user to view the result of this monitoring process in real time and to intervene should a problem arise. Thus AnaTempura supports a "stop and repair" model of runtime verification but does not implement a "react at runtime" [LS09] pattern and therefore does not support automatic fault detection and recovery.

The monitoring consists of three main components

1. The program being analysed. The program contains assertion points that have been introduced strategically into the code to report any changes to the state that refers to variables used in the specification. These assertion points may be designed into the code from the outset or added retrospectively requiring a re-build of the software component.

2. The Tempura interpreter. This 'runs' a Tempura specification which is provided in a separate file. Tempura specifications are written using an executable subset of ITL and thus generate a deterministic sequence of states. This is the *expected* trace.

3. The runtime monitor. This compares the incoming states from the program with the

corresponding states from Tempura to ensure that they are in agreement. As soon as a discrepancy is discovered this is reported via an output console.

AnaTempura contains further features that are useful for developing specifications. In particular there is an *animator* that permits the user to visualise the specification as it is executed.

In AnaTempura the communication between the program under scrutiny and the monitor is achieved by the insertion of *assertion points* into the program. The coupling between the monitor and the program is therefore loose and unidirectional: snapshots of the state are sent to the monitor using a "fire and forget" strategy. Specifically, this does not permit the synchronisation of the monitor and the program, and it does not permit information to be sent back from the monitor to the program. This asynchronous communication does not require the program to wait at any time for a response from the monitor and thus the monitoring places only a very small performance penalty on the program through the execution of assertion points. It is possible that the expected trace cannot be generated as fast as the actual trace is being generated. Note that if the monitor and the program are running on the same processor then the runtime performance of both is affected.

2.5 Summary

Runtime verification has been introduced as a complementary method to model checking. Two temporal logics were discussed: LTL, used widely in model checking, and the basis for many runtime verification systems; and ITL, a logic that is not widely used for runtime verification possibly due to the computational complexity introduced by its non-deterministic sequential composition operator.

The chapter discussed the principal runtime verification architectures that have emerged from this relatively recent computer science discipline. Particular emphasis was put upon two tools, TraceContract and AnaTempura. Both of these have been selected as suitable candidates for comparison with the current work: TraceContract because it is also implemented as an internal DSL in Scala and supports LTL; AnaTempura becuase it is the only established tool that uses ITL. However, the latter uses a deterministic subset of ITL called Tempura. This thesis proposes a technique whereby ITL with fewer restrictions can be used for runtime verification. The theory to support this is introduced in the next chapter.

Chapter 3

Development of the first occurrence operator

Throughout this chapter, and the rest of the thesis, reference is made to a substantial collection of laws of ITL. These are available as [CMS19], a document in which the syntax and semantics of ITL have been encoded using Isabelle/HOL and in which every law has a mechanically checked proof. The laws in that paper include all of Moszkowski's unpublished work in [Mos14a]; Moszkowski's investigation into time reversal [Mos14b]; work by Cau on Interval Temporal Algebra [Cau08]; as well as laws provided by the author in the process of developing this work.

The work in this thesis required a thorough investigation into fixed-length intervals, strict initial intervals, and the first occurrence operator. The latter is a key part of the work and an extensive collection of laws relating first occurrence to other ITL operators has been added to ITL. All of these laws and their proofs have been checked automatically within the Isabelle/HOL framework and appear in Chapter 7 of [CMS19].

3.1 Introduction

ITL [Mos82, Mos83, CZCM96] is a mature mathematical framework for system specification. Its syntax and semantics were introduced in Section 2.1.3 along with a number of derived operators. Fundamental to ITL are the *chop* and *chopstar* operators. Recall the semantics of chop (Section 2.1.3):

$$\mathcal{F}[\![f_1 \ ; \ f_2]\!](\sigma) = \mathsf{tt} \text{ iff } (\text{exists}\, k, \text{ s.t. } \mathcal{F}[\![f_1]\!](\sigma_0 \ldots \sigma_k) = \mathsf{tt} \text{ and } \mathcal{F}[\![f_2]\!](\sigma_k \ldots \sigma_{|\sigma|}) = \mathsf{tt})$$

Note that the two subintervals share a common state at k which is referred to as the fusion point. The choice of k is non-deterministic when there is more than one way to satisfy $f\ ;\ g$ for a given interval. Figure 3.1 shows how the formula $\Box P\ ;\ \Diamond Q$ is satisfied by the interval $\langle (P, \neg\ Q), (P, \neg\ Q), (P, Q), (P, \neg\ Q), (P, \neg\ Q)\rangle$ in three ways – each representing a different fusion point.

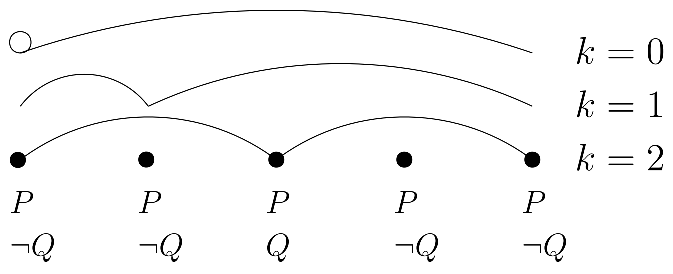


Figure 3.1: Different ways to achieve $\Box P\ ;\ \Diamond Q$ for a given interval.

If one of the subformulae were strenghtened then the set of fusion points may be reduced. For example, consider the formula $\Box(P \wedge \neg\ Q)\ ;\ \Diamond Q$ over the same interval. This reduces the fusion points to $k \in \{0, 1\}$. The set of fusion points can be reduced to one by strengthening the first subformula even further: $((\Box P) \wedge \mathsf{skip})\ ;\ \Diamond Q$. In this case the first subinterval must have unit length leaving the only possible fusion point as $k = 1$.

When using ITL for runtime verification, determining whether $\sigma \models f_0\ ;\ f_1$ for an interval σ, requires a search for an appropriate fusion point. If neither f_0 nor f_1 themselves contain chop or chopstar operators then every state may have to be considered as a potential fusion point. If a formula is of the form $f_0\ ;\ \ldots\ ;\ f_k$, for $k > 0$, where each f_i does not contain chop or chopstar operators, then the maximum number of potential fusion points is given by $\binom{|\sigma|+1}{k} = \frac{(|\sigma|+1)!}{(|\sigma|+1-k)!\ k!}$. This combinatorial complexity is prohibitively expensive in the context of runtime verification.

3.2 Timing analysis

To compare algorithms for verifying an ITL formula f over a finite interval σ the below function T is introduced. It estimates the worst-case time taken to perform a verification based upon the number of simple tests that need to be made. The assumption is made that both state lookup and establishing interval length take constant time. The primary focus is on the analysis of chop and chopstar and for this reason only a subset of ITL functions has been considered. In the definition n represents the length of the interval (i.e. the number of states minus one).

$$
\begin{aligned}
T(n,\mathsf{true}) &= 0\\
T(n,\mathsf{empty}) &= 1\\
T(n,\mathsf{more}) &= 1\\
T(n,\mathsf{skip}) &= 1\\
T(n,\mathsf{len}(k)) &= 1\\
T(n,Q) &= 1 \quad (Q \text{ represents any propositional variable})\\
T(n,\mathsf{fin}(Q)) &= 1\\
T(n,\neg f) &= 1 + T(n,f)\\
T(n,f_1 \wedge f_2) &= 1 + T(n,f_1) + T(n,f_2)\\
T(n,f_1 \; ; \; f_2) &= \sum_{k=0}^{n} (T(k,f_1) + T(k,f_2))\\
T(n,f^*) &= \begin{cases} 0, & \text{if } n = 0\\ \left(\sum_{k=1}^{n} T(k,f)\right) + \left(\sum_{k=0}^{n-1} T(k,f^*)\right), & \text{if } n > 0 \end{cases}
\end{aligned}
$$

An algorithm for establishing $\sigma \models f_1 \; ; \; f_2$ considers each state $\sigma_0, \sigma_1, \ldots$ in turn as a potential fusion point. Note that whenever $\sigma_0 \ldots \sigma_k \not\models f_1$ then the corresponding test $\sigma_k \ldots \sigma_{|\sigma|} \models f_2$ is not required. Furthermore, if $\sigma_0 \ldots \sigma_k \models f_1$ and $\sigma_k \ldots \sigma_{|\sigma|} \models f_2$ then no further fusion points need to be considered. The definition of $T(n, f_1 \; ; \; f_2)$ assumes the worst case in which every prefix interval satisfies f_1 and no suffix interval satisfies f_2 thus requiring every state to be examined as a potential fusion point. Thus f_1 is checked over each of the subintervals σ_0, $\sigma_{0..1}$, $\ldots$, $\sigma_{0..n}$ and, for each of these, f_2 must be checked over subintervals $\sigma_{0..n}$, $\sigma_{1..n}$, $\ldots$, σ_n respectively. $T(n, f_1 \; ; \; f_2)$ is therefore given by $\sum_{k=0}^{n} T(k,f_1) + \sum_{k=0}^{n} T(n-k,f_2) = \sum_{k=0}^{n} (T(k,f_1) + T(k,f_2))$.

The summation for $T(n, f^*)$ is defined a little differently. The first summation ranges over the states $1 \ldots n$. The reason that f^* is not checked over the empty interval σ_0 is because f^* holds over any single state interval. Consequently, the second summation ranges over $0 \ldots (n-1)$.

	Example formula	$T(n,f_1)$	$T(n,f_2)$	$T(n,f_1 \; ; \; f_2)$
(a)	$R \; ; \; S$	1	1	$2n+2$
(b)	$Q \; ; (R \; ; \; S)$	1	$2n+2$	n^2+4n+3
(c)	$P \; ; (Q \; ; (R \; ; \; S))$	1	n^2+4n+3	$\frac{n^3}{3}+\frac{5n^2}{2}+\frac{37n}{6}+4$
(d)	$(P \; ; \; Q) \; ; (R \; ; \; S)$	$2n+2$	$2n+2$	$2n^2+6n+4$

Figure 3.2: Example timings for $T(n, f_1 \; ; \; f_2)$

Figure 3.2 illustrates some sample formulae and associated worst-case timings. An example interval for both 3.2(c) and 3.2(d) is given below ($\bullet$ = proposition holds in state):

$$\sigma = \begin{bmatrix} P: & \bullet & & & \\ Q: & \bullet & \bullet & \dots & \bullet \\ R: & \bullet & \bullet & \dots & \bullet \\ S: & & & & \\ & \sigma_0 & \sigma_1 & \dots & \sigma_n \end{bmatrix} \quad \begin{array}{l} \sigma \not\models (P \ ; \ Q) \ ; (R \ ; \ S) \\ \sigma \not\models P \ ; (Q \ ; (R \ ; \ S)) \end{array}$$

Figure 3.3 illustrates the expression trees for $P \ ; (Q \ ; (R \ ; \ S))$ and $(P \ ; \ Q) \ ; (R \ ; \ S)$ in which the trees have depths of 3 and 2 respectively. Worst case evaluation is $\mathcal{O}(2^d)$ where d is the depth of the expression tree.

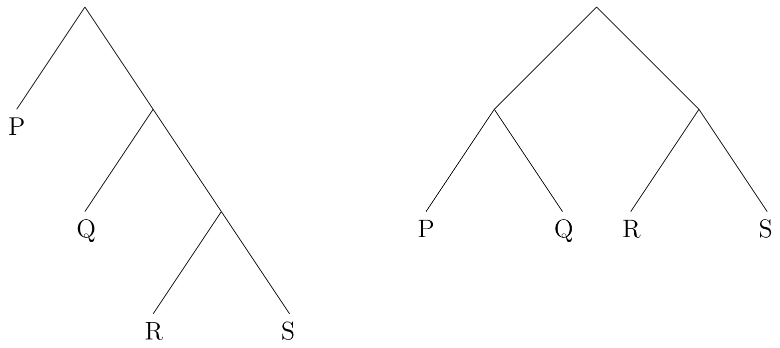


Figure 3.3: Expression trees representing $P \ ; (Q \ ; (R \ ; \ S))$ and $(P \ ; \ Q) \ ; (R \ ; \ S)$

The formula for $T(n, f^*)$ on page 45 is derived from the ITL equivalence $\vdash \ f^* \equiv (\mathsf{empty} \lor ((f \land \mathsf{more}) \ ; \ f^*))$ ($\mathit{ChopStarEqv}^{(C.46)}$). Worst-case performance assumes that all non-empty subintervals satisfy f except all terminal suffixes: a situation that can be described in ITL by the formula $((\boxed{a} \ (\mathsf{more} \supset f)) \ ; \ \mathsf{skip}) \ \land \ \Box \neg f$.

Equation 3.2.1 $T(n, f^*) = \sum_{j=1}^{n} 2^{n-j}\, T(j, f), \quad n > 0$

Proof of Equation 3.2.1 is by induction.

Case $n = 1$

$$T(1, f^*) = T(1, f) + T(0, f^*) = T(1, f) = \sum_{j=1}^{1} 2^{1-j}\, T(j, f)$$

Case $n + 1$

$$\begin{aligned} T(n+1, f^*) &= \left(\sum_{k=1}^{n+1} T(k, f) \right) + \left(\sum_{k=0}^{n} T(k, f^*) \right) \\ &= T(n+1, f) + T(n, f^*) + \left(\sum_{k=1}^{n} T(k, f) \right) + \left(\sum_{k=0}^{n-1} T(k, f^*) \right) \end{aligned}$$

$$
\begin{aligned}
&= T(n+1,f) + T(n,f^*) + T(n,f^*)\\
&= T(n+1,f) + 2T(n,f^*)\\
&= T(n+1,f) + 2\left(\sum_{j=1}^{n} 2^{n-j}\,T(j,f)\right) && \text{Equation 3.2.1 by induction}\\
&= T(n+1,f) + \left(\sum_{j=1}^{n} 2^{n+1-j}\,T(j,f)\right)\\
&= \sum_{j=1}^{n+1} 2^{n+1-j}\,T(j,f)
\end{aligned}
$$

Example 3.2.1 The formula $P \wedge \mathsf{fin}\, P$ holds for any interval in which proposition P holds in both the initial and final states. $T(n, P \wedge \mathsf{fin}\, P) = 3$, therefore $T(n,(P \wedge \mathsf{fin}\, P)^*) = 3 \times \sum_{j=1}^{n} 2^{n-j} = 3 \times (2^n - 1)$. ■

As discussed in [MGL14], such exponential complexity makes it is infeasible to perform runtime verification of ITL formulae containing multiple chop operators over nontrivial intervals.

3.3 Determining fusion points

The worst-case performance described in the previous section can be reduced significantly by constraining the number of potential fusion points that need to be considered when checking formulae containing chop and chopstar. The rest of this chapter develops a framework in ITL that supports such an approach.

The timing function for $f_1 \,;\, f_2$ can be modified if it is known that there exists a unique fusion point whereby the interval satisfies the formula. Suppose that $\sigma \models f_1 \,;\, f_2$ holds, and that m, $0 \leq m \leq n = |\sigma|$, is the unique fusion point, then

Equation 3.3.1 $T(n, f_1 \overset{m}{;} f_2) = \left(\sum_{k=0}^{m} T(k, f_1)\right) + T(n-m, f_2)$

The chop operator has been annotated to indicate the unique length of the prefix interval (to the 'left' of the fusion point). The summation captures the iteration through the prefix intervals until the unique fusion point m is determined. The algorithm does not assume prior knowledge of the unique fusion point m although it may be pre-determined with certain formulae such as, for example, $\mathsf{len}(5) \overset{5}{;} f$. [1]

[1] Code in the implementation of the ITL library that exploits such fixed-length formulae is shown on page 100.

The associativity of the chop operator means that it is possible to rewrite formulae such as $(f_1 \; ; \; f_2) \; ; \; f_3$ as $f_1 \; ; (f_2 \; ; \; f_3)$. To see why it is better to use a right-parenthesised form, consider each in turn (see also Figure 3.2):

- Left-parenthesised:

$$
\begin{aligned}
& T(n, (f \overset{m}{;} g) \overset{m+m'}{;} h) \\
&= \left(\sum_{k=0}^{m+m'} T(k, f \overset{m}{;} g) \right) + T(n-m-m', h) \\
&= \left(\sum_{k=0}^{m+m'} \left(\left(\sum_{j=0}^{m} T(j, f) \right) + T(m', g) \right) \right) + T(n-m-m', h)
\end{aligned}
$$

  The outermost sum repeatedly tries to establish $\sigma_0 \mathinner{..} \sigma_{m+m'} \models f \; ; \; g$, $(m + m' + 1)$ times. In the context of runtime verification, the innermost summations (over j) would be bounded by k because each test would take place over the most recent prefix interval $\sigma_0 \mathinner{..} \sigma_k$. Using $\downarrow$ and $\uparrow$ to represent the *min* and *max* functions respectively, the formula would be:

$$
\left(\sum_{k=0}^{m+m'} \left(\left(\sum_{k=0}^{k \downarrow m} T(k, f) \right) + T((k-m') \uparrow 0, g) \right) \right) + T(n-m-m', h)
$$

  Nevertheless, the nested summation indicates an unnecessary complexity when compared to right-parenthesisation.

- Right-parenthesised:

$$
\begin{aligned}
& T(n, f \overset{m}{;} (g \overset{m'}{;} h)) \\
&= \left(\sum_{k=0}^{m} T(k, f) \right) + \left(\sum_{k=0}^{m'} T(k, g) \right) + T(n-m-m', h)
\end{aligned}
$$

  In contrast to the formula for $T(n, f_1 \; ; \; f_2)$ given on page 45, in this case a fully right-parenthesised formula is advantageous. There is no need for an algorithm to backtrack across any of the fixed fusion points. This means that evaluation can proceed linearly, which conveniently aligns with the execution requirements of runtime verification.

Figure 3.4 illustrates some right-parenthesised formulae with their associated timings.

	Example formula	$T(n, f_1)$	$T(n, f_2)$	$T(n, f_1 \; ; \; f_2)$
(a)	$R \overset{m''}{;} S$	1	1	$m'' + 2$
(b)	$Q \overset{m'}{;} (R \overset{m''}{;} S)$	1	$m'' + 2$	$m' + m'' + 3$
(c)	$P \overset{m}{;} (Q \overset{m'}{;} (R \overset{m''}{;} S)$	1	$m' + m'' + 3$	$m + m' + m'' + 4$
(d)	$(Q \; ; \; R) \overset{m''}{;} S$	$2n + 2$	1	$\left(\sum_{k=0}^{m''} 2k + 2\right) + 1$ $= (m'')^2 + 3m'' + 3$
(e)	$(P \; ; \; Q) \overset{m'}{;} (R \overset{m''}{;} S)$	$2n + 2$	$m'' + 2$	$\left(\sum_{k=0}^{m'} 2k + 2\right) + m'' + 2$ $= (m')^2 + 3m' + m'' + 4$

Figure 3.4: Example timings for $T(n, f_1 \overset{m}{;} f_2)$

In a similar way, a timing formula for f^* can be constructed upon the assumption that each f is satisfied over a unique, non-empty subinterval. As before, let n be the length of the interval σ, i.e. $n = |\sigma|$. Let ms be a sequence of non-zero subinterval lengths such that $sum(ms) < n$. The *chopstar* operator $(^*)$ is annotated with the sequence representing the fusion points. For example, $f \overset{\langle m, m', m'' \rangle}{*} = f \overset{m}{;} (f \overset{m'}{;} (f \overset{m''}{;} f))$.

Note that:

(i) The fusion points are only hypothetical for the purpose of estimating the timing formula;

(ii) The interval lengths are non-zero because empty sub-intervals do not contribute towards establishing $\sigma \models f^*$.

Equation 3.3.2 $$T(n, f \overset{ms}{*}) = \begin{cases} 0, & \text{if } n = 0 \\ T(n, f), & \text{if } n \geq 1 \text{ and } ms = \langle\rangle \\ T(n, f \overset{hd(ms)}{;} f \overset{tl(ms)}{*}), & \text{if } n \geq 1 \text{ and } ms \neq \langle\rangle \\ \text{where } hd \text{ and } tl \text{ are the sequence head and tail functions} \end{cases}$$

In order to simplify the analysis of $T(n, f \overset{ms}{*})$, assume that the interval σ can be split into l

subintervals of length m, i.e. $lm = n$, and that for each $1 \leq i \leq l, \sigma_{(i-1)m} \ldots \sigma_{im} \models f$. This provides a simplified formula:

Equation 3.3.3 $T(l \times m, f^*) = l \times \left(\sum_{k=0}^{m} T(k, f) \right)$

Thus, comparing equations (3.2.1) and (3.3.3) the former grows exponentially whereas the latter grows linearly.

Reconsider the previous example 3.2.1 (page 47) $(P \wedge \mathsf{fin}\, P))^*$. Assume that $|\sigma| = n = l \times m$ for some l and m. Using equation (3.2.1) the worst-case timing to establish satisfaction of an interval was calculated to be $\mathcal{O}(2^n)$. Using equation (3.3.3), the timing is calculated thus: $l \times (\sum_{k=0}^{m} 3) = 3l(m+1) = 3n + 3l$ which is $\mathcal{O}(n)$.

3.3.1 Introducing first occurrence

In the previous section when considering a formula such as $f_1 \overset{m}{;} f_2$ it was assumed that the prefix over which f_1 was satisfied had length m and that this was uniquely determined. This thesis introduces a new operator, $\triangleright$, into ITL which specifies the *first occurrence* of a formula. Specifically, if $\sigma \models \triangleright f$ then there is no strict prefix interval that satisfies f: i.e. for all $k < |\sigma|$, $\sigma_0 \ldots \sigma_k \not\models f$. The semantics of $\triangleright f$ is given below.

Equation 3.3.4 $\mathcal{F}[\![\triangleright f]\!](\sigma) = \mathsf{tt}$ iff $\mathcal{F}[\![f]\!](\sigma) = \mathsf{tt}$ and for all $0 \leq i < |\sigma|$, $\mathcal{F}[\![\neg f]\!](\sigma_0 \ldots \sigma_i) = \mathsf{tt}$

This operator can be used in conjunction with a chop operator to ensure that a fusion point is uniquely determined, e.g. $f_1 \wedge f$. The intuition is that if any prefix interval satisfies f_1, i.e. $\sigma \models \diamondsuit f_1$, then *at least one* prefix of σ satisfies f_1, and $\triangleright f_1$ specifies the shortest (first) such prefix ($DiImpExistsOneDiLenAndFst^{(C.251)}$).

In the context of runtime verification, the primary role of the $\triangleright$ operator is to define a unique partitioning of the incoming interval. The importance of establishing deterministic fusion points using formulae such as $\triangleright f_1 ; (\triangleright f_2 ; \ldots$ is that no backtracking across these fusion points is necessary – because these are the *only* candidate fusion points.

3.3.2 Non-determinism

The operator $\triangleright$ can be used to structure specifications sequentially to facilitate efficient runtime verification: $\triangleright f_1 ; \triangleright f_2 ; \ldots$. However, it is possible to combine each deterministic formula, $\triangleright f_i$, with a non-deterministic formula, g_i, in the following way: $(\triangleright f_1 \wedge g_1) ; (\triangleright f_2 \wedge g_2) ; \ldots$. For example, the pattern illustrated in Figure 3.5 demonstrates the formula $(\triangleright f_1 \wedge g_1) ; (\triangleright f_2 \wedge g_2) ; (\triangleright f_3 \wedge g_3) ; (\triangleright f_4 \wedge g_4)$. If this formula is satisfied by an interval then so is $\rhd(f_1 \wedge g_1) ; \rhd(f_2 \wedge g_2) ; \rhd(f_3 \wedge g_3) ; \rhd(f_4 \wedge g_4)$ due to the laws *FstWithAndImp*(C.223) and *LeftChopImpChop*(C.101).

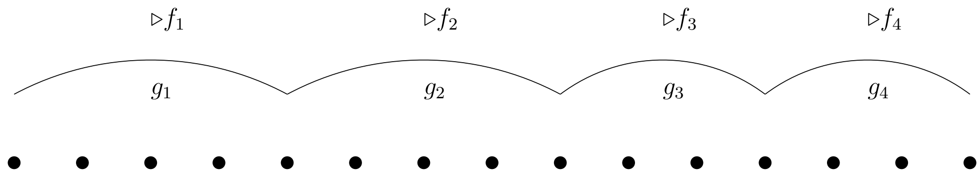


Figure 3.5: Splitting an incoming interval into subintervals using $\rhd$. Each $\rhd f_i$ is deterministic whereas each of the g_i may be non-deterministic formulae.

This pattern allows each $(\rhd f_i \wedge g_i)$ to combine a 'control' element, $\rhd f_1$, which uniquely determines the next fusion point, and a 'payload' element, g_i, which may include any ITL formula. The complexities of the subformulae f_i and g_i will, of course, determine the overall complexity of evaluating $\rhd f_i \wedge g_i$ but, the 'payload' formula needs to be calculated at most once.

To analyse the performance assume that $\sigma \models (\rhd f_1 \wedge f) ; f_2$ for some interval σ, and therefore that there exists a unique fusion point m such that $0 \leq m \leq |\sigma|$. In this case it follows that $\sigma_0 .. \sigma_{m-1} \models \boxdot \neg f_1$, $\sigma_0 .. \sigma_m \models f_1 \wedge f$, and $\sigma_m .. \sigma_{|\sigma|} \models f_2$. The number of tests required to discover m, and establish that the formula holds, is given by:

Equation 3.3.5 $T(n, (\rhd f_1 \wedge f) \overset{m}{;} f_2) = \left(\sum_{k=0}^{m} T(k, f_1)\right) + T(m, f) + T(n-m, f_2)$

The rationale for this is as follows: f_1 must be tested for each prefix interval until the fusion point (m) is found. At this point the conjoined formula f must be tested to check that it holds over the same prefix. Finally, the second formula, f_2, must be tested on the remaining suffix interval.

The case when a control and payload combination is repeated is given by the formula $(\rhd f \wedge f)^*$. The timing analysis for this formula is given below:

Equation 3.3.6

$$T(n, (\rhd f_1 \wedge f)\overset{ms}{^*}) = \begin{cases} 0, & \text{if } n = 0 \\ \left(\sum_{k=0}^{n} T(k, f_1)\right) + T(n, f), & \text{if } n \geq 1 \text{ and } ms = \langle\rangle \\ T(n, (\rhd f_1 \wedge f) \overset{hd(ms)}{;} (\rhd f_1 \wedge f)\overset{tl(ms)}{^*}), & \text{if } n \geq 1 \text{ and } ms \neq \langle\rangle \end{cases}$$

Once again, for the purpose of analysing performance, it is useful to consider a special case in which the interval can be split into $l \times m$ subintervals each of which satisfies $\rhd f_1 \wedge f$.

Equation 3.3.7 $T(l \times m, (\rhd f_1 \wedge f)^*) = l \times \left(\sum_{k=0}^{m} T(k, f_1) \right) + l \times T(m, f)$

3.4 Managing termination

The runtime monitors that are the subject of this thesis each represent a formula in (finite) ITL. Therefore they are designed to terminate, and the termination condition is part of the specification. There are three principal patterns of termination whose templates are presented below. Note that although a monitor is designed to verify a finite execution trace, the termination condition can be dependent upon the system being verifiied and, in particular, may be a $STOP$ signal issued when the system is itself ready to halt.

Monitor specifications often utilise iterated subformulae. The templates below consider two cases in which the termination condition aligns with the end of a repeated subinterval, and one in which it does not. The latter is more complicated, requiring a specification of how a subinterval may be interrupted successfully.

Template 3.4.1 Iteration

$$(\rhd f \wedge g)^k$$

It is straightforward to determine the length of a runtime verification by repeating a specific formula a given number of times. The interval is split into a sequence of k deterministic subintervals, each specified by $\rhd f$. If g also holds within each subinterval then the verification succeeds.

Template 3.4.2 Managed halt

$$\mathsf{halt}(w) \wedge (\rhd f \wedge g)^* \qquad \textit{provided that} \ \ \mathsf{halt}(w) \supset (\rhd f)^*$$

If it is known (or can be arranged) that a terminating condition w always aligns with the termination of a subinterval, then a 'managed halt' can be specified. Figure 3.6 illustrates a managed halt showing how the terminating condition must align with a deterministic fusion point.

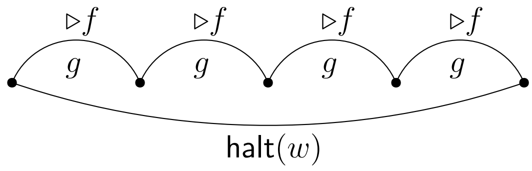


Figure 3.6: A managed halt. $\mathsf{halt}(w) \supset (\rhd f)^*$.

Template 3.4.3 Exception

$$\mathsf{halt}(w) \wedge (\rhd(\mathsf{fin}(w) \vee f) \wedge (\mathsf{fin}(w) \vee g))^*$$

The formula $\mathsf{halt}(w)$ specifies that the runtime verification must terminate as soon as state formula w holds – this could be a propositional variable, $STOP$, for example. The termination condition represents an *exceptional* condition in the sense that it may occur part way through an iteration.

The formula $\rhd(\mathsf{fin}(w) \vee f)$ specifies the shortest interval such that either w occurs in the final state or the interval satisfies f. If $\rhd f$ is established, and the associated subinterval does not satisfy w in its final state, then formula g must be satisfied. However, if the subinterval satisfies $\rhd f$ and $\mathsf{fin}(w)$ then g is not required to hold. Figure 3.7 illustrates an exceptional termination under two different circumstances.

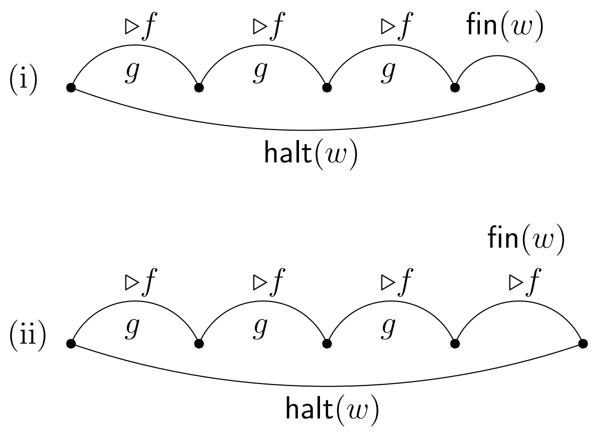


Figure 3.7: A exceptional termination: (i) non-aligned and, (ii) aligned with $\rhd f$.

Example 3.4.1 A critical section is managed by a counting semaphore S whose value can vary between 0 and n.

$$\mathsf{halt}(STOP) \wedge$$

$$(\ \rhd(\mathsf{fin}(STOP) \vee \neg\ \mathsf{stable}(S)) \\ \wedge (\mathsf{fin}(STOP) \vee ((abs(\mathsf{fin}(S) - S) = 1) \wedge (\mathsf{fin}(S) \geq 0) \wedge (\mathsf{fin}(S) \leq n)) \ )^*$$

The formula $\mathsf{stable}(S)$ requires that the value of S remains constant throughout the interval. The first occurrence of $\neg\ \mathsf{stable}(S)$ therefore is the smallest initial interval for which S is constant in all but the last state. Whenever S changes its value must have increased or decreased by one and its value must remain within the semaphore limits of 0 and n. ■

3.5 Properties of interval length

The first occurrence operator $\rhd$ was introduced in Section 3.3.1. Due to its extensive rôle in constructing ITL-Monitor specifications, the relationship between $\rhd$ and other ITL operators must be explored. The investigation begins with intervals of fixed length and introduces the notion of a fixed-length formula which specifies such an interval.

Definition 3.1 Fixed-length formula *A formula f is a* fixed-length formula *if, whenever $\sigma \models f$ for some interval σ, there is no $k < |\sigma|$ such that $\sigma_0 \ldots \sigma_k \models f$.*

ITL contains three formulae that specify intervals of specific length: empty denotes a single-state interval; skip denotes a two-state (unit) interval; and $\mathsf{len}(k)$ defines an interval with $k+1$ states. The formula $\mathsf{halt}(w)$ also specifies a fixed-length interval given that this determines that w must hold only in the final state. $\rhd$ is more general than halt but acts in a similar way to determine an interval of unique length.

There is not a body of laws available in the literature relating to ITL intervals of fixed length and the development below constitutes an addition in this area. The proofs of the laws in this section are recorded in Section 6.4 of [CMS19].

3.5.1 Interval length

$\mathsf{len}(k)$ specifies that the interval length is k – i.e. that the interval has $k+1$ states.

$$\mathcal{F}[\![\mathsf{len}(k)]\!](\sigma) = \mathsf{tt} \quad \text{iff} \quad |\sigma| = k \qquad \textit{LenIffModSig}^{(C.150)}$$

The definitions of iteration and interval length are given below:

$$f^0 \ \widehat{=}\ \mathsf{empty} \qquad \textit{IterZeroDef}^{(C.23)}$$
$$f^{n+1} \ \widehat{=}\ f\ ;\ f^n, \quad [n \geq 0] \qquad \textit{IterDef}^{(C.24)}$$
$$\mathsf{len}(n) \ \widehat{=}\ \mathsf{skip}^n \qquad \textit{LenDef}^{(C.35)}$$

It follows directly that:

$\vdash \quad \mathsf{len}(0) \equiv \mathsf{empty}$ *LenZeroEqvEmpty*(C.143)
$\vdash \quad \mathsf{len}(1) \equiv \mathsf{skip}$ *LenOneEqvSkip*(C.144)
$\vdash \quad \mathsf{len}(n+1) \equiv \mathsf{skip} \ ; \ \mathsf{len}(n)$ *LenNPlusOneA*(C.145)

Therefore, an interval with length $i+j, (i, j \geq 0)$ can be chopped into two intervals of length i and j respectively.

$\vdash \quad \mathsf{len}(i+j) \equiv \mathsf{len}(i) \ ; \ \mathsf{len}(j)$ *LenEqvLenChopLen*(C.146)

Using *LenEqvLenChopLen*(C.146) (setting $i = n, j = 1$), and *LenOneEqvSkip*(C.144) it follows that

$\vdash \quad \mathsf{len}(n+1) \equiv \mathsf{len}(n) \ ; \ \mathsf{skip}$ *LenNPlusOneB*(C.147)

Note that every (finite) interval, σ, must have a (finite) length, $|\sigma|$, and therefore, this tautological statement can be conjoined with any formula.

$\vdash \quad \exists\, k \bullet \mathsf{len}(k)$ *ExistsLen*(C.148)
$\vdash \quad f \equiv f \wedge \exists\, k \bullet \mathsf{len}(k)$ *AndExistsLen*(C.149)

3.5.2 Laws with fixed-length formulae

A fixed-length formula on either side of a chop operator uniquely determines the fusion point. Straightforward examples of this include, e.g., $(f \ ; \ \mathsf{skip})$ or $(\mathsf{len}(5) \ ; \ g)$. The following two laws capture the way in which chop can distribute through conjunction when the fusion point is deterministic.

$\vdash \quad (f \wedge \mathsf{len}(k)) \ ; \ p \wedge (g \wedge \mathsf{len}(k)) \ ; \ q \equiv (f \wedge g \wedge \mathsf{len}(k)) \ ; (p \wedge q)$ *LFixedAndDistr*(C.151)
$\vdash \quad p \ ; (f \wedge \mathsf{len}(k)) \wedge q \ ; (g \wedge \mathsf{len}(k)) \equiv (p \wedge q) \ ; (f \wedge g \wedge \mathsf{len}(k))$ *RFixedAndDistr*(C.152)

In ITL $((f_1 \wedge f_2) \ ; \ p)$ only implies $(f_1 \ ; \ p) \wedge (f_2 \ ; \ p)$. However, if the lengths of intervals defined by f_1 and f_2 are equal then this can be strengthened to an equivalence. Four symmetrical specialisations of the above laws are presented below:

$\vdash \quad (f \wedge \mathsf{len}(k)) \ ; \ p \wedge (g \wedge \mathsf{len}(k)) \ ; \ p \equiv (f \wedge g \wedge \mathsf{len}(k)) \ ; \ p$ *LFixedAndDistrA*(C.153)
$\vdash \quad (f \wedge \mathsf{len}(k)) \ ; \ p \wedge (f \wedge \mathsf{len}(k)) \ ; \ q \equiv (f \wedge \mathsf{len}(k)) \ ; (p \wedge q)$ *LFixedAndDistrB*(C.154)
$\vdash \quad p \ ; (f \wedge \mathsf{len}(k)) \wedge p \ ; (g \wedge \mathsf{len}(k)) \equiv p \ ; (f \wedge g \wedge \mathsf{len}(k))$ *RFixedAndDistrA*(C.155)
$\vdash \quad p \ ; (f \wedge \mathsf{len}(k)) \wedge q \ ; (f \wedge \mathsf{len}(k)) \equiv (p \wedge q) \ ; (f \wedge \mathsf{len}(k))$ *RFixedAndDistrB*(C.156)

3.5.3 Fixed-length formulae and negation

There are few useful laws involving the negation of formulae containing chop due to the non-determinism of the fusion point. There are some exceptions including the unit length interval skip. If the fusion point is uniquely determined, irrespective of where it occurs, then formulae can be negated easily.

$$\vdash\ \neg\,(f\ ;\ h) \equiv \neg\,\Diamond h \vee (\neg\,f\ ;\ h) \quad \text{where}\ \ h \equiv g \wedge \mathsf{len}(k) \qquad \mathit{NotChopFixed}^{(C.160)}$$
$$\vdash\ \neg\,(h\ ;\ f) \equiv \neg\,\Diamond_{\mathsf{i}}\, h \vee (h\ ;\ \neg\,f) \quad \text{where}\ \ h \equiv g \wedge \mathsf{len}(k) \qquad \mathit{NotFixedChop}^{(C.161)}$$

Cau had previously communicated the following laws[2] that involve negation with chop and skip. They can both be derived from the latter, more general laws, by setting $h \equiv \mathsf{skip}$.

$$\vdash\ \neg\,(\mathsf{skip}\ ;\ \neg\,g) \equiv \mathsf{empty} \vee (\mathsf{skip}\ ;\ g) \qquad \mathit{NotSkipNotChop}^{(C.129)}$$
$$\vdash\ \neg\,(\neg\,f\ ;\ \mathsf{skip}) \equiv \mathsf{empty} \vee (f\ ;\ \mathsf{skip}) \qquad \mathit{NotNotChopSkip}^{(C.130)}$$

3.6 Strict initial intervals

This section introduces the new ITL operators $\boxed{\mathsf{s}}$ (all strict initial intervals) and $\Diamond_{\mathsf{s}}$ (some strict initial interval) along with their reflected counterparts, $\boxed{\mathsf{t}}$ (all strict final intervals) and $\Diamond_{\mathsf{t}}$ (some strict final interval). Their definitions and the investigation into their properties form part of the original contribution of this thesis. $\boxed{\mathsf{s}}$ is required for the definition of the first occurrence operator, $\triangleright$, which was introduced in Section 3.3.1 and which is defined formally in Section 3.7. Following the introduction of the new operators, the focus will concentrate upon $\boxed{\mathsf{s}}$ and $\Diamond_{\mathsf{s}}$ alone. All of their properties have equivalent versions for $\boxed{\mathsf{t}}$ and $\Diamond_{\mathsf{t}}$ under time reversal [Mos14b].

Firstly, the behaviour of $\boxed{\mathsf{s}}$ and $\boxed{\mathsf{t}}$ is illustrated diagramatically in Figure 3.8.

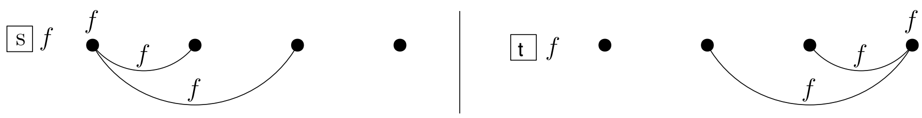


Figure 3.8: all strict prefixes, $\boxed{\mathsf{s}}$, and all strict suffixes, $\boxed{\mathsf{t}}$.

Note that $\boxed{\mathsf{s}}\,f$ (and $\boxed{\mathsf{t}}\,f$) hold vacuously over an empty interval. Their duals, $\Diamond_{\mathsf{s}}\,f$ and $\Diamond_{\mathsf{t}}\,f$, do *not* hold over the empty interval. The formal definitions of these new operators are given below:

[2] Private communication but the laws are included in [CMS19].

$$\boxed{s}\, f \;\widehat{=}\; \mathsf{empty} \vee \boxed{i}\, f \;;\; \mathsf{skip} \qquad \mathit{BsDef}^{(C.185)}$$
$$\Diamond_{s}\, f \;\widehat{=}\; \neg \boxed{s} \neg f \qquad \mathit{DsDef}^{(C.186)}$$
$$\boxed{t}\, f \;\widehat{=}\; \mathsf{empty} \vee \mathsf{skip} \;;\; \Box f \qquad \mathit{BtDef}^{(C.215)}$$
$$\Diamond_{t}\, f \;\widehat{=}\; \neg \boxed{t} \neg f \qquad \mathit{DtDef}^{(C.216)}$$

The effect of each these operators can be appreciated more readily by considering their semantics.

$$\mathcal{F}[\![\boxed{s}\, f]\!](\sigma) = \mathsf{tt} \text{ iff for all } 0 \le i < |\sigma|, \mathcal{F}[\![f]\!](\sigma_0 \mathinner{\ldotp\ldotp} \sigma_i) = \mathsf{tt}$$
$$\mathcal{F}[\![\Diamond_{s}\, f]\!](\sigma) = \mathsf{tt} \text{ iff exists } 0 \le i < |\sigma|, \mathcal{F}[\![f]\!](\sigma_0 \mathinner{\ldotp\ldotp} \sigma_i) = \mathsf{tt}$$
$$\mathcal{F}[\![\boxed{t}\, f]\!](\sigma) = \mathsf{tt} \text{ iff for all } 1 \le i \le |\sigma|, \mathcal{F}[\![f]\!](\sigma_i \mathinner{\ldotp\ldotp} \sigma_{|\sigma|}) = \mathsf{tt}$$
$$\mathcal{F}[\![\Diamond_{t}\, f]\!](\sigma) = \mathsf{tt} \text{ iff exists } 1 \le i \le |\sigma|, \mathcal{F}[\![f]\!](\sigma_i \mathinner{\ldotp\ldotp} \sigma_{|\sigma|}) = \mathsf{tt}$$

$\Diamond_{s}$ and $\Diamond_{t}$ satisfy a range of useful algebraic properties which are listed below.

Distributive laws

$$\vdash\; \boxed{s}\, f \wedge \boxed{s}\, g \equiv \boxed{s}\, (f \wedge g) \qquad \mathit{BsAndEqv}^{(C.194)}$$
$$\vdash\; \Diamond_{s}\, f \vee \Diamond_{s}\, g \equiv \Diamond_{s}\, (f \vee g) \qquad \mathit{DsOrEqv}^{(C.195)}$$
$$\vdash\; \boxed{s}\, f \vee \boxed{s}\, g \supset \boxed{s}\, (f \vee g) \qquad \mathit{BsOrImp}^{(C.196)}$$
$$\vdash\; \Diamond_{s}\, (f \wedge g) \supset \Diamond_{s}\, f \wedge \Diamond_{s}\, g \qquad \mathit{DsAndImp}^{(C.197)}$$

Absorption laws

$$\vdash\; \Diamond_{i}\, f \vee \Diamond_{s}\, f \equiv \Diamond_{i}\, f \qquad \mathit{DiOrDsEqvDi}^{(C.207)}$$
$$\vdash\; \Diamond_{i}\, f \wedge \Diamond_{s}\, f \equiv \Diamond_{s}\, f \qquad \mathit{DiAndDsEqvDs}^{(C.208)}$$

Complement laws

$$\vdash\; f \vee \Diamond_{s}\, f \equiv \Diamond_{i}\, f \qquad \mathit{OrDsEqvDi}^{(C.209)}$$
$$\vdash\; f \wedge \boxed{s}\, f \equiv \boxed{i}\, f \qquad \mathit{AndBsEqvBi}^{(C.210)}$$

Laws relating to $\boxed{i}$ and $\Diamond_{i}$

$$\vdash\; \Diamond_{s}\, f \equiv \Diamond_{i}\, f \;;\; \mathsf{skip} \qquad \mathit{DsDi}^{(C.188)}$$
$$\vdash\; \boxed{s}\, f \equiv \boxed{i}\, (\mathsf{more} \supset f \;;\; \mathsf{skip}) \qquad \mathit{BsEqvBiMoreImpChop}^{(C.214)}$$
$$\vdash\; \boxed{i}\, f \supset \boxed{s}\, f \qquad \mathit{BiImpBs}^{(C.200)}$$
$$\vdash\; \boxed{s}\, f \supset \boxed{s}\, \boxed{s}\, f \qquad \mathit{BsImpBsBs}^{(C.201)}$$
$$\vdash\; \boxed{s}\, f \equiv \boxed{s}\, \boxed{i}\, f \qquad \mathit{BsEqvBsBi}^{(C.211)}$$
$$\vdash\; f \supset g \quad\Rightarrow\quad \vdash\; \boxed{s}\, f \supset \boxed{s}\, g \qquad \mathit{BsImpBsRule}^{(C.203)}$$

$\vdash \ \diamondsuit_s (f \ ; \ g) \supset \diamondsuit_s f$ *DsChopImpDsB*(C.204)

$\vdash \ \boxed{s} f \vee \boxed{s} g \equiv \boxed{s} (\boxed{i} f \vee \boxed{i} g)$ *BsOrBsEqvBsBiOrBi*(C.206)

State formulae

A state formula w refers only to variables in the first state (see Section 2.1.3.2). Therefore if w holds (in the first state) of an interval then it must hold over all non-empty, strict initial intervals. In the case that the interval is empty, then $\boxed{s}\, w$ holds vacuously (*BsDef*(C.185)).

$\vdash \ w \supset \boxed{s}\, w$ *StateImpBs*(C.212)

Time reversal

Recent developments in ITL include Moszkowski's work on time reversal [Mos14b]. Both $\boxed{s}$ and $\diamondsuit_s$ operate in 'forward time' – i.e. over $\sigma_0 \mathinner{..} \sigma_k$ for *increasing* k. However, each of these operators has a counterpart under time reversal, namely $\boxed{t}$ and $\diamondsuit_t$.

In keeping with established practice in ITL, these are referred to informally as "box-t" and "diamond-t" respectively.[3] The following theorems state that each operator is the reflection of its counterpart:

$\vdash \ (\boxed{s} f)^r \equiv \boxed{t} f^r$ *BsrEqvBtr*(C.217)

$\vdash \ (\diamondsuit_s f)^r \equiv \diamondsuit_t f^r$ *DsrEqvDtr*(C.218)

$\vdash \ (\boxed{t} f)^r \equiv \boxed{s} f^r$ *BtrEqvBsr*(C.219)

$\vdash \ (\diamondsuit_t f)^r \equiv \diamondsuit_s f^r$ *DtrEqvDsr*(C.220)

All of the laws relating to $\boxed{s}$ and $\diamondsuit_s$ have related laws for $\boxed{t}$ and $\diamondsuit_t$ under time reversal. These are not considered further in this thesis since they are not required for the development of runtime monitors, nor the operator $\rhd$ upon which they are based. Laws relating to these operators are available in Section 6.2 of [CMS19].

3.7 Formalisation of the first occurrence operator

The introduction of the first occurrence operator $\rhd$ along with a thorough investigation of its properties is one of the main contributions of this thesis. Its rôle in restricting the chop operator was described in Section 3.3.1. The formal definition of $\rhd f$ is given below.

$\rhd f \ \ \widehat{=} \ \ f \wedge \boxed{s} \neg f$ *FstDef*(C.222)

[3] The use of 't' was selected because, while it is usefully follows 's' alphabetically, 't' can bring to mind the word "tail". The reflected operators $\boxed{t}$ and $\diamondsuit_t$ refer to strict suffixes (tails).

The definition reflects its semantics (Equation 3.3.4, page 50) repeated below for convenience, and comprises two parts: (i) the whole interval satisfies f, and (ii) no strict initial interval satisfies f.

$$\mathcal{F}[\![\rhd f]\!](\sigma) = \mathsf{tt} \text{ iff } \mathcal{F}[\![f]\!](\sigma) = \mathsf{tt} \text{ and for all } 0 \leq i < |\sigma|,\ \mathcal{F}[\![\neg f]\!](\sigma_0 \mathinner{.\,.} \sigma_i) = \mathsf{tt}$$

Figure 3.9 illustrates an interval over which $\rhd f$ holds.

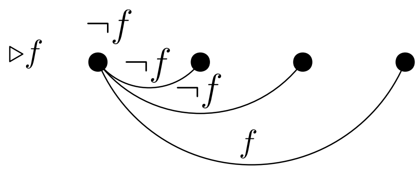


Figure 3.9: $\rhd f$

Other authors have proposed restricting the chop operator using a combination of first and last occurrences, most notably in Eagle [BB08] which was discussed previously in Section 2.4.3. The approach taken here is different: in this thesis no change to the semantics of the chop operator is proposed. The new operator $\rhd$, derived within existing ITL, is defined independently of chop, although one of its principal applications is to restrict chop in specific circumstances. In fact, $\rhd$ turns out to be a generalisation of halt – specifically, $\rhd(\mathsf{fin}(w)) \equiv \mathsf{halt}(w)$. A discussion of the relationship between $\rhd$ and halt is presented in Section 3.8.6.

$\rhd f$ denotes the *first* initial interval that satisfies f – i.e. no strict prefix satisfies f. Therefore the length of such a satifying interval is uniquely determined:

$$\vdash \ \diamondsuit\!\!\!\cdot\ (\rhd f \wedge \mathsf{len}(i)) \wedge \diamondsuit\!\!\!\cdot\ (\rhd f \wedge \mathsf{len}(j)) \supset i = j \qquad \textit{FstLenSame}^{(C.250)}$$

A related law states that if f holds for *some* initial interval then there is a unique initial interval that satisfies $\rhd f$ defined by its length.

$$\vdash \ \diamondsuit\!\!\!\cdot\ f \supset \exists_1\, k \bullet \diamondsuit\!\!\!\cdot\ (\mathsf{len}(k) \wedge \rhd f) \qquad \textit{DiImpExistsOneDiLenAndFst}^{(C.251)}$$

3.8 Algebraic properties of the first occurrence operator

In the remainder of this chapter the algebraic properties of $\rhd$ are investigated. The majority of the work below was developed to establish and reason about the semantics of the runtime monitors whose formal semantics will be given in Chapter 4. Indeed, it was the consideration of how one might compose runtime monitors that led to the development of $\rhd$ as a suitable structuring mechanism. These laws, along with their proofs, are presented in Sections 6.3 and 6.4 of [CMS19].

3.8.1 First with simple formulae

The shortest interval is empty.

$\vdash \quad \triangleright \mathsf{true} \equiv \mathsf{empty}$ *FstTrue*$^{(C.226)}$

Since no interval satisfies false[4] there can be no shortest interval that does so.

$\vdash \quad \triangleright \mathsf{false} \equiv \mathsf{false}$ *FstFalse*$^{(C.227)}$

The shortest interval that satisfies more (i.e. has at least two states) is an interval of unit length.

$\vdash \quad \triangleright \mathsf{more} \equiv \mathsf{skip}$ *FstMoreEqvSkip*$^{(C.233)}$

Any formula of the form, $\mathsf{len}(k)$ is equivalent to its own first occurrence. This is an obvious consequence of specifying a length.

$\vdash \quad \triangleright \mathsf{len}(k) \equiv \mathsf{len}(k)$ *FstLenEqvLen*$^{(C.272)}$

There are two special cases of *FstLenEqvLen*$^{(C.272)}$ deriving from *LenZeroEqvEmpty*$^{(C.143)}$ and *LenOneEqvSkip*$^{(C.144)}$:

$\vdash \quad \triangleright \mathsf{empty} \equiv \mathsf{empty}$ *FstEmpty*$^{(C.229)}$
$\vdash \quad \triangleright \mathsf{skip} \equiv \mathsf{skip}$ *FstSkip*$^{(C.273)}$

Furthermore, conjoining a formula f with empty renders the $\triangleright$ operator redundant.

$\vdash \quad \triangleright f \wedge \mathsf{empty} \equiv f \wedge \mathsf{empty}$ *FstAndEmptyEqvAndEmpty*$^{(C.230)}$

Conversely, disjoining formula f with empty renders f redundant.

$\vdash \quad \triangleright(\mathsf{empty} \vee f) \equiv \mathsf{empty}$ *FstEmptyOrEqvEmpty*$^{(C.231)}$

State formulae, denoted conventionally by w, do not include any temporal operators (Section 2.1.3.2) and hold whenever w is true in the first state. Therefore, $\triangleright w$ can only be satisfied

[4] In infinite-time ITL the formula $\mathsf{true}\ ;\ \mathsf{false}$ is satisfied by an infinite interval. However, this thesis only uses finite ITL.

by an empty interval:

$$\vdash \quad \triangleright w \equiv \mathsf{empty} \wedge w \qquad \textit{FstState}^{(C.243)}$$

3.8.2 First with conjunction and disjunction

$$\vdash \quad \triangleright f \wedge g \supset \triangleright(f \wedge g) \qquad \textit{FstWithAndImp}^{(C.223)}$$
$$\vdash \quad \triangleright(f \vee g) \equiv (\triangleright f \wedge \boxed{s}\,\neg\, g) \vee (\triangleright g \wedge \boxed{s}\,\neg\, f) \qquad \textit{FstWithOrEqv}^{(C.224)}$$

FstWithAndImp(C.223) states that if $\triangleright f$ and g are both satisfied then this implies that the interval satisfies the first occurrence of $f \wedge g$ together. This follows because $f \wedge g$ are satisfied by the interval and, since no strict initial interval satisfies f, this must be the first occurrence of the conjunction. *FstWithOrEqv*(C.224) separates the first occurrence of a disjunction $f \vee g$ into two cases: either the interval satisfies the first occurrence of f with no strict initial interval satisfying g, or vice versa. Note that this includes the *possibility* of $\triangleright f \wedge \triangleright g$.

It may appear that a corresponding law for $\triangleright(f \wedge g)$ would be appropriate. However, such a formula does not permit anything more interesting than $f \wedge g$, or $\boxed{s}\,\neg\,(f \wedge g)$, to be deduced. Both of these formulae follow directly from the definition *FstDef*(C.222), so nothing new is derived. The first occurrence of the conjunction does not preclude any number of strict initial intervals satisfying either f or g – but not both together. However, if the first conjunct is of the form $\triangleright f$ then there is a useful law which allows the introduction or elimination of $\triangleright$ around an expression of the form $\triangleright f \wedge g$.

$$\vdash \quad \triangleright(\triangleright f \wedge g) \equiv \triangleright f \wedge g \qquad \textit{FstFstAndEqvFstAnd}^{(C.225)}$$

Consider the first occurrence of a disjunction $\triangleright(f \vee g)$. Suppose an interval satisfies $\triangleright f$, then it only satisfies $\triangleright(f \vee g)$ if no strict initial interval satisfies g. Otherwise $\triangleright g$ would occur before $\triangleright f$. This is the basis of the law *FstWithOrEqv*(C.224).

$$\vdash \quad \triangleright(f \vee g) \equiv (\triangleright f \wedge \boxed{s}\,\neg\, g) \vee (\triangleright g \wedge \boxed{s}\,\neg\, f) \qquad \textit{FstWithOrEqv}^{(C.224)}$$

The following laws state that in the formula $\triangleright(f \vee g)$, both $f \equiv \triangleright f$ and $g \equiv \triangleright g$ hold. These laws are useful for moving $\triangleright$ outside of parentheses in proofs.

$$\vdash \quad \triangleright(\triangleright f \vee g) \equiv \triangleright(f \vee g) \qquad \textit{FstFstOrEqvFstOrL}^{(C.269)}$$
$$\vdash \quad \triangleright(f \vee \triangleright g) \equiv \triangleright(f \vee g) \qquad \textit{FstFstOrEqvFstOrR}^{(C.270)}$$
$$\vdash \quad \triangleright(\triangleright f \vee \triangleright g) \equiv \triangleright(f \vee g) \qquad \textit{FstFstOrEqvFstOr}^{(C.271)}$$

3.8.3 First with prefix intervals

Although $\boxed{s}\,\neg f \wedge f$ appears, at first sight, to be stronger than $\boxed{s}\,\neg f \wedge \diamondsuit f$, it turns out that they are, in fact, equivalent. This may be understood by considering the following argument: $\boxed{s}\,\neg f$ states that *no strict* initial interval satisfies f; therefore $\boxed{s}\,\neg f \supset (\diamondsuit f \equiv f)$. This leads to an alternative equivalence for $\rhd f$:

$$\vdash \ \rhd f \equiv \boxed{s}\,\neg f \wedge \diamondsuit f \qquad \textit{FstEqvBsNotAndDi}^{(C.257)}$$

The two laws, *FstOrDiEqvDi*(C.234) and *FstAndDiEqvFst*(C.235), permit the absorption of either $\rhd f$ or $\diamondsuit f$ depending upon whether they are disjoined or conjoined.

$$\vdash \ \rhd f \vee \diamondsuit f \equiv \diamondsuit f \qquad \textit{FstOrDiEqvDi}^{(C.234)}$$
$$\vdash \ \rhd f \wedge \diamondsuit f \equiv \rhd f \qquad \textit{FstAndDiEqvFst}^{(C.235)}$$

The two laws, *DiEqvDiFst*(C.236) and *FstDiEqvFst*(C.237), show that terms involving both of the operators $\diamondsuit$ and $\rhd$ in either order consecutively can be reduced by the removal of one of the operators.

$$\vdash \ \diamondsuit f \equiv \diamondsuit \rhd f \qquad \textit{DiEqvDiFst}^{(C.236)}$$
$$\vdash \ \rhd \diamondsuit f \equiv \rhd f \qquad \textit{FstDiEqvFst}^{(C.237)}$$

The laws below can be derived from *DiEqvDiFst*(C.236) and *FstDiEqvFst*(C.237). The second law, *DiOrFstAndEqvDi*(C.239), demonstrates an absorptive property, and the third law, *FstDiAndDiEqv*(C.240), shows how $\rhd$ limits the effect of $\diamondsuit$.

$$\vdash \ \diamondsuit f \wedge (\rhd f \vee g) \equiv \rhd f \vee (\diamondsuit f \wedge g) \qquad \textit{DiAndFstOrEqvFstOrDiAnd}^{(C.238)}$$
$$\vdash \ \diamondsuit f \vee (\rhd f \wedge g) \equiv \diamondsuit f \qquad \textit{DiOrFstAndEqvDi}^{(C.239)}$$
$$\vdash \ \rhd(\diamondsuit f \wedge \diamondsuit g) \equiv (\rhd f \wedge \diamondsuit g) \vee (\rhd g \wedge \diamondsuit f) \qquad \textit{FstDiAndDiEqv}^{(C.240)}$$

Finally, the following two laws demonstrate that if there is no initial interval (or strict initial interval) satisfying $\rhd f$ then there is no initial interval (or strict initial interval) satisfying f. Both laws are equivalences.

$$\vdash \ \boxed{i}\,\neg \rhd f \equiv \boxed{i}\,\neg f \qquad \textit{BiNotFstEqvBiNot}^{(C.241)}$$
$$\vdash \ \boxed{s}\,\neg \rhd f \equiv \boxed{s}\,\neg f \qquad \textit{BsNotFstEqvBsNot}^{(C.242)}$$

3.8.4 Distribution

3.8.4.1 Through conjunction and disjunction

This section includes a key result in this thesis: $\mathit{LFstAndDistr}^{(C.252)}$. Essentially this law permits the distribution of chop across conjunction because the interval length on the left side has been fixed thus making the chop point deterministic. The law has four useful specialisations.

$$\vdash\ (\rhd f \wedge g_1)\ ;\ h_1 \wedge (\rhd f \wedge g_2)\ ;\ h_2 \equiv (\rhd f \wedge g_1 \wedge g_2)\ ;\ (h_1 \wedge h_2) \qquad \mathit{LFstAndDistr}^{(C.252)}$$

There are some special cases of $\mathit{LFstAndDistr}^{(C.252)}$ that are also useful laws in their own right. Each is a straightforward derivation.

- Setting $h_1 \equiv h_2$ in $\mathit{LFstAndDistr}^{(C.252)}$ generates the law:

  $$\vdash\ (\rhd f \wedge g_1)\ ;\ h \wedge (\rhd f \wedge g_2)\ ;\ h \equiv (\rhd f \wedge g_1 \wedge g_2)\ ;\ h \qquad \mathit{LFstAndDistrA}^{(C.253)}$$

- Setting $g_1 \equiv g_2$ in $\mathit{LFstAndDistr}^{(C.252)}$ generates the law:

  $$\vdash\ (\rhd f \wedge g)\ ;\ h_1 \wedge (\rhd f \wedge g)\ ;\ h_2 \equiv (\rhd f \wedge g)\ ;\ (h_1 \wedge h_2) \qquad \mathit{LFstAndDistrB}^{(C.254)}$$

- Setting $g_1 \equiv g_2 \equiv \mathsf{true}$ in $\mathit{LFstAndDistr}^{(C.252)}$ generates the law:

  $$\vdash\ \rhd f\ ;\ h_1 \wedge \rhd f\ ;\ h_2 \equiv \rhd f\ ;\ (h_1 \wedge h_2) \qquad \mathit{LFstAndDistrC}^{(C.255)}$$

  Note that it is unnecessary to specify a similar (valid) law for disjunction:

  $$\rhd f\ ;\ g \vee \rhd f\ ;\ h \equiv \rhd f\ ;\ g \vee \rhd f\ ;\ h$$

  since it is a direct application of $\mathit{ChopOrEqv}^{(C.107)}$.

- Setting $h_1 \equiv h_2 \equiv \mathsf{true}$ in $\mathit{LFstAndDistr}^{(C.252)}$, and using $\mathit{DiDef}^{(C.13)}$ generates the law:

  $$\vdash\ \diamondplus(\rhd f \wedge g_1) \wedge \diamondplus(\rhd f \wedge g_2) \equiv \diamondplus(\rhd f \wedge g_1 \wedge g_2) \qquad \mathit{LFstAndDistrD}^{(C.256)}$$

3.8.4.2 Through chop

This section introduces a number of laws that describe the behaviour of the first occurrence operator as it distributes through chop. It contains two of the most important mathematical results of this thesis. The first is $\mathit{FstFstChopEqvFstChopFst}^{(C.260)}$ whose intuitive appeal is obvious but which turned out to be remarkably difficult to prove. It expresses the fact that the sequential composition (the fusion) of two first occurrences itself denotes a first occurrence. The second important result, *FstFixFst*$^{(C.261)}$, is a corollary of *FstFstChopEqvFstChopFst*$^{(C.260)}$ and states another intuitive idea that the first occurrence of f has no other first occurrence of f as a strict prefix.

There is a particular boundary case that occurs when chop is combined with formulae on an empty interval. This results in the following theorem:[5]

$$\vdash\ f\ ;\ g \wedge \mathsf{empty} \equiv f \wedge g \wedge \mathsf{empty} \qquad \textit{ChopEmptyAndEmpty}^{(C.139)}$$

The chop operator requires a shared state. The only way that the composition $f\ ;\ g$ can occur in a *single state* is if each of f and g is true over the empty interval. A useful corollary is:

$$\vdash\ f\ ;\ \mathsf{skip} \wedge \mathsf{empty} \equiv \mathsf{false} \qquad \textit{ChopSkipAndEmptyEqvFalse}^{(C.140)}$$

which states that it is impossible for any formula of the form $f\ ;\ \mathsf{skip}$ to be satisfied over an empty interval.

Before the two particularly important results are presented the next two laws provide some necessary background. They involve the negation of chop in which one of the formulae is a first occurrence. The law *NotFstChop*$^{(C.258)}$ states that if the first occurrence of f followed by g does not hold, then *either* there is no initial interval that satisfies the first occurrence of f, *or* there is an initial interval that satisfies $\triangleright f$ but the corresponding suffix interval *does not* satisfy g.

$$\vdash\ \neg\,(\triangleright f\ ;\ g) \equiv \neg \diamondsuit\!\!\!\cdot\ \triangleright f \vee \triangleright f\ ;\ \neg\, g \qquad \textit{NotFstChop}^{(C.258)}$$

NotFstChop$^{(C.258)}$ can be compared to *NotSkipNotChop*$^{(C.129)}$, $\vdash\ \neg\,(\mathsf{skip}\ ;\ \neg\, g) \equiv \mathsf{empty} \vee (\mathsf{skip}\ ;\ g)$, and, indeed, can be considered a generalisation of it. This is interesting because combining negation and chop is difficult due to the non-determinism inherent in the chop operator. However, when the length of one of the subintervals is fixed then a useful law emerges. The aforementioned *NotSkipNotChop*$^{(C.129)}$ is such a special case and has proved to be very useful in its own right. However, it is interesting to note that this new law, *NotFstChop*$^{(C.258)}$, is a generalisation.

To illustrate the point an informal argument will be employed to show how to specialise *NotFstChop*$^{(C.258)}$ and produce *NotSkipNotChop*$^{(C.129)}$.

[5] The importance of this law emerged during informal discussions about this research with Peter Messer, previously Head of School of Computing at De Montfort Univeristy. The law is used in the proof of *FstChopEmptyEqvFstChopFstEmpty*$^{(C.232)}$ which, in turn, is used in the proof of *FstFstChopEqvFstChopFst*$^{(C.260)}$. This latter law expresses a key property of the first operator $\triangleright$.

Set $f \equiv \mathsf{skip}$, then it follows that $\neg\ (\rhd \mathsf{skip}\ ;\ g) \equiv \neg\ \diamondsuit\ \rhd \mathsf{skip} \lor \rhd \mathsf{skip}\ ;\ \neg\ g$. As will be established later, $\rhd \mathsf{skip} \equiv \mathsf{skip}$ (*FstSkip*(C.273)), so the law reduces to $\neg\ (\mathsf{skip}\ ;\ g) \equiv \neg\ \diamondsuit\ \mathsf{skip} \lor \mathsf{skip}\ ;\ \neg\ g$. Arguing informally, the first disjunct expresses the fact that no initial subinterval has two states (i.e. skip), which is more concisely written as empty. Hence: $\neg\ (\mathsf{skip}\ ;\ g) \equiv \mathsf{empty} \lor \mathsf{skip}\ ;\ \neg\ g$ which, negating g, leads to the more specific law.

The law *BsNotFstChop*(C.259) further extends *NotFstChop*(C.258) in that it requires the property to hold over all strict initial intervals. This variation is required in the proof of the main result that follows it.

$$\vdash\ \boxed{s}\ \neg\ (\rhd f\ ;\ g) \equiv \mathsf{empty} \lor \neg\ \diamondsuit\ \rhd f \lor \rhd f\ ;\ \boxed{s}\ \neg\ g \qquad \textit{BsNotFstChop}^{(C.259)}$$

The following theorem is one of the most important mathematical results in this thesis and expresses an important property about the sequential composition of first occurrences.

$$\vdash\ \rhd(\rhd f\ ;\ g) \equiv \rhd f\ ;\ \rhd g \qquad \textit{FstFstChopEqvFstChopFst}^{(C.260)}$$

The intuition is that the expression $\rhd f\ ;\ \rhd g$ requires that the first occurrence of f is followed by the first occurrence of g. This is not equivalent to the first occurrence of f followed by *any* occurrence of g because $\nvdash\ g \supset \rhd g$. However, the introduction of $\rhd$ around the whole expression, $\rhd(\rhd f\ ;\ g)$, forces the first occurrence of g.

Another of the important mathematical results in this thesis is a corollary of *FstFstChopEqvFstChopFst*(C.260):

$$\vdash\ \rhd \rhd f \equiv \rhd f \qquad \textit{FstFixFst}^{(C.261)}$$

which expresses the idea that if an interval satisfies $\rhd f$ then no strict prefix interval can satisfy $\rhd f$ – i.e. the interval not only represents the first occurrence of f but it also represents the first (and only) occurrence of $\rhd f$. Thus $\rhd f$ is a *fixpoint* of $\rhd$. This result underpins another important theorem that appears later in this thesis – the First Fixpoint Law for primary monitors, *MFixFst*(C.309).

3.8.5 First occurrence with iteration

The previous section discussed the combination of the first occurrence operator and chop. However, care must be taken when combining first occurrence with chopstar. The propositional axioms for ITL include

$$\vdash\ f^* \equiv (\mathsf{empty} \lor ((f \land \mathsf{more})\ ;\ f^*)) \qquad \textit{ChopStarEqv}^{(C.46)}$$

which specifies that an empty interval satisfies f^*[6] Therefore

$$\vdash \quad \triangleright(f^*) \equiv \mathsf{empty} \qquad \textit{FstCSEqvEmpty}^{(C.276)}$$

and, in general,

$$\triangleright(f^*) \not\equiv (\triangleright f)^*$$

Alternatively, it may be useful to write specifications including, e.g., $\triangleright(f^n)$, as this determines a specific number of iterations rather than a choice. The following law states that the finite iteration of $\triangleright f$ is itself a first occurrence.

$$\vdash \quad (\triangleright f)^n \equiv \triangleright((\triangleright f)^n), \quad [n \geq 0] \qquad \textit{FstIterFixFst}^{(C.277)}$$

One must distinguish between $(\triangleright f)^n$ and $\triangleright(f^n)$ because

$$\triangleright(f^n) \not\equiv (\triangleright f)^n$$

As an example consider the formula $f \equiv (\mathsf{fin}\ X \bmod 2 = 1 \wedge \mathsf{fin}\ X = X + 2) \vee (\mathsf{fin}\ X = 2^X)$ and the interval σ in which $\sigma_0(X) = 1, \sigma_1(X) = 2, \sigma_2(X) = 3, \ldots$.

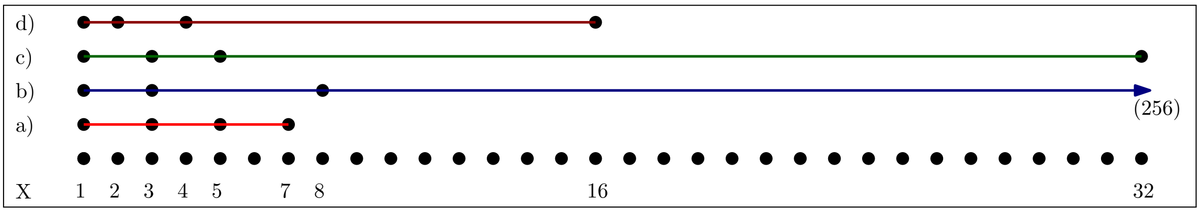


Figure 3.10: Comparing the four initial intervals satisfying f^3

In Figure 3.10 the four prefix intervals satisfying f^3 have been labelled $a) \ldots d)$. The interval marked $a)$ represents $\triangleright(f^3)$: the shortest initial interval satisfying f^3. The interval marked $d)$ represents $(\triangleright f)^3$: at each step the shortest interval satisfying f is taken. This is the shortest path $(\triangleright((\triangleright f)^3))$ that can be generated in this way (see *FstIterFixFst*$^{(C.277)}$). The intervals $b)$ and $c)$ represent neither $\triangleright(f^3)$ nor $(\triangleright f)^3$ but do represent other ways of satisfying f^3.

3.8.6 First and halt

Both $\mathsf{halt}\ w$ and $\triangleright f$ define a fixed-length interval. The difference is that halt is used with a state formula w whereas $\triangleright$ can be used with a temporal formula. Consider the semantics of

[6] The empty interval even satisfies false^*.

these operators:

$$\mathcal{F}[\![\rhd f]\!](\sigma) = \mathsf{tt} \text{ iff } \mathcal{F}[\![f]\!](\sigma) = \mathsf{tt} \text{ and for all } 0 \leq i < |\sigma|,\ \mathcal{F}[\![\neg f]\!](\sigma_0 \mathinner{..} \sigma_i) = \mathsf{tt}$$

$$\mathcal{F}[\![\mathsf{halt}\, f]\!](\sigma) = \mathsf{tt} \text{ iff } \mathcal{F}[\![f]\!](\langle \sigma_{|\sigma|} \rangle) = \mathsf{tt} \text{ and for all } 0 \leq i < |\sigma|,\ \mathcal{F}[\![\neg f]\!](\sigma_i \mathinner{..} \sigma_{|\sigma|}) = \mathsf{tt}$$

For example, the formula $\mathsf{halt}\, P$ holds over an interval in which P is true in the final state and not in any previous states. $\mathsf{halt}\, P$ holds over an empty interval if P does. The following laws each express $\mathsf{halt}\, w$ in terms of $\rhd$:

$$\vdash\ \mathsf{halt}\, w \equiv \rhd(\mathsf{fin}\, w) \qquad \mathit{HaltStateEqvFstFinState}^{(C.246)}$$

$$\vdash\ \mathsf{halt}\, w \equiv \rhd(\mathsf{halt}\, w) \qquad \mathit{HaltStateEqvFstHaltState}^{(C.247)}$$

$$\vdash\ \rhd(\Diamond w) \equiv \mathsf{halt}\, w \qquad \mathit{FstDiamondStateEqvHalt}^{(C.248)}$$

3.9 The *last occurrence* operator

A reflected first occurrence operator $\lhd f$ could be considered to represent the *most recent* interval to satisfy f.

$$\lhd f \quad \widehat{=} \quad f \wedge \boxed{\text{t}}\, \neg f \qquad \mathit{LstDef}^{(C.282)}$$

It is possible to determine a relationship between $\rhd$ and $\lhd$ as follows:

$$\begin{aligned}
& (\rhd f)^r \\
& \equiv (f \wedge \boxed{\text{s}}\, \neg f)^r && \mathit{FstDef}^{(C.222)} \\
& \equiv f^r \wedge (\boxed{\text{s}}\, \neg f)^r && \mathit{TRAnd}^{(C.61)} \\
& \equiv f^r \wedge \boxed{\text{t}}\, (\neg f)^r && \mathit{BsrEqvBtr}^{(C.217)} \\
& \equiv f^r \wedge \boxed{\text{t}}\, \neg f^r && \mathit{TRNot}^{(C.57)} \\
& \equiv \lhd f^r && \mathit{LstDef}^{(C.282)}
\end{aligned}$$

And symmetrically:

$$\begin{aligned}
& (\lhd f)^r \\
& \equiv (f \wedge \boxed{\text{t}}\, \neg f)^r && \mathit{LstDef}^{(C.282)} \\
& \equiv f^r \wedge (\boxed{\text{t}}\, \neg f)^r && \mathit{TRAnd}^{(C.61)} \\
& \equiv f^r \wedge \boxed{\text{s}}\, (\neg f)^r && \mathit{BtrEqvBsr}^{(C.219)} \\
& \equiv f^r \wedge \boxed{\text{s}}\, \neg f^r && \mathit{TRNot}^{(C.57)} \\
& \equiv \rhd f^r && \mathit{FstDef}^{(C.222)}
\end{aligned}$$

Thus the following laws hold:

$\vdash\ (\rhd f)^r \equiv \lhd f^r$ *FstrEqvLstr*(C.284)

$\vdash\ (\lhd f)^r \equiv \rhd f^r$ *LstrEqvFstr*(C.285)

A straightforward corollary can be obtained using *FstrEqvLstr*(C.284) and *TRChop*(C.59):

$\vdash\ (\rhd f\ ;\ \rhd g)^r \equiv \lhd g^r\ ;\ \lhd f^r$ *FstChopFstREqvLstrChopLstr*(C.286)

Recall that one of the most important results in this thesis, *FstFstChopEqvFstChopFst*(C.260), proved that the sequential composition of two first occurrence expressions is itself a first occurrence: $\vdash\ \rhd(\rhd f\ ;\ g) \equiv \rhd f\ ;\ \rhd g$. It is expected that the reflected law would also hold: i.e. that $\vdash\ \lhd(g\ ;\ \lhd f) \equiv \lhd g\ ;\ \lhd f$. The following argument can be used:

$\lhd g^r\ ;\ \lhd f^r$

$\equiv (\rhd f\ ;\ \rhd g)^r$ *FstChopFstREqvLstrChopLstr*(C.286)

$\equiv (\rhd(\rhd f\ ;\ g))^r$ *FstFstChopEqvFstChopFst*(C.260)

$\equiv \lhd(\rhd f\ ;\ g)^r$ *FstrEqvLstr*(C.284)

$\equiv \lhd(g^r\ ;(\rhd f)^r)$ *TRChop*(C.59)

$\equiv \lhd(g^r\ ;\ \lhd f^r)$ *FstrEqvLstr*(C.284)

Thus, relabelling, the following holds:

$\vdash\ \lhd(g\ ;\ \lhd f) \equiv \lhd g\ ;\ \lhd f$ *LstChopLstEqvLstChopLst*(C.288)

This final section has shown how the reflected theory could be developed and future work will investigate these relationships in more detail.

3.9.1 Last and until

The LTL operator $\mathcal{U}$ is not provided as part of the standard library of derived operators in ITL [CM16]. However, a definition of $\mathcal{U}$ for finite intervals was provided by Moszkowski [Mos83]:

Equation 3.9.1 $f_1\,\mathcal{U}\,f_2 \ \widehat{=}\ \exists P \bullet (P \wedge \Box(P \supset (f_2 \vee (f_1 \wedge \bigcirc P))))$
where P does not occur free in f_1 or f_2.

The ITL operator $\mathcal{U}$ is illustrated in Figure 3.11.

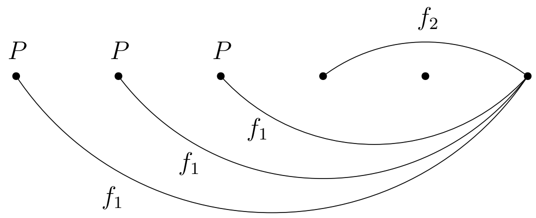


Figure 3.11: Moszkowski's definition of $f_1 \,\mathcal{U}\, f_2$ in ITL

$f_1 \mathcal{U} f_2$ is defined over a finite interval such that for some $0 \le k \le |\sigma|$, $\sigma_0 \mathinner{..} \sigma_{|\sigma|} \models f_1,\ \sigma_1 \mathinner{..} \sigma_{|\sigma|} \models f_1,\ \ldots,\ \sigma_{k-1} \mathinner{..} \sigma_{|\sigma|} \models f_1$ and $\sigma_k \mathinner{..} \sigma_{|\sigma|} \models f_2$. There may be more than one value of k that satisfies $f_1 \,\mathcal{U}\, f_2$. However, it is possible to specify that k is maximal, and therefore that f_1 holds up to the last occurrence of f_2, by writing $f_1 \mathcal{U} (\triangleleft f_2)$. Thus the reflected first occurrence operator can be used to construct a deterministic until formula.

3.10 Summary

This chapter introduced the background to the first occurrence operator. An investigation into fixed length intervals within ITL discovered a collection of useful laws. Three new ITL operators, $\boxed{s}$ and $\diamond s$, and $\triangleright$ were introduced. A significant number of laws relating these new operators to each other and to existing ITL operators was developed. These laws provide the basis of ITL-Monitor, the runtime monitor system introduced in Chapter 4.

Two of the most important mathematical results in this thesis have been presented: the laws *FstFstChopEqvFstChopFst*$^{(C.260)}$ and *FstFixFst*$^{(C.261)}$. These laws capture fundamental, intuitive properties of $\triangleright$ upon which the later theory relies. A discussion pointing to future work in which the theory can be developed using reflection was presented.

Chapter 4

ITL Monitor

ITL-Monitor is a restricted subset of ITL used to construct specifications whose components describe a deterministic partitioning of an execution trace. This was illustrated previously in Section 3.3.2. Furthermore, this deterministic partitioning property is preserved under monitor composition – this is a key property of ITL-Monitor which is based upon the fact that every ITL-Monitor represents its own first occurrence.

ITL-Monitor has also been realised as a DSL (cf. 2.4.1) in Scala. As code, a monitor performs the rôle depicted in Figure 2.15 (page 32), receiving states from a running program, and maintaining an internal representation of the execution trace. At each deterministic fusion point the monitor assesses whether or not the current trace satisfies its ITL formula. The implementation of ITL-Monitor in Scala is presented in the following chapter.

In this chapter, the syntax of ITL-Monitor expressions is introduced along with a translation function to their respective ITL formulae. The behaviour of each monitor operator is explained with an emphasis on their practical rôle in runtime verification. In Section 4.4 the algebraic properties of ITL-Monitor are presented. The analysis of these properties formed another major aspect of the current work and resulted in a comprehensive list of monitor laws and associated proofs. All of these have been mechanically checked by Isabelle/HOL and appear as Chapter 8 of [CMS19].

The chapter concludes with a small example specification.

4.1 Monitor syntax and translation to ITL

The syntax of a monitor expression, ITL-Monitor, is given in Figure 4.1.

$$\begin{aligned}\text{ITL-Monitor} ::= \;& \textbf{FIRST}\,(\text{ITL-Formula})\\ \mid\;& \text{ITL-Monitor}\ \ \textbf{UPTO}\ \ \text{ITL-Monitor}\\ \mid\;& \text{ITL-Monitor}\ \ \textbf{THRU}\ \ \text{ITL-Monitor}\\ \mid\;& \text{ITL-Monitor}\ \ \textbf{THEN}\ \ \text{ITL-Monitor}\\ \mid\;& \text{ITL-Monitor}\ \ \textbf{WITH}\ \ \text{ITL-Formula}\end{aligned}$$

where ITL-Formula represents any well-formed ITL formula.

Figure 4.1: ITL-Monitor syntax

The unary operator **FIRST** has the highest priority. Each of the binary operators has equal priority and is left-associative. Parentheses can be used to override these defaults. The ITL formula represented by each ITL-Monitor expression is defined by the translation function $\mathcal{M}$: ITL-Monitor $\rightarrow$ ITL-Formula (Figure 4.2).

$\mathcal{M}(\textbf{FIRST}\,(f))$	$\mathrel{\widehat=}$	$\triangleright f$	*MFirstDef*$^{(C.289)}$
$\mathcal{M}(a\ \textbf{UPTO}\ b)$	$\mathrel{\widehat=}$	$\triangleright(\mathcal{M}(a) \vee \mathcal{M}(b))$	*MUptoDef*$^{(C.290)}$
$\mathcal{M}(a\ \textbf{THRU}\ b)$	$\mathrel{\widehat=}$	$\triangleright(\diamondplus\mathcal{M}(a) \wedge \diamondplus\mathcal{M}(b))$	*MThruDef*$^{(C.291)}$
$\mathcal{M}(a\ \textbf{THEN}\ b)$	$\mathrel{\widehat=}$	$\mathcal{M}(a)\ ;\ \mathcal{M}(b)$	*MThenDef*$^{(C.292)}$
$\mathcal{M}(a\ \textbf{WITH}\ f)$	$\mathrel{\widehat=}$	$\mathcal{M}(a) \wedge f$	*MWithDef*$^{(C.293)}$

Figure 4.2: ITL-Monitor translations to ITL formulae

A description of each of the ITL-Monitor operators is given below. (Note that all monitors are first occurrences: $\vdash\ \mathcal{M}(a) \equiv \triangleright\mathcal{M}(a)$ (*MFixFst*$^{(C.309)}$). This law will be presented formally in Section 4.4.1).

FIRST (f)

This monitor succeeds as soon as the states consumed comprise an interval that satisfies $\triangleright f$. This basic monitor is used to define the extent of the subintervals into which an execution trace is divided.

a **UPTO** b

If a specification can be expressed as the first occurrence of an interval that satisfies either one of two independent formulae, $\mathcal{M}(a)$ or $\mathcal{M}(b)$, then a **UPTO** b permits each to be run independently and both terminated as soon as one is satisfied.

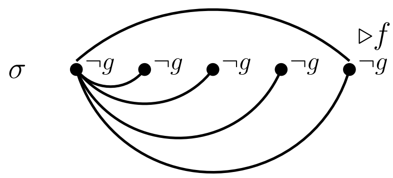


Figure 4.3: **FIRST** (f) **UPTO FIRST** (g)

Figure 4.3 illustrates a situation in which the expression **FIRST** (f) **UPTO FIRST** (g) is satisfied with formula f occuring before formula g.

a **THRU** b

This monitor specifies the shortest interval containing prefixes in which both $\mathcal{M}(a)$ and $\mathcal{M}(b)$ are satisfied. Necessarily, this means that a satisfying interval is a model for the first occurrence of either (or both) of these formulae. The monitor terminates as soon as either a or b has terminated: this ensures that the fixed-length property is maintained. Furthermore, the monitor fails if either of its two component monitors, a or b, fails.

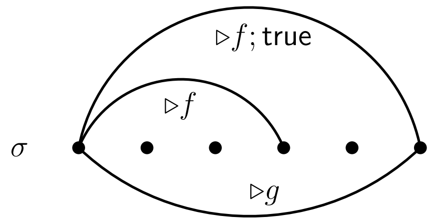


Figure 4.4: **FIRST** (f) **THRU FIRST** (g)

Figure 4.4 illustrates an interval that satisfies **FIRST** (f) **THRU FIRST** (g). The interval satisfies $\triangleright g$ and a prefix interval satisfies $\triangleright f$: i.e. the example interval satisfies $\Diamond\!\!\!\!\cdot\ \triangleright f \wedge \triangleright g$.

a **THEN** b

The ITL formula represented by $\mathcal{M}(a$ **THEN** $b)$ is the sequential composition of two first occurrences, $\mathcal{M}(a)$; $\mathcal{M}(b)$. The resulting formula is itself a first occurrence:

$$
\begin{aligned}
& \mathcal{M}(a) \ ; \ \mathcal{M}(b) & \\
& \equiv \triangleright \mathcal{M}(a) \ ; \ \triangleright \triangleright \mathcal{M}(b) & \quad MFixFst^{(C.309)},\ FstFixFst^{(C.261)} \\
& \equiv \triangleright(\triangleright \mathcal{M}(a) \ ; \ \triangleright \mathcal{M}(b)) & \quad FstFstChopEqvFstChopFst^{(C.260)} \\
& \equiv \triangleright(\mathcal{M}(a) \ ; \ \mathcal{M}(b)) & \quad MFixFst^{(C.309)}
\end{aligned}
$$

Since $\mathcal{M}(a)$ is a first-occurrence ($MFixFst^{(C.309)}$), this ensures that the monitor composition $\mathcal{M}(a \textsf{ THEN } b)$ defines a unique point of fusion within the execution trace.

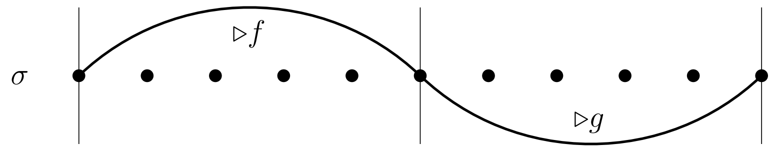


Figure 4.5: $\textsf{FIRST}\,(f)\ \textsf{THEN FIRST}\,(g)$

Figure 4.5 illustrates the sequential composition of two monitors $\textsf{FIRST}\,(f)$ $\textsf{THEN}$ $\textsf{FIRST}\,(g)$. To succeed both components must succeed: $\triangleright f$ up to the uniquely determined fusion point, and then $\triangleright g$ to the end of the interval.

a **WITH** f

The conjoinment of $\mathcal{M}(a)$ and f. This monitor consumes sufficient states to satisfy $\mathcal{M}(a)$ and then checks that the formula f, which can be any ITL formula, holds over the same interval. Mathematically, this is the same monitor as $\textsf{FIRST}\,(\mathcal{M}(a) \wedge f)$

$$\begin{aligned}
&\mathcal{M}(\textsf{FIRST}\,(\mathcal{M}(a) \wedge f)) && \\
&\equiv \triangleright(\mathcal{M}(a) \wedge f) && MFirstDef^{(C.289)} \\
&\equiv \triangleright(\triangleright \mathcal{M}(a) \wedge f) && MFixFst^{(C.309)} \\
&\equiv \triangleright \mathcal{M}(a) \wedge f && FstFstAndEqvFstAnd^{(C.225)} \\
&\equiv \mathcal{M}(a) \wedge f && MFixFst^{(C.309)} \\
&\equiv \mathcal{M}(a \textsf{ WITH } f) && MWithDef^{(C.293)}
\end{aligned}$$

By separating the components of the formula using a **WITH** f, it is possible for an implementation to separate the search for a suitable interval ($\triangleright \mathcal{M}(a)$ must be checked with the addition of each new state) from the final verification of f – a check that only needs to be performed once.

A number of derived monitors are defined in Figure 4.6 below. These capture common specification patterns. The unary operators take precedence over the binary operators. Each of the latter has equal priority and is left-associative. Parentheses can be used to override these defaults.

$$\mathsf{LEN}\,(k) \;\widehat{=}\; \mathsf{FIRST}\,(\mathsf{len}(k)) \qquad MLenDef^{(C.295)}$$

$$\mathsf{SKIP} \;\widehat{=}\; \mathsf{FIRST}\,(\mathsf{skip}) \qquad MSkipDef^{(C.297)}$$

$$\mathsf{FAIL} \;\widehat{=}\; \mathsf{FIRST}\,(\mathsf{false}) \qquad MFailDef^{(C.300)}$$

$$\mathsf{EMPTY} \;\widehat{=}\; \mathsf{FIRST}\,(\mathsf{empty}) \qquad MEmptyDef^{(C.296)}$$

$$\mathsf{HALT}\,(w) \;\widehat{=}\; \mathsf{FIRST}\,(\mathsf{fin}\ w) \qquad MHaltDef^{(C.294)}$$

$$\begin{array}{lcl} a\ \mathsf{TIMES}\ 0 & \widehat{=} & \mathsf{EMPTY} \\ a\ \mathsf{TIMES}\ (k+1) & \widehat{=} & a\ \mathsf{THEN}\ (a\ \mathsf{TIMES}\ k), \quad k \geq 0 \end{array} \qquad MTimesDef^{(C.299)}$$

$$\mathsf{GUARD}\,(w) \;\widehat{=}\; \mathsf{EMPTY}\ \mathsf{WITH}\ w \qquad MGuardDef^{(C.298)}$$

$$w_1\ \mathsf{UNTIL}\ w_2 \;\widehat{=}\; (\mathsf{HALT}\ w_2)\ \mathsf{WITH}\ (\boxed{\mathrm{m}}\ w_1) \qquad MUntilDef^{(C.303)}$$

$$a\ \mathsf{ALWAYS}\ w \;\widehat{=}\; a\ \mathsf{WITH}\ (\boxed{\mathrm{i}}\ \mathsf{fin}\ w) \qquad MAlwaysDef^{(C.301)}$$

$$a\ \mathsf{SOMETIME}\ w \;\widehat{=}\; a\ \mathsf{WITH}\ (\diamondsuit\ \mathsf{fin}\ w) \qquad MSometimeDef^{(C.302)}$$

$$a\ \mathsf{WITHIN}\ f \;\widehat{=}\; a\ \mathsf{WITH}\ (\boxed{\mathrm{s}}\ \neg\ f) \qquad MWithinDef^{(C.304)}$$

$$a\ \mathsf{AND}\ b \;\widehat{=}\; a\ \mathsf{WITH}\ \mathcal{M}(b) \qquad MAndDef^{(C.305)}$$

$$a\ \mathsf{ITERATE}\ b \;\widehat{=}\; a\ \mathsf{WITH}\ (\mathcal{M}(b))^* \qquad MIterateDef^{(C.306)}$$

Figure 4.6: Derived monitors

These derived monitors are useful in three respects.

- To express useful zero and unit elements in the algebra: **EMPTY** and **FAIL**.
- To improve readability: e.g. **LEN** , **HALT** and **GUARD** .
- To improve efficiency. This includes the group of monitors defined using **WITH**. The discussion of **WITH** on page 74 showed how the right-hand operand only needs to be evaluated once. The specific behaviours of each of these monitors have been used to provide more efficient code implementations.

Each of the derived monitors is described below.

LEN (k) and **SKIP**. These monitors are satisfied by any intervals of the required length. Recall that skip $\equiv$ len(1). For example, a **THEN** b shares a common state, whereas in a **THEN SKIP THEN** b, a and b have no common state.[1] The difference is demonstrated by Figure 4.7.

[1] **THEN** is an associative operator ($MThenAssoc^{(C.355)}$).

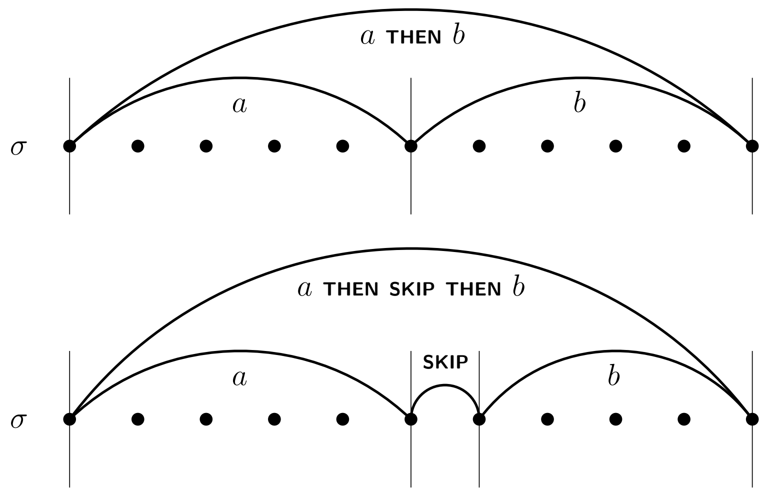


Figure 4.7: The effect of introducing a **SKIP** monitor

FAIL and **EMPTY**. These monitors represent annihilators and units when combined with various binary monitor operators. As such they perform an important rôle in the ITL-Monitor algebra which is presented below in Section 4.4.

HALT (w). This is a special case of **FIRST** in which an interval is defined up to the first occurrence of state formula w, or, in ITL, $\boxed{\text{m}} \neg w \wedge \mathsf{fin}\, w$. The relationship between first occurrence, $\triangleright$, and halt was discussed in Section 3.8.6. The **HALT** monitor can be used to define a finite subinterval up to the first occurrence of a state event – for example, **HALT** *Button_Pressed*. However, **FIRST** can be used to define a subinterval based upon temporal relationships – for example, **FIRST** $(\Diamond((X \leftarrow X) \wedge \mathsf{more}))$: "*Up to the first point at which the state variable X receives a value that it was assigned previously.*" The addition of more ensures that progress is made because $X \leftarrow X$ is trivially satisfied by an empty interval.

a **TIMES** k. This monitor represents a specific number of instances of a fused together. An example showing a **TIMES** 8 is shown in Figure 4.8.

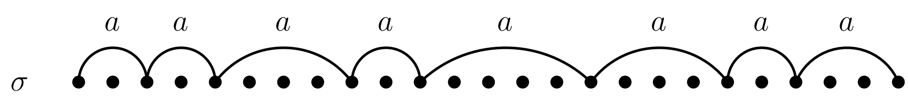


Figure 4.8: a **TIMES** 8

If $k = 1$ then this is equivalent to a, alone; and if $k = 0$ then this is equivalent to **EMPTY**.

GUARD (w). This monitor treats the current state as an empty interval and establishes whether or not w holds in this empty interval. A guard can be useful for specifying an intial condition, e.g. **GUARD** $(X = 0)$ **THEN** a, or for analysing the final state of an interval, e.g. a **THEN GUARD** $(X = 0)$. A series of guards can also be used to determine future behaviour as described in Section 4.3.

w_1 **UNTIL** w_2. This monitor specifies the *shortest* interval that satisfies $w_1 \, \mathcal{U} \, w_2$. This is equivalent to $\triangleright(\boxed{m}\ w_1\ ;\ w_2)$.

The remaining derived monitors are special cases of a **WITH** f. Each monitor can be implemented more efficiently by exploiting the properties related to its specific function.

a **ALWAYS** w. w can be checked in each state. If ever $\neg\, w$ holds then the whole monitor fails immediately.

a **SOMETIME** w. w can be checked in each state. If w holds in the final state of some prefix interval $\sigma_0 \mathinner{.\,.} \sigma_i$, then the property holds for any extended interval $\sigma_0 \mathinner{.\,.} \sigma_j$ where $0 \leq i \leq j$.

a **WITHIN** f. As each new state is supplied, f is checked against each newly-extended prefix interval. If f should ever be satisfied, but $\mathcal{M}(a)$ is not satisfied, then the monitor fails.

The monitor **FAIL** is equal to **FIRST** (false) **WITHIN EMPTY**.

a **AND** b. The monitors a and b must be satisfied simultaneously by the same interval. It is possible for an implementation to run these monitors simultaneously, failing if, and as soon as, either monitor fails. Figure 4.9 illustrates **FIRST** (f) **AND FIRST** (g).

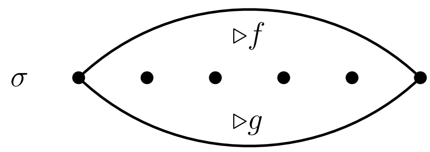


Figure 4.9: **FIRST** (f) **AND FIRST** (g)

a **ITERATE** b. This monitor combines a with a repetition of b. This is similar to **AND** above in that for a satisfying interval both $\mathcal{M}(a)$ and $(\mathcal{M}(b))^*$ must hold. Figure 4.10 provides an illustrative example.

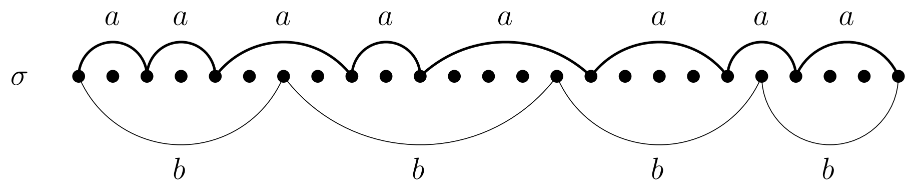


Figure 4.10: $(a$ **TIMES** $8)$ **ITERATE** b

Example 4.1.1 *Partition.* Consider the ITL-Monitor m where

$m = a$ **ITERATE** b
$b\ = $ **FIRST** $(X \lessapprox X+1)$ **WITH** f

The ITL formula represented by $\mathcal{M}(b)$ is $(X \lessapprox X+1) \wedge f$. The first three cycles of $(\mathcal{M}(b))^*$ are illustrated in Figure 4.11.

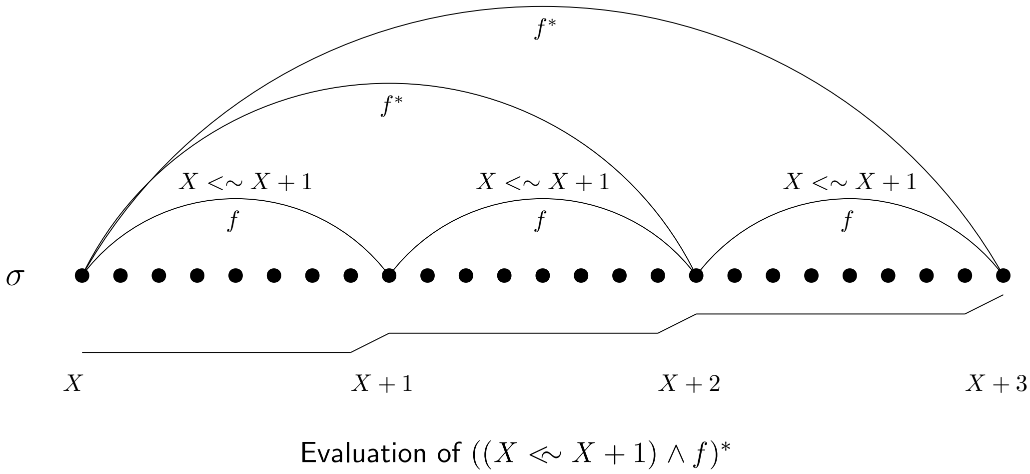


Figure 4.11: Partitioning using a counter

This demonstrates a partitioning of an evolving system using one of the state variables as a counter. The checkout example (Section 6.3) uses a transaction counter to partition the execution trace by customer transaction. The example can be generalised by writing, e.g.

$m = a$ **ITERATE** b
$b\ = $ **FIRST** $(X \lessapprox X+1)$ **ITERATE** c

This demonstrates that levels of partitioning can be nested. ■

4.2 Importable assumptions and exportable commitments

Moszkowski demonstrated how the use of temporal fixpoints could be used to reason compositionally in ITL [Mos94, Mos96a, Mos98]. The original work was applied primarily to safety and liveness conditions. However, it also provided a general framework for reasoning about ITL specifications compositionally. The application of this theory within ITL-Monitor is explored in this section.

4.2.1 Background

Recall that sequential composition connects formulae using ITL's chop operator, $\sigma \models f_1 \;;\; f_2$ iff exists k s.t. $\sigma_0..\sigma_k \models f_1$ and $\sigma_k..\sigma_{|\sigma|} \models f_2$. Parallel composition relates to formulae composed with $\wedge$ requiring both conjuncts to be satisfied simultaneously: $\sigma \models f_1 \wedge f_2$ iff $\sigma \models f_1$ and $\sigma \models f_2$.

Consider the sequential composition, $f_1 \;;\; f_2$. Suppose that $f_1 \supset Co$, $f_2 \supset Co$, and $Co^* \equiv Co$. In this case if $f_1 \;;\; f_2$ holds for some interval, then so does $Co \;;\; Co$ and hence Co^* which is equivalent to Co. A formula Co with the property that $Co^* \equiv Co$ is an *exportable commitment*.

Furthermore, consider some property As that holds over an interval and also satisfies $As \equiv \boxed{a}\, As$. In this case As holds over *every* subinterval. Such a formula is an *importable assumption*.

It is possible for a formula to be simultaneously an importable assumption and an exportable commitment. An example of such a formula is $\mathsf{keep}\, f$. Such formulae are referred to as *very compositional*. Moszkowski [Mos96a] considers the following general form of an ITL theorem:

$$\vdash w \wedge As \wedge Sys \supset Co \wedge \mathsf{fin}\, w'$$

in which w is an initial state formula, As is an assumption about the *overall* interval, Sys is the system under consideration, Co is a commitment about the *overall* interval, and w' is a final state formula. Thus, an interval that satisfies the formula $w \wedge As \wedge Sys$ also satisfies the commitment Co and final state w'. The composition of two systems ($Sys \;;\; Sys'$) with suitable importable assumption As and exportable commitment Co is summarised in the following proof rule 4.1 [Mos96a].

Rule 4.1

$$\frac{\begin{array}{l}\vdash w \wedge As \wedge Sys \supset Co \wedge \mathsf{fin}\, w' \\ \vdash w' \wedge As \wedge Sys' \supset Co \wedge \mathsf{fin}\, w''\end{array}}{\vdash w \wedge As \wedge (Sys \;;\; Sys') \supset Co \wedge \mathsf{fin}\, w''}$$

For this rule to be sound the following conditions must be satisfied:[2]

Importable assumption	$As \equiv \boxed{a}\ As$	(1)
Exportable commitment	$Co \equiv Co^{\star}$	(2)

Note that $(Sys\ ;\ Sys')$ is satisfied by the whole interval but the fusion point is non-deterministic. Therefore, if the global assumption As is to apply to an arbitrary subinterval then it must be a fixpoint of $\boxed{a}$ (all subintervals) (condition (1)). Furthermore, if both Sys and Sys' imply Co then the whole interval satisfies $(Co\ ;\ Co)$, and hence Co (condition (2)).

Sequential composition can be generalised to handle zero or more iterations, (Sys^{*}). Once again, (4.2) is taken from [Mos96a]. The soundness conditions (1) and (2) apply.

Rule 4.2

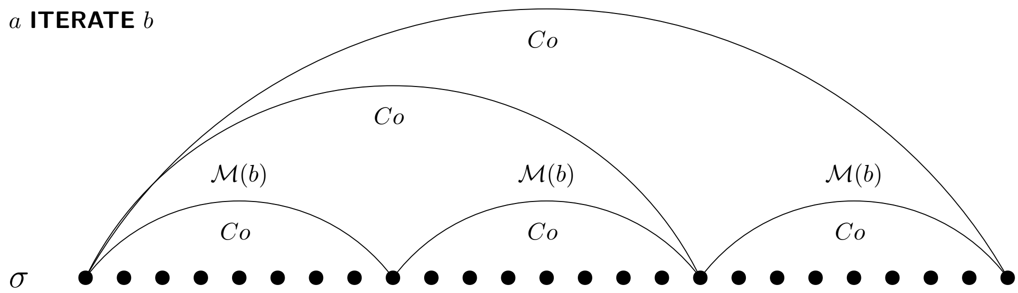
$$\frac{\vdash w \wedge As \wedge Sys \supset Co \wedge \mathsf{fin}\ w}{\vdash w \wedge As \wedge Sys^{\star} \supset Co \wedge \mathsf{fin}\ w}$$

Both proof rules 4.1 and 4.2 have simpler counterparts that may be generated by setting various components to true. This technique can be used to remove the initial and final states (w and w'), the assumptions As, or the commitments Co, or any combination of these as required.

A collection of formulae that are fixpoints of chopstar and $\boxed{a}$ – and hence can be used for exportable commitments and importable assumptions – are contained in [Mos96c].

Consider an ITL-Monitor, a **ITERATE** b, that is monitoring the property $\mathcal{M}(a) \wedge (\mathcal{M}(b))^{*}$. If $\mathcal{M}(b)$ implies some exportable commitment Co, then each successful iteration of b maintains Co^{*} which, in turn, establishes Co over the whole interval. This is illustrated in Figure 4.12 and demonstrates how an exportable commitment Co can be verified incrementally.

[2] These conditions are expressed as equivalences. However, to establish them one only needs to show that $Co^{*} \supset Co$ and $As \supset \boxed{a}\ As$ because their converses are laws.



Within the monitor a **ITERATE** b, the image shows three cycles of b. If $\mathcal{M}(b) \supset Co$ then Co^* is established at the end of each cycle and, if Co is an exportable commitment (i.e. $Co \equiv Co^*$), then Co holds over the whole interval.

Figure 4.12: Exportable commitment

4.2.2 Examples

Example 4.2.1 *Invariant.* Consider the ITL-Monitor m where

$$m = a \textbf{ ITERATE } (\textbf{FIRST } (f) \textbf{ WITH } (A \leftarrow A))$$

The monitor verifies an execution trace against the ITL formula $\mathcal{M}(a) \wedge (\rhd(f) \wedge (A \leftarrow A))^*$. The subformula $A \leftarrow A$ specifies that the value of A in the final state of the subinterval equals its value in the initial state of the subinterval. The formula $A \leftarrow A$ is also a fixpoint of chopstar, and hence an exportable commitment. Therefore, since $\mathcal{M}(m) \supset (A \leftarrow A)^*$, and $(A \leftarrow A)^* \equiv (A \leftarrow A)$, each iteration re-establishes $A \leftarrow A$ and, when the loop terminates, this invariant property will also hold over the whole interval. ■

Example 4.2.2 *Refactor.* Consider the ITL-Monitor m where

$$m = a \textbf{ ITERATE } (\textbf{FIRST } (f) \textbf{ ALWAYS } (w)) \textbf{ ITERATE } (\textbf{FIRST } (g) \textbf{ ALWAYS } (w))$$

which monitors the ITL formula $\mathcal{M}(a) \wedge (\rhd f \wedge \Box w)^* \wedge (\rhd g \wedge \Box w)^*$. Suppose that this monitor is determining an exportable commitment Co where

$$\begin{array}{ll} Co & \equiv Co_f \wedge Co_g \\ \Box w \wedge f \supset Co_f & \\ \Box w \wedge g \supset Co_g & \end{array}$$

The monitor (inefficiently) duplicates the evaluation of the subexpression **ALWAYS** (w). In general, one cannot refactor $(\rhd f \wedge h)^* \wedge (\rhd g \wedge h)^*$ to $(\rhd f)^* \wedge (\rhd g)^* \wedge h$. However, the formula $\Box w$ is a fixpoint of $\boxed{a}$ and hence an importable assumption. Therefore, it is possible to establish separately that $\Box w$ holds over the whole interval, and import it into all of the subintervals specified by the individual $\triangleright f$ and $\triangleright g$ subformulae.

$$m = a \ \textsf{\textbf{ALWAYS}}\ (w)\ \textsf{\textbf{ITERATE FIRST}}\ (f)\ \textsf{\textbf{ITERATE FIRST}}\ (g)$$

Thus **ALWAYS** (w) runs concurrently with the two parallel iterations. If w is discovered not to hold in any state then the whole monitor fails immediately. (See discussion on **ALWAYS** (page 77). ■

4.3 Selection

The monitor a **UPTO** b is satisfied by an interval provided that the interval satisfies either a or b (or both). Both monitors analyse the evolving interval until either $\mathcal{M}(a)$ or $\mathcal{M}(b)$ holds, at which point the composition succeeds. This concept can be specialised so that the choice between a and b is based upon the first state of the interval. Such a situation arises at the point of fusion between two monitors a **THEN** b. The final state of a becomes the initial state of b thus providing the possibility of communication between the two subintervals. The continuation branch can be selected following the application of a single-state monitor that involves a state expression, w. For example, consider the following pattern:

$$\begin{array}{lllll} (a\ \textsf{\textbf{UPTO}}\ (\textsf{\textbf{HALT}}\ (w)))\ \textsf{\textbf{THEN}}\ (& (\textsf{\textbf{GUARD}}\ (w) & \textsf{\textbf{THEN}} & b_0) & \textsf{\textbf{UPTO}} \\ & (\textsf{\textbf{GUARD}}\ (\neg\ w) & \textsf{\textbf{THEN}} & b_1) & \\ ) & & & & \end{array} \qquad \text{(S1)}$$

The monitor expression $(a$ **UPTO** $($**HALT** $(w)))$ is satisfied by an interval over which either $\mathcal{M}(a)$ holds, or $\mathsf{halt}(w)$ holds. Suppose that $\mathcal{M}(a) \wedge \Box(\neg\ w)$ represents normal termination and $\mathsf{halt}(w)$ represents some exceptional condition. The **GUARD**s can be used to analyse the value of this formula and determine which subsequent monitor (b_0 or b_1) to evaluate. Both branches of **UPTO** will be evaluated simultaneously and will continue until one of the branches succeeds. However, if the guards are mutually exclusive, as is the case in this example, then one of these guarded expressions will be eliminated immediately.

This pattern can be extended by using an integer flag, I say, to denote normal termination $(I = 0)$ or an error code $(I > 0)$. In general, if $I \in \{0 \ldots M\}$ then each one of M different interrupts might be matched.

$$
\begin{array}{llllll}
a\ \textsf{UPTO}\ (\textsf{FIRST}\ (\mathsf{fin}\ I > 0))\ \textsf{THEN}\ (& (\textsf{GUARD}\ (I = 0) & \textsf{THEN} & b_0) & \textsf{UPTO} \\
 & (\textsf{GUARD}\ (I = 1) & \textsf{THEN} & b_1) & \textsf{UPTO} \\
 & \vdots & & & \\
 & (\textsf{GUARD}\ (I = M) & \textsf{THEN} & b_M) & \\
) & & & &
\end{array}
\tag{S2}
$$

All of the guards are checked in parallel. In this case all except one will fail and the monitor will quickly reduce to one of the $b_i, i \in \{0 \dots M\}$.

If no guard succeeds then the monitor fails. The style of selection described by this pattern is reminiscent of Dijkstra's Guarded Command Language [Dij75].

4.4 Algebraic properties of monitors

The ITL-Monitor operators respect a number of algebraic laws which are presented in this section. The calculus permits the specification/monitor designer to compose and refactor ITL-Monitor specifications whilst maintaining equivalent ITL formulae for analysis. All of the laws in this Chapter have been developed as part of this thesis and appear in Chapter 7 of [CMS19].

4.4.1 First occurrence fixpoint law

Primary monitors satisfy a fixpoint law: $\mathcal{M}(a)$ is a *fixpoint* of $\triangleright$:

$$\vdash\ \mathcal{M}(a) \equiv \triangleright\, \mathcal{M}(a) \qquad \mathit{MFixFst}^{(C.309)}$$

This law states that the formula represented by any monitor is a first-occurrence formula. Therefore, every monitor operator preserves this property. This result is used extensively throughout the proofs of many of the algebraic monitor laws.

4.4.2 Equivalence of monitors

The following equivalence relation[3] facilitates a succinct expression of the algebraic monitor laws.

$$(a \simeq b) \equiv (\vdash\ \mathcal{M}(a) = \mathcal{M}(b)) \qquad \mathit{EqDef}^{(C.324)}$$

$$a \simeq a \qquad \mathit{MonEqRefl}^{(C.325)}$$

$$a \simeq b \vdash b \simeq a \qquad \mathit{MonEqSym}^{(C.326)}$$

[3]Defined by A Cau as part of the Isabelle translation of the laws [CMS19].

$a \simeq b,\ b \simeq c \vdash a \simeq c$ *MonEqTrans*(C.327)

4.4.3 Annihilator and identity laws

FAIL UPTO $a \simeq a$ *MFailUpto*(C.330)

FAIL THRU $a \simeq$ **FAIL** *MFailThru*(C.331)

FAIL AND $a \simeq$ **FAIL** *MFailAnd*(C.332)

a **THEN FAIL** $\simeq$ **FAIL** *MThenFail*(C.333)

FAIL THEN $a \simeq$ **FAIL** *MFailThen*(C.334)

FAIL WITH $f \simeq$ **FAIL** *MFailWith*(C.335)

a **WITH** false $\simeq$ **FAIL** *MWithFalse*(C.336)

a **WITH** true $\simeq a$ *MWithTrue*(C.337)

EMPTY UPTO $a \simeq$ **EMPTY** *MEmptyUpto*(C.338)

EMPTY THRU $a \simeq a$ *MEmptyThru*(C.339)

a **THEN EMPTY** $\simeq a$ *MThenEmpty*(C.340)

EMPTY THEN $a \simeq a$ *MEmptyThen*(C.341)

EMPTY ITERATE $b \simeq$ **EMPTY** *MEmptyIterate*(C.342)

4.4.4 Idempotence laws

a **UPTO** $a \simeq a$ *MUptoIdemp*(C.344)

a **THRU** $a \simeq a$ *MThruIdemp*(C.345)

a **AND** $a \simeq a$ *MAndIdemp*(C.346)

(**WITH** f) $\circ$ (**WITH** f) $\simeq$ (**WITH** f) *MWithIdemp*(C.347)

a **ITERATE** $a \simeq a$ *MIterateIdemp*(C.343)

The law *MWithIdemp*(C.347) uses the notation of operator sections: (**WITH** f) $= \lambda\, m \bullet m$ **WITH** f

4.4.5 Commutativity laws

a **UPTO** $b \simeq b$ **UPTO** a *MUptoCommut*(C.348)

a **THRU** $b \simeq b$ **THRU** a *MThruCommut*(C.349)

a **AND** $b \simeq b$ **AND** a *MAndCommut*(C.350)

(**WITH** f) $\circ$ (**WITH** g) $\simeq$ (**WITH** g) $\circ$ (**WITH** f) *MWithCommut*(C.351)

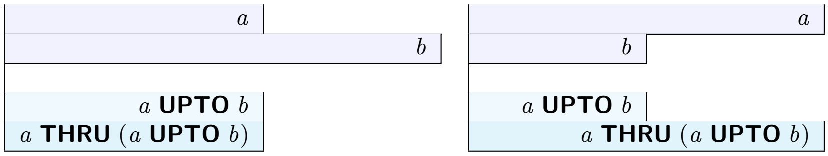


Figure 4.13: Pictorial representation of $MThruUptoAbsorp$[(C.358)]

4.4.6 Associativity laws

$$(a \ \mathsf{UPTO}\ b)\ \mathsf{UPTO}\ c \simeq a\ \mathsf{UPTO}\ (b\ \mathsf{UPTO}\ c) \qquad MUptoAssoc^{(C.352)}$$

$$(a \ \mathsf{THRU}\ b)\ \mathsf{THRU}\ c \simeq a\ \mathsf{THRU}\ (b\ \mathsf{THRU}\ c) \qquad MThruAssoc^{(C.353)}$$

$$(a \ \mathsf{AND}\ b)\ \mathsf{AND}\ c \simeq a\ \mathsf{AND}\ (b\ \mathsf{AND}\ c) \qquad MAndAssoc^{(C.354)}$$

$$(a \ \mathsf{THEN}\ b)\ \mathsf{THEN}\ c \simeq a\ \mathsf{THEN}\ (b\ \mathsf{THEN}\ c) \qquad MThenAssoc^{(C.355)}$$

4.4.7 Absorption laws

$$a\ \mathsf{UPTO}\ (a\ \mathsf{THRU}\ b) \simeq a \qquad MUptoThruAbsorp^{(C.357)}$$

$$a\ \mathsf{THRU}\ (a\ \mathsf{UPTO}\ b) \simeq a \qquad MThruUptoAbsorp^{(C.358)}$$

$$(\mathsf{WITH}\ f) \circ (\mathsf{WITH}\ g) \simeq (\mathsf{WITH}\ (f \wedge g)) \qquad MWithAbsorp^{(C.356)}$$

The second of these laws is illustrated in Figure 4.13.

4.4.8 Distributivity laws

a **UPTO** $(b$ **THRU** $c) \simeq (a$ **UPTO** $b)$ **THRU** $(a$ **UPTO** $c)$ *MUptoThruDistrib*(C.359)
$(a$ **THRU** $b)$ **UPTO** $c \simeq (a$ **UPTO** $c)$ **THRU** $(b$ **UPTO** $c)$ *MThruUptoRDistrib*(C.362)
a **THRU** $(b$ **UPTO** $c) \simeq (a$ **THRU** $b)$ **UPTO** $(a$ **THRU** $c)$ *MThruUptoDistrib*(C.361)
$(a$ **UPTO** $b)$ **THRU** $c \simeq (a$ **THRU** $c)$ **UPTO** $(b$ **THRU** $c)$ *MUptoThruRDistrib*(C.360)
a **THEN** $(b$ **AND** $c) \simeq (a$ **THEN** $b)$ **AND** $(a$ **THEN** $c)$ *MThenAndDistrib*(C.364)
a **THEN** $(b$ **UPTO** $c) \simeq (a$ **THEN** $b)$ **UPTO** $(a$ **THEN** $c)$ *MThenUptoDistrib*(C.366)
a **THEN** $(b$ **THRU** $c) \simeq (a$ **THEN** $b)$ **THRU** $(a$ **THEN** $c)$ *MThenThruDistrib*(C.367)
$(($**HALT** $w)$ **WITH** $f)$ **UPTO** $(($**HALT** $w)$ **WITH** $g) \simeq ($**HALT** $w)$ **WITH** $(f \vee g)$
MHaltWithUptoHaltWithEqvHaltWithOr(C.369)
$(($**HALT** $w)$ **WITH** $f)$ **AND** $(($**HALT** $w)$ **WITH** $g) \simeq ($**HALT** $w)$ **WITH** $(f \wedge g)$
MHaltWithAndDistrib(C.368)
$(($**HALT** $w)$ **WITH** $f)$ **THRU** $(($**HALT** $w)$ **WITH** $g) \simeq (($**HALT** $w)$ **WITH** $f)$ **AND** $(($**HALT** $w)$ **WITH** $g)$
MHaltWithThruHaltWithEqvHaltWithAndHaltWith(C.370)

Figure 4.14 provides a pictorial representation of the law *MThruUptoDistrib*(C.361).

4.4.9 Algebraic structures

The monitor properties listed in the previous sections permit various algebraic structures to be identified. Table 4.15 summarises the algebraic properties that are satisfied by these operators.

Using the properties summarised in Table 4.15 the following categorisations can be made.

- (ITL-Monitor, **AND**) is an idempotent, commutative semigroup.
- (ITL-Monitor, **THEN**, **EMPTY**) is a monoid.
- (ITL-Monitor, **UPTO**, **FAIL**) is an idempotent, commutative monoid. This is also a bounded, meet-semilattice with an order relation $a \leq b \Leftrightarrow a$ **UPTO** $b \simeq a$ with greatest element **FAIL**.
- (ITL-Monitor, **THRU**, **EMPTY**) is an idempotent, commutative monoid. This is also a bounded, join-semilattice with an order relation $a \geq b \Leftrightarrow a$ **THRU** $b \simeq a$ with least element **EMPTY**.
- (ITL-Monitor, **UPTO**, **FAIL**, **THRU**, **EMPTY**) is an idempotent semiring $(R, +, 0, \circ, 1)$. The absorption laws, *MThruUptoAbsorp*(C.358) and *MUptoThruAbsorp*(C.357), combine the

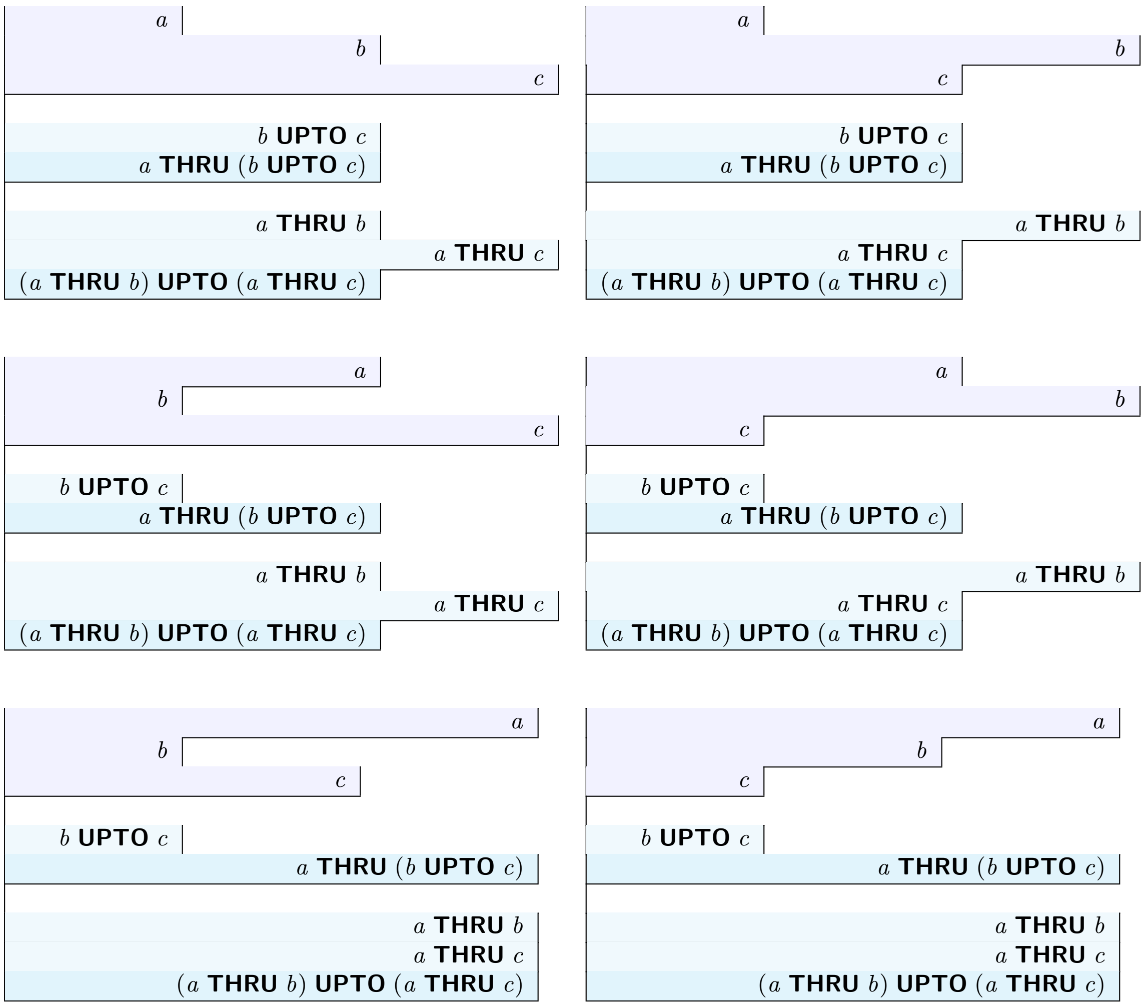
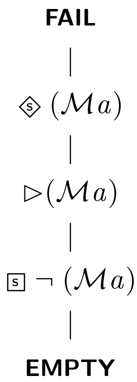


Figure 4.14: Pictorial representation of $MThruUptoDistrib^{(C.361)}$

	UPTO	**THRU**	**AND**	**THEN**	**ITERATE**	**WITH**
Annihilator(L)		**FAIL**	**FAIL**	**FAIL**	**FAIL**	**FAIL**
Annihilator(R)		**FAIL**	**FAIL**	**FAIL**		false
Identity(L)	**FAIL**	**EMPTY**		**EMPTY**	**EMPTY**	
Identity(R)	**FAIL**	**EMPTY**		**EMPTY**		true
Idempotent	✓	✓	✓		✓	
Commutative	✓	✓	✓			
Associative	✓	✓	✓	✓		
UPTO distributes through		LR				
THRU distributes through	LR					
THEN distributes through	L	L	L			
UPTO absorption		✓				
THRU absorption	✓					

Figure 4.15: Summary table of algebraic properties

two semilattices into a bounded lattice with top element **FAIL** and bottom element **EMPTY**.



The relationship between **UPTO**, **THRU**, **AND**, **EMPTY**, and **FAIL** is illustrated in Figure 4.16.

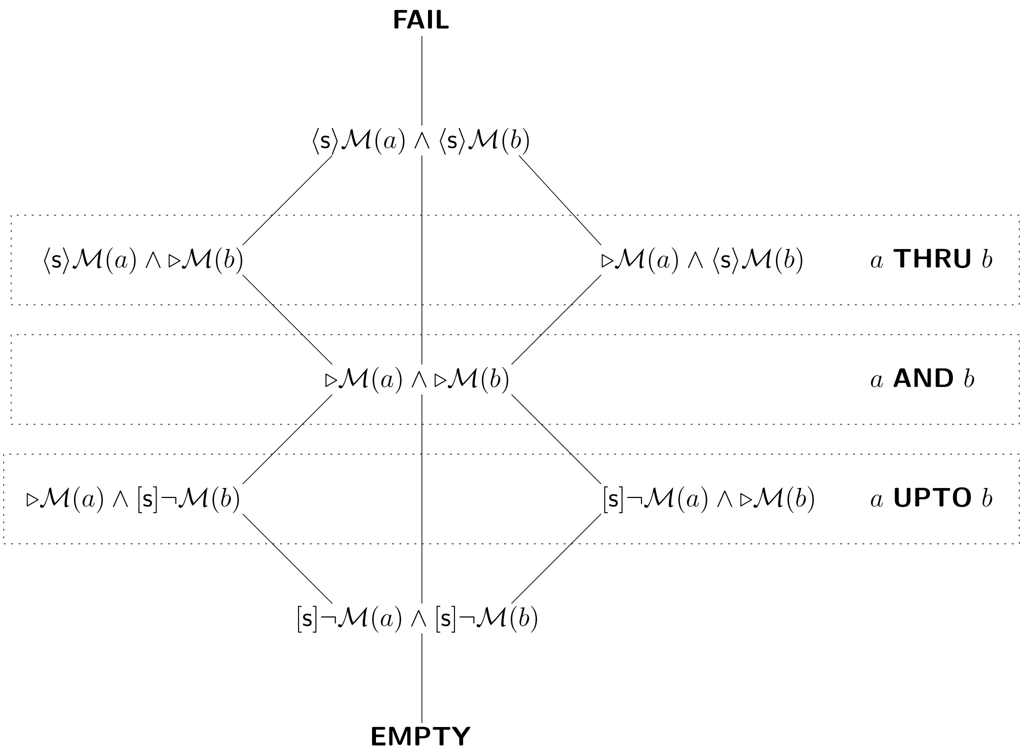


Figure 4.16: Lattice showing the relationship between **UPTO**, **AND**, and **THRU**

4.5 Example specification - scoring tennis

In this section a small example specification is presented that demonstrates the use of the ITL-Monitor operators. To assist with the explanation of the scenario, Z notation [Spi01] is used alongside the explanatory text.

Consider a piece of software that is monitoring two players' scores throughout a tennis match. The players are represented by the type *Player*.

$$Player ::= p1 \mid p2$$

In this example, the two players compete by playing the best of five sets. To win a set a player has to win at least six games and to have won at least two more games than their opponent. Thus the set may end with a scores of 6-0 or 9-7 for example, but not 6-5.[4]

A game is won by the first player to win at least four points and to have at least two more points than their opponent. Rather than use simple numbering, $1 \mathrel{..} 4$, the points in tennis games have special names: *fifteen*, *thirty*, *forty*, and *game*, respectively, with a special extra point called *advantage*. The type *Point* enumerates these values.

$$Point ::= love \mid fifteen \mid thirty \mid forty \mid advantage \mid game$$

Because a player must win a game by two clear points, if the score is *forty-forty* then the player that wins the next point moves to *advantage* rather than *game*. Similarly, if a player at *advantage* fails to win the next point then the score reverts to *forty-forty*. The point transitions that occur when player $p1$ wins a point are specified by the function $updatePoints$. Each pair of points represents scores for players $(p1, p2)$ respectively.

$$\begin{array}{|l}
updatePoints : (Point \times Point) \rightarrow (Point \times Point) \\
\hline
updatePoints = \\
\quad \{ (love, love) \mapsto (fifteen, love), (love, fifteen) \mapsto (fifteen, fifteen), \\
\quad\quad (love, thirty) \mapsto (fifteen, thirty), (love, forty) \mapsto (fifteen, forty), \\
\quad\quad (fifteen, love) \mapsto (thirty, love), (fifteen, fifteen) \mapsto (thirty, fifteen), \\
\quad\quad (fifteen, thirty) \mapsto (thirty, thirty), (fifteen, forty) \mapsto (thirty, forty), \\
\quad\quad (thirty, love) \mapsto (forty, love), (thirty, fifteen) \mapsto (forty, fifteen), \\
\quad\quad (thirty, thirty) \mapsto (forty, thirty), (thirty, forty) \mapsto (forty, forty), \\
\quad\quad (forty, love) \mapsto (game, love), (forty, fifteen) \mapsto (game, fifteen), \\
\quad\quad (forty, thirty) \mapsto (game, thirty), (forty, forty) \mapsto (advantage, forty), \\
\quad\quad (forty, advantage) \mapsto (forty, forty), (advantage, forty) \mapsto (game, forty) \}
\end{array}$$

The tennis match may be represented by the following state comprising variables for the

[4] In this example tie breaks for a set are not used.

number of points, games, and sets. Each variable holds a pair of values for $p1$ and $p2$ respectively.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Match}} \\
\hline
Points : Point \times Point \\
Games : \mathbb{N} \times \mathbb{N} \\
Sets : \mathbb{N} \times \mathbb{N} \\
\hline
\end{array}$$

At the start of the match all the scores are zero:[5]

$$StartMatch \mathrel{\widehat=} [Match' \mid Points' = (love, love) \land Games' = (0,0) \land Sets' = (0,0)]$$

The operation to update the scores when a point is won is defined as the sequential composition of three operations: *UpdatePoints*, then *UpdateGames*, then *UpdateSets*. Each of these is presented below. (In Z an operation relates the *before* values of the state and their *after* values. Within each operation schema the before states are introduced by including *Match* and the after states are introduced by including *Match′*. The primed variables represent the after states.)

UpdatePoints inputs the winner of the point and uses this to decide how to update the points. If $p2$ wins the point then the arguments to, and results from, the *updatePoints* function must be reversed. The other state variables are not changed.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{UpdatePoints}} \\
\hline
Match \\
Match' \\
winner? : Player \\
\hline
winner? = p1 \Rightarrow Points' = updatePoints(Points) \\
winner? = p2 \Rightarrow Points' = \ \mathbf{let}\ (x1, x2) == Points; \\
\qquad\qquad\qquad\qquad\qquad\qquad (y2, y1) == updatePoints(x2, x1) \bullet (y1, y2) \\
Games' = Games \\
Sets' = Sets \\
\hline
\end{array}$$

UpdateGames must check to see if a game has just been won by either of the players. This is determined by inspecting the first and second fields of the *Points* variable respectively. If either player's score has reached *game* that player's game count is incremented. If neither player has won a game then the state variables are unchanged. In any case the *Sets* variable is unchanged at this stage and will be checked in the subsequent schema. Note that the first two conjuncts are mutually exclusive because only one player can win a game.

[5]In Z initial states are conventionally primed – i.e. they are considered as *after* states.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \textit{UpdateGames} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\textit{Match} \\
\textit{Match}' \\
\hline
\textit{first}(\textit{Points}) = \textit{game} \Rightarrow \textit{Games}' = (\textit{first}(\textit{Games}) + 1, \textit{second}(\textit{Games})) \\
\textit{second}(\textit{Points}) = \textit{game} \Rightarrow \textit{Games}' = (\textit{first}(\textit{Games}), \textit{second}(\textit{Games}) + 1) \\
\textit{first}(\textit{Points}) \neq \textit{game} \land \textit{second}(\textit{Points}) \neq \textit{game} \Rightarrow \textit{Games}' = \textit{Games} \\
\textit{Points}' = \textit{Points} \\
\textit{Sets}' = \textit{Sets} \\
\hline
\end{array}
$$

UpdateSets must check to see if a set has just been won by one of the players. If neither player has won a set then no state variables are changed. The operation also outputs a set of winners which is empty if neither player has won, or contains the winning player otherwise. Once again, the first two conjuncts are mutually exclusive because only one player can win a game.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \textit{UpdateSets} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\textit{Match} \\
\textit{Match}' \\
\textit{winner}! : \mathbb{P}\, \textit{Player} \\
\hline
\textit{first}(\textit{Games}) \geq 6 \land \textit{first}(\textit{Games}) > (\textit{second}(\textit{Games}) + 1) \Rightarrow \\
\qquad \textit{Sets}' = (\textit{first}(\textit{Sets}) + 1, \textit{second}(\textit{Sets})) \\
\textit{second}(\textit{Games}) \geq 6 \land \textit{second}(\textit{Games}) > (\textit{first}(\textit{Games}) + 1) \Rightarrow \\
\qquad \textit{Sets}' = (\textit{first}(\textit{Sets}), \textit{second}(\textit{Sets}) + 1) \\
\left(\begin{array}{l} (\textit{first}(\textit{Games}) < 6 \ \lor \textit{second}(\textit{Games}) < 6 \lor \\ \textit{abs}(\textit{first}(\textit{Games}) - \textit{second}(\textit{Games})) \leq 1) \end{array} \right) \Rightarrow \textit{Sets}' = \textit{Sets} \\
\textit{Games}' = \textit{Games} \\
\textit{Points}' = \textit{Points} \\
\textit{winner}! = \ \begin{array}[t]{l} \textbf{if } \textit{first}(\textit{Sets}) \geq 2 \textbf{ then } \{p1\} \\ \textbf{else if } \textit{second}(\textit{Sets}) \geq 2 \textbf{ then } \{p2\} \\ \textbf{else } \varnothing \end{array} \\
\hline
\end{array}
$$

Finally, the *PlayPoint* operation can be expressed as the sequential composition of the three component operations. (In Z schema composition $S \fatsemi T$ equates the after state of S with the before state of T and hides this intermediate state.)

$$\textit{PlayPoint} \mathrel{\widehat=} \textit{UpdatePoints} \fatsemi \textit{UpdateGames} \fatsemi \textit{UpdateSets}$$

The following two operations specify the state transitions required to reset the *Points* scores and the *Sets* scores following a game-win and a set-win respectively. The schemas *ResetPoints* and *ResetGames* specify the resetting of the relevant state variables. The schema *NewGame* is called following a game-win but not a set-win. The schema *NewSet* is called following a set-win and it incorporates the act of starting a new game.

$$\begin{array}{ll}
\mathit{ResetPoints} & \mathrel{\widehat=} [\mathit{Match};\ \mathit{Match}' \mid \mathit{Points}' = (\mathit{love}, \mathit{love})] \\
\mathit{ResetGames} & \mathrel{\widehat=} [\mathit{Games}' = (0,0)] \\
\mathit{NewGame} & \mathrel{\widehat=} [\mathit{ResetPoints} \mid \mathit{Games}' = \mathit{Games} \land \mathit{Sets}' = \mathit{Sets}] \\
\mathit{NewSet} & \mathrel{\widehat=} [\mathit{ResetPoints};\ \mathit{ResetGames} \mid \mathit{Sets}' = \mathit{Sets}]
\end{array}$$

The scoring of a particular tennis match is recorded by a trace in which each state holds the variables *Points*, *Games*, and *Sets*. For example, the trace after seven points have been played might be:

Points	$(0,0)$	$(0,0)$	$(0,15)$	$(0,30)$	$(0,40)$	$(15,40)$	$(15,G)$	$(0,0)$	$(15,0)$
Games	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,1)$	$(0,1)$	$(0,0)$
Sets	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
σ	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8

In the above example, σ_0 represents the initial state specified by *StartMatch*. The state σ_1 is the start of the first set.[6] Each of the transitions $\sigma_1 \to \sigma_2$, $\sigma_2 \to \sigma_3$, $\sigma_3 \to \sigma_4$, $\sigma_4 \to \sigma_5$, $\sigma_5 \to \sigma_6$, and $\sigma_7 \to \sigma_8$ represent state changes specified by *PlayPoint*. The state transition $\sigma_6 \to \sigma_7$ represents the state change specified by *NewGame*. It is useful to consider the suffix of some possible later trace that includes a set win:

Points	$(15,30)$	$(15,40)$	$(15,G)$	$(0,0)$	$(0,15)$	$(0,30)$	$(0,40)$	$(0,G)$	$(0,0)$
Games	$(4,5)$	$(4,5)$	$(4,6)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,1)$	$(0,1)$
Sets	$(0,0)$	$(0,0)$	$(0,1)$	$(0,1)$	$(0,1)$	$(0,1)$	$(0,0)$	$(0,1)$	$(0,1)$
σ	σ_{70}	σ_{71}	σ_{72}	σ_{73}	σ_{74}	σ_{75}	σ_{76}	σ_{77}	σ_{78}

In this example the final game in the first set is won in state σ_{72} – this state is the end of a game subinterval and a set subinterval. The next state σ_{73} shows the reset values of *Points* and *Games* and maintains the updated *Sets*. Play continues and state σ_{77} shows the end of the next game, etc.

The previously-defined *updatePoints* function enumerates all of the possible point score transitions for both players from the perspective of $p1$ winning the point. However, the set of legal point score transitions for a single player can be specified as follows:

[6] The fact that $\sigma_1 = \sigma_0$ looks odd but it is only a special case because σ_1 is the beginning of the *first* set.

$$
\begin{array}{|l}
\_ \rightsquigarrow \_ : Point \leftrightarrow Point \\
\hline
\forall\, p_1, p_2 : Point \bullet \\
\quad p_1 \rightsquigarrow p_2 \Leftrightarrow \\
\qquad (p_1, p_2) \in \{\ (love, fifteen), (fifteen, thirty), (thirty, forty), (forty, game) \\
\qquad\qquad (forty, advantage), (advantage, forty), (advantage, game)\ \}
\end{array}
$$

Thus it is possible to capture a temporal property relating $Points(p)$ and $\bigcirc(Points(p))$ for each player p following each point-win.[7]

$$
\begin{aligned}
winPoint = skip \wedge (\ & (\mathsf{stable}(Points(p1)) \wedge (Points(p2) \rightsquigarrow \bigcirc(Points(p2)))\ ) \vee \\
& (\mathsf{stable}(Points(p2)) \wedge (Points(p1) \rightsquigarrow \bigcirc(Points(p1)))\ )\ )
\end{aligned}
$$

validGame specifies an interval representing a single game. Initially the *Points* score for each player is set to *love*, and then a finite sequence of *winPoint* transitions occurs. The relationship between the *Games* scores for each player are defined by their values across the whole interval. The ITL formula $A \lessdot\!\sim e$ means that the value of state variable A is stable until the final state in which its value must equal e. Thus, over the course of a single game one player's game score must remain constant, whereas the opponent's game score will increase by one in the final state: i.e. when the game is won. The end of the interval is determined as soon as either player's *Points* score reaches *game*.

$$
\begin{aligned}
validGame &= (Points(p1) = love) \wedge (Points(p2) = love) \wedge \\
&\quad winPoint^* \wedge \\
&\quad (\ ((Games(p1) \lessdot\!\sim (Games(p1)+1)) \wedge \mathsf{stable}(Games(p2))) \vee \\
&\quad\ \ ((Games(p2) \lessdot\!\sim (Games(p2)+1)) \wedge \mathsf{stable}(Games(p1)))\ ) \\
gameOver &= (Points(p1) = game) \vee (Points(p2) = game)
\end{aligned}
$$

An interval representing a game must satisfy $validGame \wedge \mathsf{halt}(gameOver)$. The reason it is convenient to split this into two parts is because $\mathsf{halt}(gameOver)$ can be used to determine the length of the interval consumed by the corresponding ITL-Monitor. Specifically, the interval will extend to the first occurrence of the state formula *gameOver*. Then the formula *validGame* can be checked against the accrued interval.

A set of tennis is won by the player who is the first to win at least two more games than their opponent *and* who has won at least six games. Initially both players' *Games* scores are set to zero.

[7] In ITL the formula $\mathsf{stable}(A)$ means that A's value does not change throughout the interval.

$$\begin{aligned}
\mathit{validSet} = {} & (\mathit{Games}(p1) = 0) \wedge (\mathit{Games}(p2) = 0) \wedge \\
& (\ ((\mathit{Sets}(p1) \lessapprox (\mathit{Sets}(p1) + 1)) \wedge \mathsf{stable}(\mathit{Sets}(p2))) \vee \\
& \ \ ((\mathit{Sets}(p2) \lessapprox (\mathit{Sets}(p2) + 1)) \wedge \mathsf{stable}(\mathit{Sets}(p1)))\ ) \\
\mathit{setOver} = {} & (\ ((\mathit{Games}(p1) >= 6) \wedge (\mathit{Games}(p2) + 1 < \mathit{Games}(p1))) \vee \\
& \ \ ((\mathit{Games}(p2) >= 6) \wedge (\mathit{Games}(p1) + 1 < \mathit{Games}(p2)))\ )
\end{aligned}$$

Once again, note that it is convenient to split the specification for playing a single set into two parts. A set must satisfy $\mathit{validSet} \wedge \mathsf{halt}(\mathit{setOver})$. In a similar way to *gameOver* (above) the first occurrence of *setOver* will be used to define the extent of the interval covered by the corresponding monitor. A match is over as soon as one of the players wins two sets. Thus the extent of a match is specified as $\mathsf{halt}(\mathit{matchOver})$, where

$$\mathit{matchOver} = (\mathit{Sets}(p1) = 3) \vee (\mathit{Sets}(p2) = 3)$$

The scoring system for a tennis match conveniently splits the specification into subintervals. Furthermore, the fact that *Points* are reset to zero at the start of each new game, and *Games* are reset to zero at the start of each new set, provides a set of natural fusion points across which backtracking is unnecessary. In the first state the initialisation requires:

$$\begin{aligned}
\mathit{startMatch} = {} & \mathit{Points}(P1) = \mathit{love} \wedge \mathit{Points}(P2) = \mathit{love} \wedge \\
& \mathit{Games}(P1) = 0 \wedge \mathit{Games}(P2) = 0 \wedge \\
& \mathit{Sets}(P1) = 0 \wedge \mathit{Sets}(P2) = 0
\end{aligned}$$

A runtime monitor for checking a tennis scoring program against this specification can now be constructed using ITL-Monitor.

$$\begin{aligned}
\mathit{bygame} = {} & \textbf{GUARD}\ (\mathit{startMatch})\ \textbf{THEN HALT}\ (\mathit{matchOver})\ \textbf{ITERATE}\ (\\
& \qquad (\textbf{SKIP THEN HALT}\ (\mathit{setOver})\ \textbf{ITERATE}\ (\\
& \qquad\qquad \textbf{SKIP THEN HALT}\ (\mathit{gameOver})\ \textbf{WITH}\ (\mathsf{skip}\ ;\ \mathit{validGame}) \\
& \qquad\qquad) \\
& \qquad)\ \textbf{WITH}\ (\mathsf{skip}\ ;\ \mathit{validSet}) \\
&)
\end{aligned}$$

4.5.1 Running the verification

As the tennis program under test proceeds it sends each update to any of the state variables to the monitor. Let us consider the evaluation of the first set – for the purpose of this discussion it will be assumed that the set consists of six games. Each game is preceded by **SKIP** to separate each one from the next - i.e. by skipping from the end of one game to the start of the next game. The diagram in Figure 4.17 shows the interval evolving as each of the games in a set is verified.

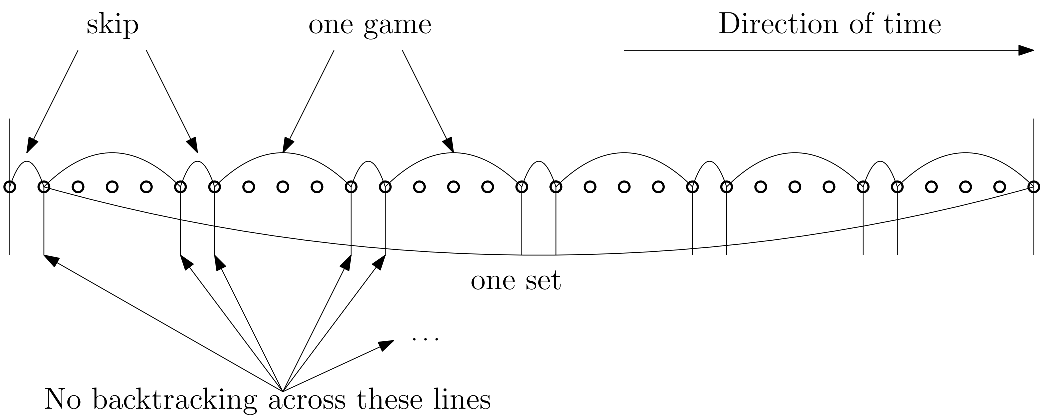


Figure 4.17: A set of tennis

Games do not share states which is why each is preceded by a single skip. The verification of a single game may involve backtracking since the specification is non-deterministic. However, backtracking cannot cross the vertical lines: each game is extended to the *first occurrence* of *gameOver* which cannot be satisfied by any other fusion point.

4.5.2 Adjusting the granularity of the analysis

The *match* monitor reports on whether or not the system under test meets the specification after each game is played: this is illustrated in Figure 4.17. Thus the 'granularity' of the analysis is at the level of a single tennis game. Recall that the specification of a game involves non-determinism which is why it cannot be checked after every state.

It is possible to decrease the granularity of the analysis by testing an interval after *each set*. The following ITL specification covers a single set of tennis:

$$
\begin{aligned}
&byset = \textsf{\textbf{GUARD}}\,(startMatch)\ \textsf{\textbf{THEN HALT}}\,(matchOver)\ \textsf{\textbf{ITERATE}}\ (\\
&\qquad\qquad (\textsf{\textbf{SKIP THEN HALT}}\,(setOver))\ \textsf{\textbf{WITH}}\\
&\qquad\qquad\qquad ((\mathsf{skip}\ ;\ (\mathsf{halt}(gameOver) \wedge validGame))^{*} \wedge (\mathsf{skip}\ ;\ validSet))\\
&\qquad)
\end{aligned}
$$

This monitor will not be able to report upon any violations of the specification until a whole set has been played. Furthermore, at the points of verification – in this case when a set has completed – the evaluation will take longer when compared to the sum of several much smaller evaluations following each game.

Ultimately, it is possible to decrease the granularity of the analysis to cover an entire tennis match. This effectively generates the entire trace for subsequent analysis. The revised ITL formula would be:

$$byMatch = \textsf{\textbf{GUARD}}\,(startMatch)\ \textsf{\textbf{THEN HALT}}\,(matchOver)\ \textsf{\textbf{WITH}}\ validMatch$$

This requires significant backtracking given the embedded choice within each of the sub-specifications, and the employment of two levels of chopstar. In this case all of the verification would take place as soon as the match is over but no verification would occur while the program is running.

This example has demonstrated a decrease of the granularity of the analysis from games, to sets, to the whole match. Is it possible to *increase* the granularity of the analysis to the level of individual states? It is *not* possible simply to reduce the scope of each monitor to analyse every pair of adjacent states: the non-determinism in the specification does not permit this.

4.6 Summary

This chapter has presented ITL-Monitor syntax and translation into ITL. The monitor operators were described informally with illustrative examples. The small case study at the end of the chapter showed how ITL-Monitor could be used to specify a runtime monitor for an example system. An investigation into the mathematical properties of monitors was undertaken and the results presented. Algebraic structures were identified for combinations of ITL-Monitor operators and these have been organised and presented. An important monitor law, $MFixFst^{(C.309)}$, was established and its rôle in the maintenance of first occurrence properties of monitors was explained.

Chapter 5

ITL Monitor implementation

This chapter describes the implementation of ITL-Monitor. The library contains two principal objects, `ITL` and `Monitor`, each of which is described below. `ITL` defines an Application Programming Interface (API) for creating ITL formulae, and `Monitor` defines the API for specifying runtime verification monitors in ITL-Monitor. Section 5.1.1 introduces the `ITL` library, including the implementation of each of its key components. Section 5.1.2 introduces the `Monitor` library from the perspective of constructing user-defined specifications. The translation of the derived monitors using a range of optimisation flags is also explained. Section 5.2 looks in detail at the underlying representation of runtime monitors as a network of Akka actors [Wya13, Akk17]. The process of creating an Akka network representing an abstract monitor is described, and the way the network acts as a monitor, receiving messages from a program, forming and returning judgements, is illustrated with a worked example.

Throughout this chapter both ITL formulae and ITL-Monitor specifications, which represent ITL formulae, may be referred to both as temporal logic formulae – as mathematical objects – or as processes. Mathematically, **FIRST** (f) represents an ITL formula, $\triangleright f$, that may satisfied by some finite interval. Operationally, **FIRST** (f) is a monitor (a process) that maintains an internal representation of the states that have been passed to it (an interval), and terminates successfully as soon as the interval satisfies f. Mathematically, **FIRST** (f) **AND FIRST** (g) is an executable ITL-Monitor representing the ITL formula $\triangleright f \wedge \triangleright g$. Operationally, it represents a concrete monitor – a hierarchy of Akka actors – in which the two submonitors **FIRST** (f) and **FIRST** (g) could be executing in parallel. These examples reflect the purpose of this thesis: to create a library of executable, runtime monitors that represent ITL formulae directly, and whose combination reflect new executable monitors that can verify the composition of their respective formulae.

Listing 5.1: Monitor.scala

```
object ITL {
  abstract class Var[T]
  case class Val[T](v: T)
  type VarUpdate = (Var[T],T) forSome {type T}
  class Interval {
    def get[T](k: Int, v: Var[T]): Option[Val[T]]
    def add(updates: VarUpdate* ): Interval
  }
  abstract class Expr[T] {
    def evalExpr[T](expr: Expr[T], sigma: Interval): Option[Const[T]]
    case class Const[T](c: T)                                  extends Expr[T]
    case class Ref[T](v: Var[T])                               extends Expr[T]
    case class Unary[T,U](op: T=>U, x: Expr[T])                extends Expr[U]
    case class Binary[T,U,V](op: (T,U)=>V, x: Expr[T], y: Expr[U]) extends Expr[V]
    case class With[T,U](x: Expr[T], f: Const[T] => Expr[U])   extends Expr[U]
    case class Next[T](v: Var[T])                              extends Expr[T]
    case class Fin[T](v: Var[T])                               extends Expr[T]
    case class IntLen()                                        extends Expr[Int]
  }
  abstract class Formula {
    def evalFormula(sigma: Interval): Boolean
    case class Exp     (x: Expr[Boolean])                 extends Formula
    case class Not     (f: Formula)                       extends Formula
    case class Final   (f: Formula)                       extends Formula
    case class And     (f: Formula, g: Formula)           extends Formula
    case class Len     (n: Int)                           extends Formula
    case class Chop    (f1: Formula, f2: Formula)         extends Formula
    case class Chopstar(f: Formula)                       extends Formula
  }
}
```

Figure 5.1: Overview of ITL.scala

5.1 Application programming interface

ITL-Monitor is implemented as a collection of objects and classes in Scala. These are defined in two Scala files: ITL.scala which contains support for ITL specifications and intervals; and Monitor.scala which provides the implementation of ITL-Monitor. Each of these is presented below.

5.1.1 ITL

The ITL API provides all the components necessary to define and evaluate an ITL formula with respect to an interval. An overview of the key objects, classes, types, and methods is presented in Figure 5.1. (The code appears in full in Listing A.1). The figure shows the classes relating to variables, values, intervals, expressions and formulae. Unnecessary detail has been omitted including some superclass relationships, internal data structures, and derived operators.

ITL expressions and formulae are implemented as data structures (trees) whose definitions

```
Not(
  Chop(
    Chop(
      Exp(
        Const(true)
      ),
      And(
        Len(1),
        Not(
          Exp(
            Binary(
              RelInt.EQ,
              Binary(
                NumInt.Mul,
                Ref(A),
                Const(2)
              ),
              Fin(A)
            )
          )
        )
      )
    ),
    Exp(
      Const(true)
    )
  )
)
```

Figure 5.2: Representation of the ITL formula $A\,\mathsf{gets}\,(A*2)$

appear in Figure 5.1 (lines 9-19 and 20-29). For example, the ITL formula $A\,\mathsf{gets}\,(A*2)$ is shown in Figure 5.2.

The expressions and formulae data structure constitutes a deeply embedded DSL (cf. 2.4.1). The use of case classes to represent the ‘nodes’ in the expression/formulae trees allows Scala’s pattern matching to recognise structures for evaluation and/or rewriting. An example of rewriting is given by the function `not`: when evaluating `not(f)` the structure of `f` is matched to avoid the potential of returning a doubly-negated formula.

```
def not(f: Formula): Formula = f match { case (Not(g)) => g
                                         case _        => Not(f) }
```

Without this optimisation the subformula `And(..)` in Figure 5.2 (lines 7-23) would have been constructed as `Not(Not(And(..)))`.

Pattern matching is also used in the evaluation of formulae. For example, the (private) `f.evalFormula(sigma, i, j)` method application evaluates a formula `f` with respect to the subinterval $\mathtt{sigma}_{\mathtt{i..j}}$. Evaluation of formulae matching the pattern `Chop(f, g)` can be simplified if one of the subformulae specifies a subinterval of a specific length. The method application `f.fixed()` returns `None` if `f` is not a fixed-length formula, and `Some(m)`, where `m` is a positive integer representing the fusion point, otherwise. If either `f` or `g` has a fixed length then the search for a fusion point can be reduced from $j - i + 1$ possibilities to just one.

```
case Chop(f, g) => (f.fixed(), g.fixed()) match {
       case (None,None)       => (i to j).toStream.map(k =>
                                   if (f.evalFormulaFromTo(sigma, i, k))
                                     g.evalFormulaFromTo(sigma, k, j)
                                   else
                                     false).contains(true)
       case (Some(m),None)    => (i+m >= i && i+m <= j) &&
                                 f.evalFormulaFromTo(sigma, i, i+m) &&
                                 g.evalFormulaFromTo(sigma, i+m, j)
       case (None,Some(n))    => (j-n >= i && j-n <= j) &&
                                 f.evalFormulaFromTo(sigma, i, j-n) &&
                                 g.evalFormulaFromTo(sigma, j-n, j)
       case (Some(m),Some(n)) => (i+m >= i && i+m <= j) &&
                                 (i+m == j-n) &&
                                 f.evalFormulaFromTo(sigma, i, i+m) &&
                                 g.evalFormulaFromTo(sigma, i+m, j)
     }
```

5.1.1.1 Variables

Figure 5.1 (line 2) introduces a parameterised, abstract class `Var[T]` representing variables for use in ITL formulae. The use of the generic type variable `T` allows the Scala type checker to perform valuable static type analysis on any user-defined variables.

User-defined variables are created as objects, subclassing from a suitably parameterised instance of `Var[T]`. The use of an object for each variable ensures that it is identified with a single `Var` instance. Each user-defined variable will inherit a range of ITL operators that are defined for state variables including $V \lessdot\!\sim e$ (padded temporal assignment), $V \leftarrow e$ (temporal assignment), $V := e$ (next-state assignment), and $V \texttt{ gets } e$ (unit delay).

For example, a state variable C that stores a character value, and B that stores a Boolean value, are declared as follows.

```
object C extends Var[Char]    { override def toString = "C" }
object B extends Var[Boolean] { override def toString = "B" }
```

The overridden `toString` methods permit the client to define how the variables should appear when printed on the output stream.

On line 4 of the listing the type `VarUpdate` represents a tuple in which the components are existentially quantified. Thus, the type `T` does not appear at top-level with the type itself. This permits a list of `VarUpdate` tuples to be constructed in which each pair has correctly matched types but differently typed tuples can appear in the same `List[VarUpdate]`, e.g., `List( (C, 'x'), (B, true) )`. This is useful in the context of runtime verification in which variables of different types may be updated simultaneously.

5.1.1.2 Values

Figure 5.1 (line 3) introduces a parameterised case class that is used to construct constant values in ITL formulae. This is parameterised so that the type checker can ensure that values are stored in variables of the same type. Looking ahead to the process of updating the state during a runtime verification, a monitor `mu` may have its variables updated using a `set` method, viz:

```
mu.set(B, true).set(C, 'x')
```

The monitor `set` method builds internally a list of variable updates as shown at the end of the previous section (5.1.1.1).

5.1.1.3 Intervals

Figure 5.1 (lines 5-8) introduces a class `Interval` as an opaque type.[1] It is not possible to parameterise the interval type, i.e. `Interval[T]`, because this would require all the variables in the state to be of the same type. Similar to the `VarUpdate` type (cf. 5.1.1.1), the interval contains mappings in which each variable is bound to a value of the appropriate type.

Intervals are represented internally using a Scala immutable `HashMap`. However, its behaviour can be captured by the following function in which the generic type T is hidden using existential quantification.

[1] See Listing A.1 for the full implementation of intervals.

$$Interval = \exists\, T \bullet Var[T] \to (\mathbb{N} \to Val[T])$$

For each variable, a mapping is maintained in which the (index,value) pairs are only stored when changes occur. The value if variable v at index i is equal to the value of v looked up at the largest index $j \leq i$ for which an entry exists. This approach avoids potentially duplicated values being stored in successive states and is similar to the representation used in ITL Tracer [Jan10]. Locating the most recent index is currently a linear search (from i down to j).[2] Once the index has been located, lookup in a Scala hashmap is constant time.

Consider the following example in which each state consists of two variables, $X : Var[Int]$ and $Y : Var[Boolean]$. The interval σ:

X	1	1	1	7	7	7	7	7	9	9
Y	t	t	t	t	f	f	f	f	f	t
index	0	1	2	3	4	5	6	7	8	9

is stored using the hashmap `sigma`:

$$\{X \mapsto \{0 \mapsto 1, 3 \mapsto 7, 8 \mapsto 9\}, Y \mapsto \{0 \mapsto t, 4 \mapsto f, 9 \mapsto t\}\}$$

In this example, $lookup(\texttt{sigma})(X)(5) = 7$.

5.1.1.4 Expressions and formulae

Figure 5.1 (lines 9-19) introduces the abstract class `Expr[T]` which represents expressions of type `T`. Lines 20-29 introduce the abstract class `Formula`. The basic forms of expressions and formulae are provided as subclasses. Scala allows class member methods to be used infix, i.e. method calls of the form `o.m(p)` can be written `o m p`. Coupled with the ability to use symbolic method names, this syntactic sugar permits a useful range of infix binary operators to be provided for expressions and formulae. Two examples from each abstract class are provided below. Firstly, infix operators are provided for addition and comparison.

```
abstract class Expr[T] {
  def + (that: Expr[T])(implicit o: Num[T]): Expr[T] = Binary(o.Add, this, that)
  def < (that: Expr[T])(implicit o:Ord[T]): Expr[Boolean] = Binary(o.LT, this, that)
```

The implicit parameters refer to objects that define the appropriate functions for each type. Thus a `Num[Int]` object, `NumInt`, contains the function `Add` for integers, and an `Ord[Char]`

[2] Future work will consider optimisations for sparse mappings which may derive or maintain index iterators.

object, `OrdChar`, contains a less-than order relation `LT` for characters etc. The `ITL` API exports implicit objects defining these functions for all of the Scala primitive types.

Scala's implicit parameters can be applied automatically by the compiler provided there is a suitably typed value in scope. Thus, if `a` and `b` are integer expressions then instead of writing `a.+(b)(NumInt)`, the expression can be written more succinctly as `a + b`, and `NumInt` is appended silently by the compiler.

Useful infix operators have also been provided for formulae.

```
abstract class Formula {
  def `;`(that: Formula) = Chop(this,that)
  def and(that: Formula) = And(this,that)
```

This syntax permits ITL formulae to be written, e.g., `(f and g) `;` h`.

5.1.1.5 Derived operators

All of ITL's standard derived operators are provided within the API as functions. Some of these use the standard names such as `next` and `eventually` whereas others use descriptions of their mathematical symbols (which is common in ITL) such as `di` ('diamond-i', $\diamondsuit$). Some examples are shown below:

```
  def chop(f1: Formula, f2: Formula): Formula = Chop(f1, f2)
  val skip: Formula                           = Len(1)
  def next(f: Formula): Formula               = chop(skip, f)
  def eventually(f: Formula): Formula         = chop(TRUE, f)
  def di(f: Formula): Formula                 = chop(f, TRUE)
```

5.1.2 Monitor

The Monitor API provides the components necessary to define and execute an ITL-Monitor. An overview of the key objects, classes, types, and methods is presented in Figure 5.3. (The full listing appears as Listing A.2). The figure shows the exported class `Abstr.Monitor` whose subclasses constitute the core ITL-Monitor syntax. Unnecessary detail has been omitted.

Object `Protocol` (Figure 5.3, lines 2-10), defines the messages which form the communication between the monitors internally, and the client program externally. `Step` extends the execution trace by one state and passes the updated state as a list of variable updates (cf. 5.1.1.1). `Show` causes the underlying implementation to display a version of itself on the

Listing 5.2: Monitor.scala

```scala
object Monitor {
  object Protocol { /* Communication protocol between monitors and clients */
    abstract class Request
    case class  Step(updates: List[VarUpdate]) extends Request
    case class  Show(indent: Int)              extends Request
    abstract class Reply
    case object Fail extends Reply
    case object More extends Reply
    case class  Done(updates: List[VarUpdate]) extends Reply
  }
  object OptimisationFlags {
    class OpTy
    case object ANY_STATE    extends OpTy
    case object ALL_STATES   extends OpTy
    case object ANY_PREFIX   extends OpTy
    case object ALL_PREFIXES extends OpTy
    case object CHECK_ONCE   extends OpTy
  }
  object Abstr {
    class Monitor {
      def ::(name:              String): Monitor = Name(name,this)
      def UPTO(that:           Monitor): Monitor = Upto(this,that)
      def THRU(that:           Monitor): Monitor = Thru(this,that)
      def THEN(that:           Monitor): Monitor = Then(this,that)
      def AND(that:            Monitor): Monitor = And(this,that)
      def ITERATE(that:        Monitor): Iterate = Iterate(this,that)
      def WITH(opt : OpTy, f: Formula): Monitor = With(this,opt,f)
      def WITH(            f :Formula): Monitor = With(this,CHECK_ONCE,f)
      def TIMES(k:                 Int): Monitor = if (k==0) EMPTY
                                                   else this THEN (this TIMES (k-1))
      def ALWAYS(w:            Formula): Monitor = With(this,ALL_STATES,w)
      def SOMETIME(w:          Formula): Monitor = With(this,ANY_STATE,w)
      def WITHIN(f:            Formula): Monitor = With(this,ALL_PREFIXES,
                                                     more implies (not(f) ';' skip))
    }
    def FIRST(f: Formula)                 = First(ANY_PREFIX, f)
    def LEN(k: Int)                       = FIRST(len(k))
    def SKIP                              = LEN(1)
    def EMPTY                             = FIRST(empty)
    def FAIL                              = FIRST(false) WITHIN (empty)
    def HALT(w: Formula)                  = First(ANY_STATE, w)
    def SKIPTO(w: Formula)                = SKIP THEN HALT(w)
    def GUARD(w: Formula)                 = EMPTY WITH (w)
    def UNTIL (w1 : Formula, w2: Formula) = HALT(w2) WITH (bm(w1))

    case class  Name    (name: String, a: Monitor)           extends  Monitor
    case class  First   (opt: OpTy,    f: Formula)           extends  Monitor
    case class  Upto    (a: Monitor,   b: Monitor)           extends  Monitor
    case class  Thru    (a: Monitor,   b: Monitor)           extends  Monitor
    case class  Then    (a: Monitor,   b: Monitor)           extends  Monitor
    case class  And     (a: Monitor,   b: Monitor)           extends  Monitor
    case class  Iterate(a: Monitor,   b: Monitor)           extends  Monitor
    case class  Project(a: Monitor, b: Monitor, p: Monitor) extends  Monitor
    case class  With    (a: Monitor, opt: OpTy, f: Formula) extends  Monitor
  }
  object Runtime {
    case class RTM(a: Abstr.Monitor, name: String, system: ActorSystem)
      def set[T](v: Var[T], a: T): RTM
      def get[T](v: Var[T]): T
      def stop
      def verify: Reply
  }
}
```

Figure 5.3: Overview of `Monitor.scala`

output terminal for visual analysis/debugging. `Fail`, `Done`, and `More` represent the three judgements that can be reported by a monitor following a `Step`.

Lines 21-44 define the ITL-Monitor operators, both infix and prefix, and their translations into the internal data structure defined on lines 46-54. Like the ITL expressions and formulae, this tree structure represents a deep-embedded DSL.

Object `OptimisationFlags` on lines 11-18 defines an optimisation type `OpTy` and a set of subclass objects, each representing a different form of evaluation. On page 77 optimisations of **WITH** were defined as derived monitors. It was noted that more efficient implementations could be provided by exploiting their specific behaviours. The effect of each optimisation flag is explained below:

ANY_STATE – used by `With` to implement `g.SOMETIME(w)` (line 32) where `w` is a state formula. The formula is checked when each new state is received. If any state satisfies `w` then the formula $\diamond w$ has succeeded and the monitor can simply become `g`.

This flag is also passed to `First` in the implemention of `HALT(w)` (line 41). This monitor succeeds as soon as a state is received that satisfies the formula. With this optimisation only the most recent state needs to be inspected obviating the need to store or search an evolving subinterval within the monitor.

ALL_STATES – used by `With` to implement `_.ALWAYS(w)` (line 31). Once again, `w` is a state formula. As each new state arrives, if any fails to satisfy `w` then the monitor fails immediately. As with ANY_STATE, the monitor does not need to maintain an evolving interval.

ANY_PREFIX – used by `First` as the default implemention of `FIRST(f)` (line 36). The semanitcs of $\triangleright f$ require that this monitor must terminate as soon as the accumulated interval satisfies `f`. Therefore, this optimisation flag requires each prefix interval to be examined in turn until the monitor succeeds.

ALL_PREFIXES – used by `With` to implement `g.WITHIN(f)` (line 33). This flag requires every prefix to satisfy the formula $\mathsf{more} \supset (\neg\ f\ ;\ \mathsf{skip})$. It guarantees that `f` does not occur before `g` although the accumulated interval could satisfy both `f` and `g` simultaneously. The monitor fails should `f` be satisfied before `g`.

CHECK_ONCE – used by `With` as the default implemention of `g.WITH(f)`. It maintains the evolving interval but does not need to check that it satisfies `f` until `g` has completed successfully. The implementation of `UNTIL(w1, w2)` (line 44) is an example of this strategy.

Line 57 shows the constructor for a runtime monitor, `RTM`, which requires three parameters: an instance of `Abstr.Monitor` which has been defined using the functions described above;

an identifying name to be used when printing to log files and the standard output stream; an Akka [Akk17] actor system to be used for the underlying implementation. The abstract monitor, `a`, is processed when the `RTM` is created, and transformed into its corresponding concrete implementation. This is discussed in more detail in the Section 5.2.

The four methods on lines 58-61 define how the runtime monitor interacts with the program being verified. `set` updates the monitor with a new variable/value pair. The monitor can collect multiple updates to the state before performing the next analysis. The reason it returns an `RTM` is because the method returns a reference to itself which allows multiple `set` calls to be chained: for example, `mu.set(B, true).set(C, 'x')`. The method `get` looks up the latest value of a variable stored in the monitor. `stop` terminates a monitor. `verify` causes the monitor to process the latest state and return a judgement.

5.2 Concrete monitors

The abstract monitors described previously (`Abstr.Monitor`) reflect the syntax of ITL-Monitor. Thus, for suitable ITL formulae `f`, `g`, `h` and `p`, a client could construct an abstract monitor such as

```
val m: Abstr.Monitor = (FIRST(f) AND FIRST(g)) UPTO (FIRST(h) WITH p)
```

representing the ITL formula $\triangleright((\triangleright f \wedge \triangleright g) \vee (\triangleright h \wedge p))$. This would be transformed (using `o1`, `o2` etc. to represent the appropriate optimisation flags) into the data structure:

```
Upto(And(First(o1, f), First(o2, g)), With(First(o3, h), o4, p))
```

Unlike the ITL formulae representations, this data structure is not evaluated directly. Rather, it is translated into a network of Akka [Akk17] actors. The parent-child relationships in the abstract syntax tree are reflected by communication channels between the corresponding actors. The resulting actor network forms the concrete realisation of the ITL-Monitor expresssion.

Figure 5.4 shows the principal components within `Monitor.scala` that implement concrete monitors (`Concr.Monitor`). This is a skeleton view and extraneous detail has been omitted. (The full code appears in Listing A.2).

5.2.1 Actor initialisation

Figure 5.4 (line 5) defines a concrete `Monitor` as an actor. Each of the classes shown on lines 8-14 are actors representing a node in the abstract monitor syntax tree. The `startUp` method

```
1  private object Concr {
2    import Protocol._
3    import OptimisationFlags._
4
5    abstract class Monitor extends Actor with ActorLogging
6
7    def startUp(mu: Abstr.Monitor, context: ActorContext): ActorRef
8    case class First(name: String, opt: OpTy, f: Formula) extends Monitor
9    case class With(name: String, a: Abstr.Monitor, opt:OpTy, f: Formula) extends Monitor
10   case class Upto(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor
11   case class Thru(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor
12   case class And(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor
13   case class Then(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor
14   case class Iterate(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor
15 }
```

Figure 5.4: Overview of concrete monitor implementation

on line 7 is responsible for initiating a concrete monitor (actor) associated with an abstract monitor. Calling `startUp` with an abstract monitor `a` creates a corresponding concrete actor `c`, which, if `a` has children, calls `startUp` for each child.

The steps in the creation of a runtime monitor for the formula ((**FIRST** (f) **AND** **FIRST** (g)) **UPTO** (**FIRST** (h) **WITH** (p)) are shown below. For clarity, only the monitor and formula parameters are given.

1. Within the program define the ITL formulae `f`, `g`, `h`, and `p`.

2. Define an ITL-Monitor:
   `val a = (FIRST(f) AND FIRST(g)) UPTO (FIRST(h) WITH (p))`

3. The following chain of events is depicted in Figure 5.5.

   (i) Define a runtime monitor (assuming a suitable Akka system, `as`, is in scope):
   `val mu: Runtime.RTM = RTM(a, "monitor name", as)`

   (ii) Initialisation of `mu` creates an `RTMActor(a)`

   (iii) `RTMActor(a)` initiates `Upto(And(First(f), First(g)), With(First(h), p))`

   (iv) `Upto(And(First(f), First(g)), With(First(h), p))`
   initiates `And(First(f), First(g))`

   (v) `And(First(f), First(g))` initiates `First(f)`

   (vi) `And(First(f), First(g))` initiates `First(g)`

   (vii) `Upto(And(First(f), First(g)), With(First(h), p))`
   initiates `With(First(h), p)`

   (viii) `With(First(h), p)` initiates `First(h)`

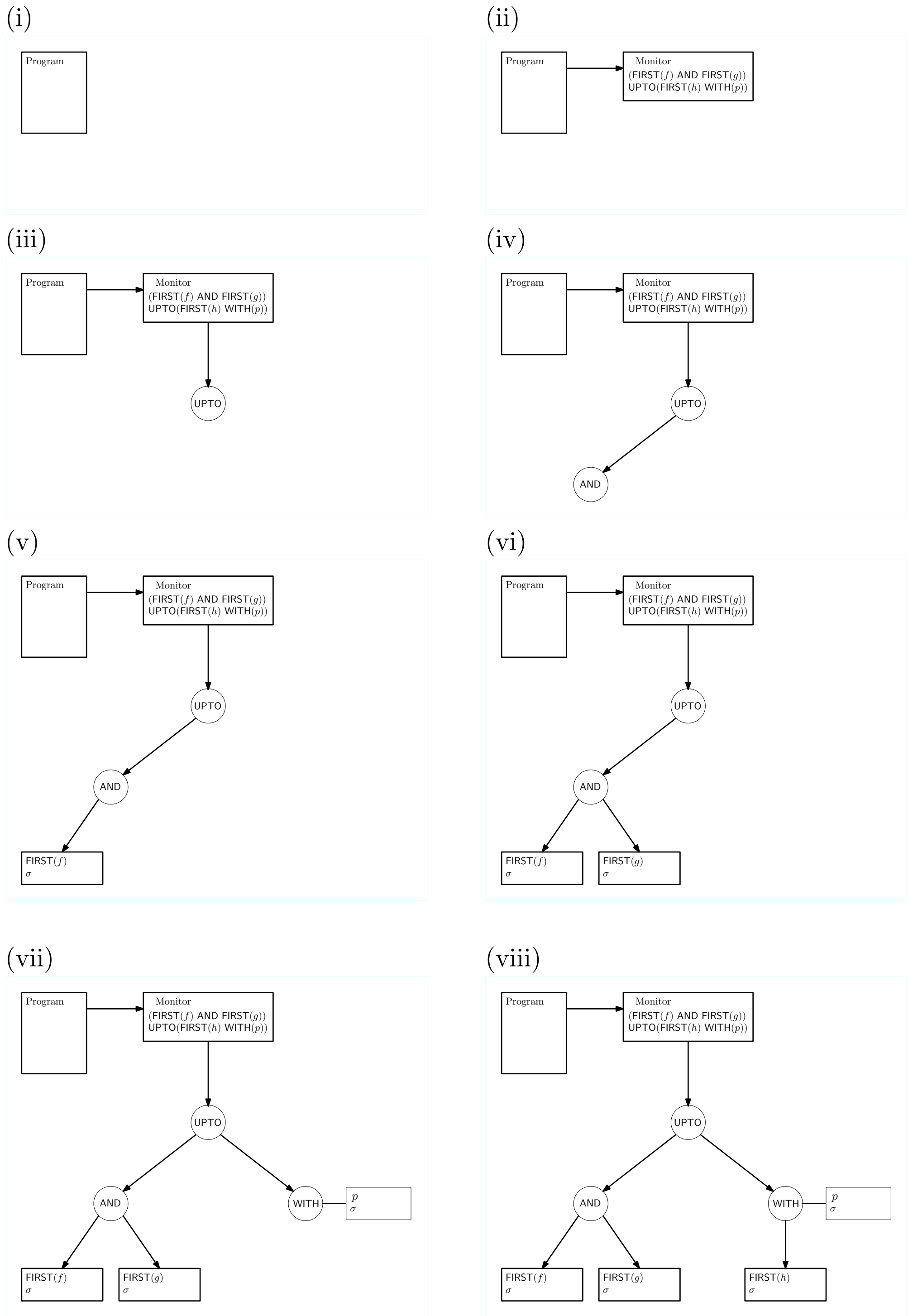


Figure 5.5: Initialisation of a concrete monitor

The behaviour of each actor is defined using an Akka *receive block* – an instance of a Scala partial function, `PartialFunction[Any, Unit]`. The domain of the partial function is `Any`, the root of Scala's typeclass system, which allows any kind of message to be passed. This is slightly dangerous because the communication channels are untyped.[3] Only instances of `Request` and `Reply` (Listing 5.3 lines 2-10) should be sent between monitors. Pattern matching associates each received message with a corresponding action. The full listing of `Monitor.scala` (Appendix A Listing A.2) shows the concrete monitor implementations and how each actor's behaviour is defined by a receive block.

5.2.2 Runtime monitoring

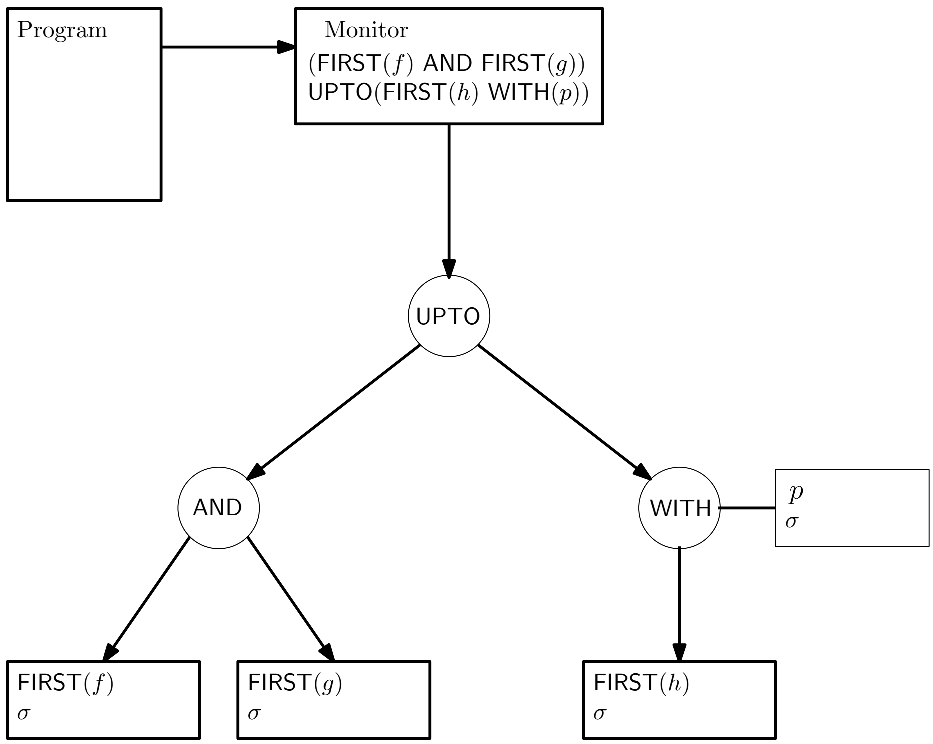
Once the actor network implementing a monitor has been initiated it receives `Step` messages from the program being verified, and can deliver verdicts (cf. Figure 2.15). The communication between the program, the monitor (actor), and each of the other actors uses a synchronous protocol. Normally, within an actor network, the communication channels are used asynchronously to avoid blocking. However, ITL-Monitor has been designed to execute in lock-step with the program being verified. Thus, when the program sends a new state to the monitor, it waits for a response. The decision to use a synchronised communication model was taken to enable the program to 'react at runtime' in response to the verdicts it receives from the monitor.

To see the runtime operation of a monitor, consider the earlier example: (($\mathbf{FIRST}\,(f)$ $\mathbf{AND}$ $\mathbf{FIRST}\,(g)$) $\mathbf{UPTO}$ ($\mathbf{FIRST}\,(h)$ $\mathbf{WITH}$ (p))). An actor representing **UPTO** is created and this actor then spawns *two* children. These, in turn, have their own children. The situation is illustrated in Figure 5.6. The architecture exhibits a loose form of coupling between the program, the monitor, and the sub-monitors. The program and monitor communicate using message passing.

Figure 5.7 illustrates what happens when the program generates the first state. It is passed on to the monitor which, in turn, passes it down through each of the nodes, replicating the state at each fork, until a copy of the state reaches each terminal node in the tree. This illustrates how each terminal node maintains its own copy of the evolving interval. The terminal nodes represent monitors such as **FIRST** which need access to a copy of the current subinterval in order to judge whether or not it satisfies the given formula.

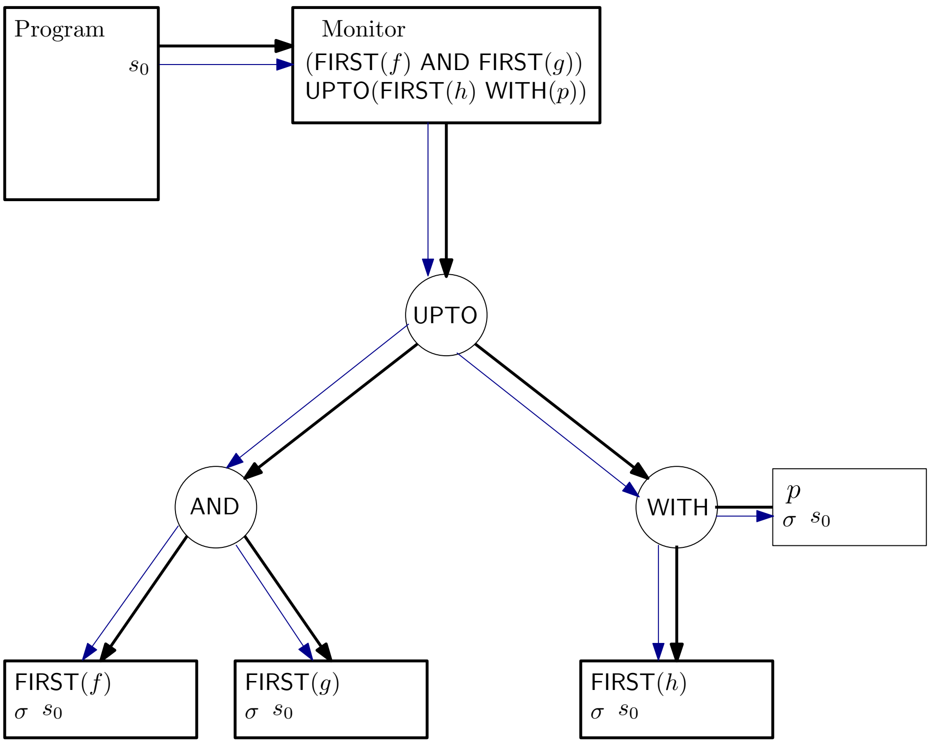
The design requires that these terminal nodes share an evolving subinterval. Any particular implementation can decide how to achieve this depending upon the functionality required. For example, references to a shared state could be used, to minimise duplication. Alternatively, duplication could be used precisely to avoid shared, mutable state. Duplication provides for

[3] This is the definition in the Akka API.



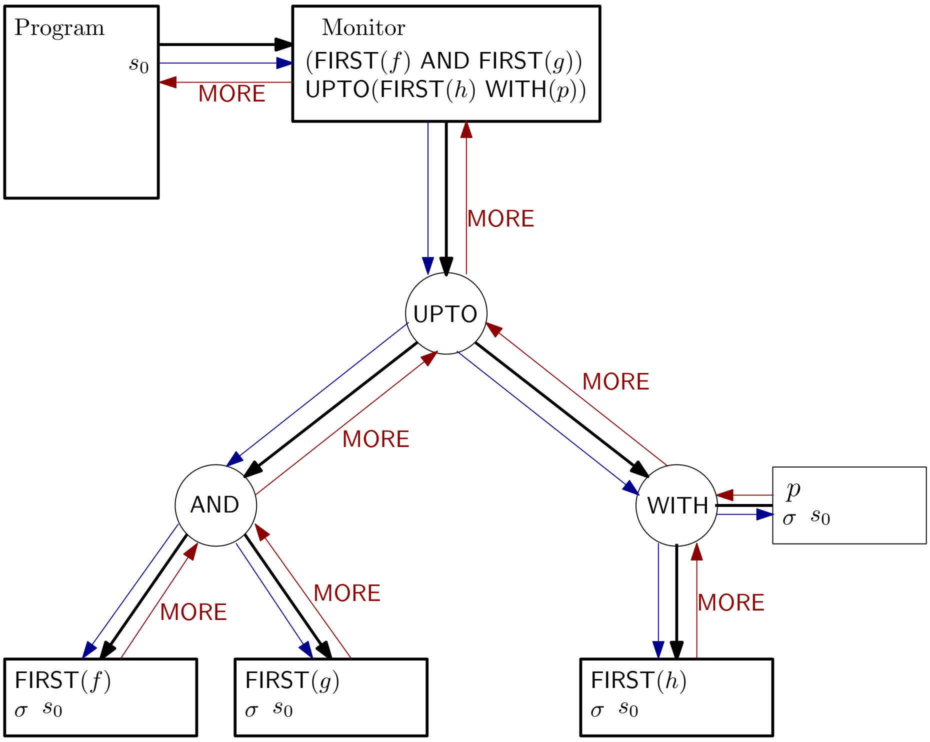
Program, monitor, and the concrete actors representing ((**FIRST** (f) **AND** **FIRST** (g)) **UPTO** (**FIRST** (h) **WITH** (p)). The picture shows the state at the point monitoring begins.

Figure 5.6: Monitoring-1:



The program generates the first states which is duplicated as it passes down the tree. A copy of the state is stored in each terminal node.

Figure 5.7: Monitoring-2



Each terminal node (monitor) makes its judgement on the interval consisting of the first state. These judgements are reported up the tree to the parent nodes which interpret the results and report their own judgements according to their behaviour.

Figure 5.8: Monitoring-3

a local copy within each monitor which minimises state-lookup overheads if parallel monitor components were to be distributed. The Scala implementation used in this thesis adopts the latter strategy.

Figure 5.8 demonstrates the first judgement that the monitor returns to the program following the generation of the first state. The dotted lines on the right indicate the reports passed back 'up' the process hierarchy. Each message is a judgement on the interval *so far*. In this example p might be a state formula which is satisfied in the first state. MORE indicates that the monitor has not reached a final judgement and is asking for more states to be provided. Figure 5.9 presents a table of the message values that can be returned by each monitor following the introduction of each new state.

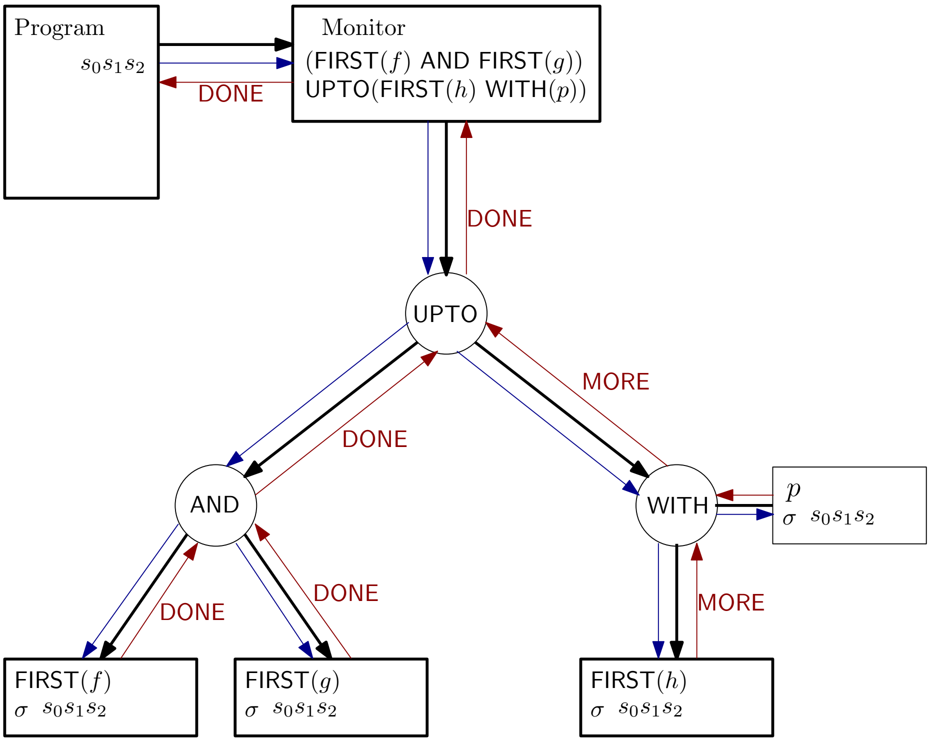
The monitor response protocol includes DONE, FAIL, and MORE, representing three possible judgements that can be returned to a program. DONE informs the program that its runtime verification has successfully terminated: the program may continue beyond this point but no further verification will occur. FAIL informs the program of a verification failure and an error code will also be communicated to indicate the nature of the non-compliance. MORE

Message	Description	Judgement	Readiness
DONE	Verification success	Final	Will accept no more states
MORE	Cannot anticipate judgement	Inconclusive	Requires more states
FAIL	Verification failure	Final	Will accept no more states

Figure 5.9: Monitor-response messages

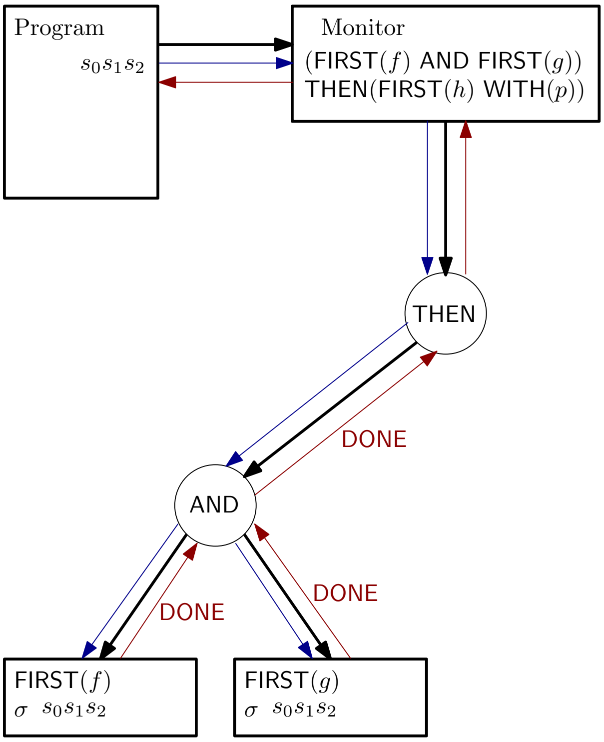
informs the program that the monitor has insufficient data to make a final judgement at this stage and, remaining *impartial* it requests more state(s). This implements a *three-valued* logic response [LS09].

In Figure 5.10 the situation arises when a final judgement can be made and the monitoring is completed. The system is shown after three states have been generated by the program. The monitor expressions **FIRST** (f) and **FIRST** (g) have completed successfully. Therefore the expression **FIRST** (f) **AND** **FIRST** (g) has completed successfully and reports DONE. On the right-hand side of the expression tree the monitors have not completed and are requesting more states. However, the semantics of **UPTO** is that it succeeds if, and as soon as, either of its operands succeeds and, since this is the case, the **UPTO** node reports DONE to the main monitor which, in turn, is relayed to the main program.



A final judgement can be made. The left-hand part of the tree succeeds. The right-hand part has not yet delivered a judgement. However, **UPTO** requires only one side to succeed, and can therefore pass DONE to its parent.

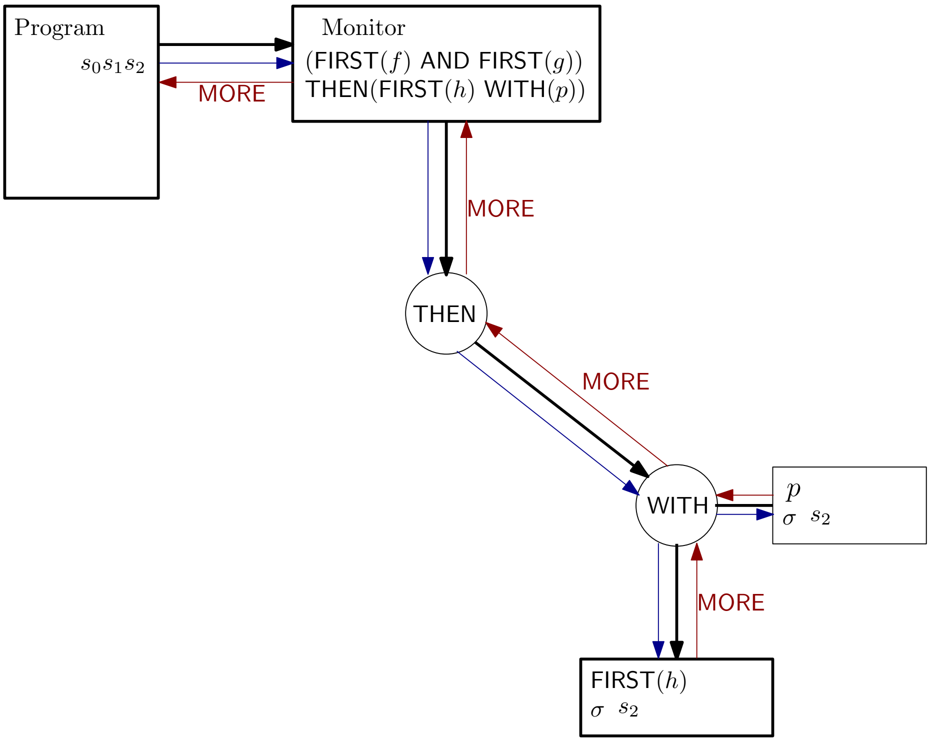
Figure 5.10: Monitoring-4



After three states the **WITH** monitor is able to report success. The **THEN** monitor must discard the old subtree and create a new one based on the right-hand subformula before it can complete its analysis.

Figure 5.11: Monitoring-5

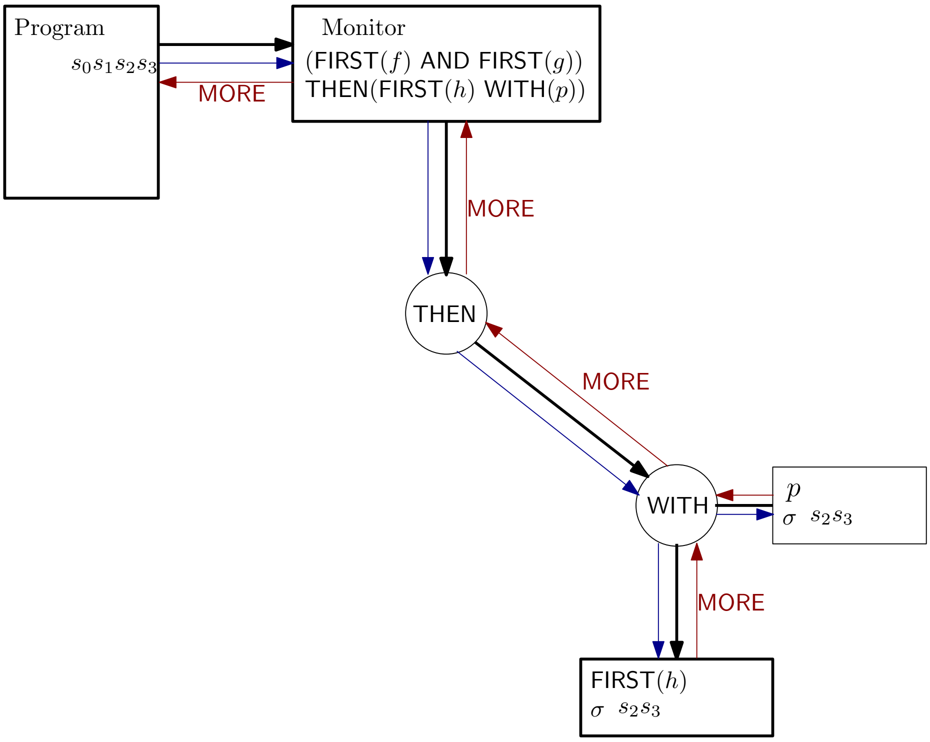
In contrast to the previous example, consider the monitor ((**FIRST** (f) **AND FIRST** (g)) **THEN** (**FIRST** (h) **WITH** (p)). In this case the top-most combinator has been changed from a parallel combinator, **UPTO**, to a sequential one, **THEN**. Figure 5.11 shows the state *part-way* through the analysis performed by the node **THEN**. The left-hand branch has just reported successful termination. However, fusion shares a state: the last state of the previous interval becomes the first state of the new interval. At this stage, therefore, **THEN** can discard its left subtree since its purpose has ended, and create a new right subtree based on the right-hand part of the formula. The newly evolved state is shown in Figure 5.12. The **THEN** operator represents a *dynamic transition* in that the shape of the graph changes when the transition from left child to right child occurs.



The newly-created right subtree is passed the *previous final state* to become its first state. The analysis is now completed.

Figure 5.12: Monitoring-6

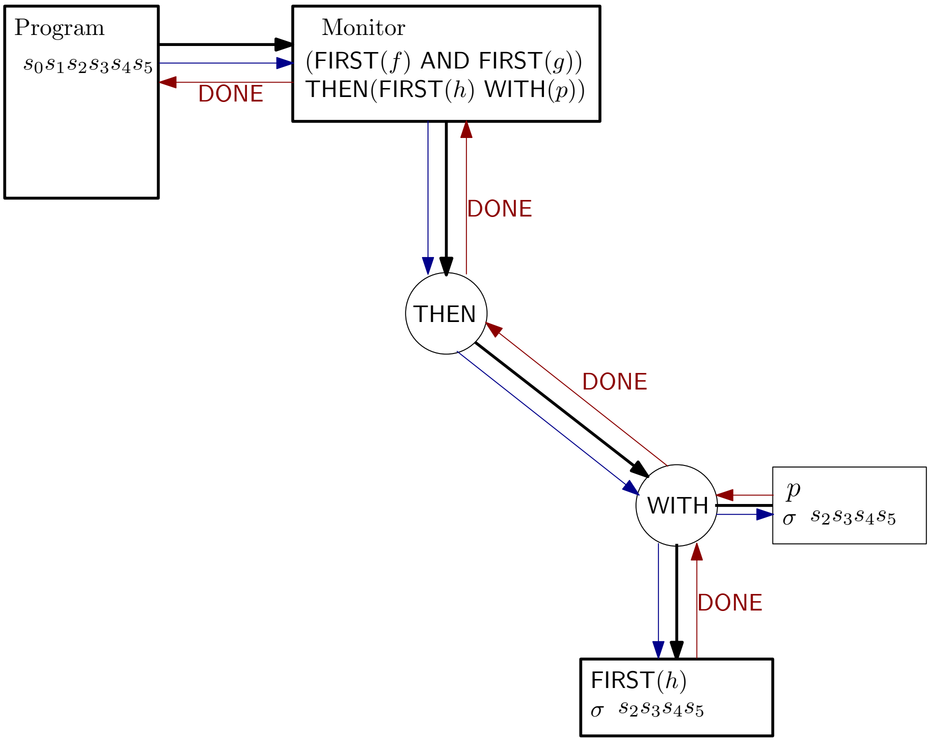
Figure 5.12 shows the system following the pruning of the now-terminated left sub-tree and the creation of the new right subtree. The newly created node, **WITH**, has been forwarded the *current state* – the state that terminated the previous subinterval – which becomes the first state of the new interval. This has been cascaded down to the terminal nodes and a suitable judgement has been reported back. At this stage no conclusion can be reached so the nodes request more information (another state). The next two figures will demonstrate how the system could evolve.



This shows the system after four states. The overall monitor remains impartial and anticipatory.

Figure 5.13: Monitoring-7

Figure 5.13 illustrates the generation of a fourth state, s_3. The full interval, $\langle s_0, s_1, s_2, s_3 \rangle$, is drawn inside the program to illustrate its history. However, see how the intervals stored in the terminal nodes are only $\langle s_2, s_3 \rangle$. This demonstrates that the terminal nodes store only the states relevant to their subinterval.



This shows the system after five states. The current network is concerned with only the suffix interval which, in this case, satisfies its components: $\langle s_2, s_3, s_4 \rangle \models \rhd h \wedge p$.

Figure 5.14: Monitoring-8

Finally, in Figure 5.14, the introduction of the fifth state, s_4, creates an interval that satisfies **WITH**: $\langle s_2, s_3, s_4 \rangle \models \rhd h \wedge p$. The judgements are reported back to the program. Following this judgement the monitor can be discarded, and garbage-collected, as it has completed its function. If, at this stage, the program has not completed, then a new monitor may be created according to a new formula and runtime verification can continue. This demonstrates the dynamic behaviour of the monitors.

The example illustrates how the program's behaviour satisfied the formula $(\rhd f \ \wedge \rhd g) ; (\rhd h \ \wedge \ p)$. Thus $\langle s_0, s_1, s_2 \rangle \models \rhd f \ \wedge \rhd g$ and $\langle s_2, s_3, s_4 \rangle \models \rhd h \ \wedge \ p$. The program also satisfies the more general formula: $(f \wedge g) ; (h \wedge p)$:

1	$\vdash \rhd f \wedge \rhd g \supset f \wedge g$	*FstAndElimL*$^{(C.264)}$, logic
2	$\vdash \rhd h \wedge p \supset h \wedge p$	*FstAndElimL*$^{(C.264)}$, logic
3	$\vdash (\rhd f \wedge \rhd g) ; (\rhd h \wedge p) \supset (f \wedge g) ; (h \wedge p)$	1, 2, *LeftChopImpChop*$^{(C.101)}$, *RightChopImpChop*$^{(C.102)}$

5.3 Summary

The chapter introduced the practical realisation of ITL-Monitor as a Scala API. The main components for building monitors were explained: `ITL` for constructing ITL formulae, and `Monitor` for constructing ITL-Monitors and executing them. Although the libraries have not been extensively optimised, certain efficiency measures have been taken in respect of determining fusion points when a formula specifices an interval of a predefined length; and adjusting the evaluation strategies of certain derived monitors to take advantage of their semantics.

The use of Akka actors to implement the concrete monitors provides the potential to exploit multiple cores which ameliorates the impact of an inline, runtime monitoring system which shares resources with the program under test. Furthermore, the distribution of actors across available cores is a task that is handled by the Akka dispatcher and not the monitor implementation itself.

Chapter 6

Examples and evaluation

6.1 Introduction

In this chapter two example scenarios are presented and analysed.

The first (latch) example (Section 6.2) is specified and verified using four different approaches and three different runtime verification tools. The tools compared are TRACECONTRACT [BH11, Hav19], a runtime monitoring system developed as a Domain Specific Language in Scala; AnaTempura [Mos96b], an established runtime verification system for ITL; and ITL-Monitor, the monitoring system developed in this thesis.

The example illustrates the use of each of these systems and provides a comparative performance analysis. To facilitate a fair comparison, the example utilises a specification that can be expressed in LTL, Tempura (a subset of ITL), and ITL.

The second (checkout) example (Section 6.3) concentrates predominantly on ITL-Monitor. Its purpose is to demonstrate the performance when monitoring a significantly more complex system capable of generating large execution traces. In this example, two of the temporal requirements are also adapted for use with TRACECONTRACT for comparison with ITL-Monitor.

All experiments were run using `sbt` [sbt19] on a Macbook Pro with 2.6GHz Intel Core i5; OSX 10.13.6; Scala version 2.12.7; Akka version 2.5.19.

6.2 Latch example

A program is written in Scala to simulate the operation of system governed by the relative temporal behaviour of three Boolean flags. A set of requirements is developed for the system and verified in four ways:

1. Using ITL-Monitor, the *inline* runtime verification system proposed in this thesis;
2. Using AnaTempura (Section 2.4.5), an *outline* runtime verification system based upon a subset of ITL;
3. Using TraceContract [Hav19, BH11], an *inline* runtime verification system also written as a Scala DSL, with two specification styles:
   (a) Future time LTL
   (b) State machines

The different styles of specification and runtime verification will then be compared both qualitatively and quantitatively. The latter will include an analysis of the runtime results and their relative timings. The full code for the latch example is provided in Section B.2 from which relevant extracts are presented below as required.

Consider a system which consists of two latches, A and B, and a signal, S. The behaviour is defined informally as follows. When latch A is down then B must remain stable up to and including the first moment when latch A is up. When latch A is up then B is free to switch between up and down states. Every time a state change in B occurs then a signal S is raised just at the point that B's state changes. Unless B changes state in the very next moment then S returns to its down state. Thus A is used to enable B's switching behaviour, and S signals every state change in B. The diagram in Figure 6.1 illustrates a prefix of some example simulation run.

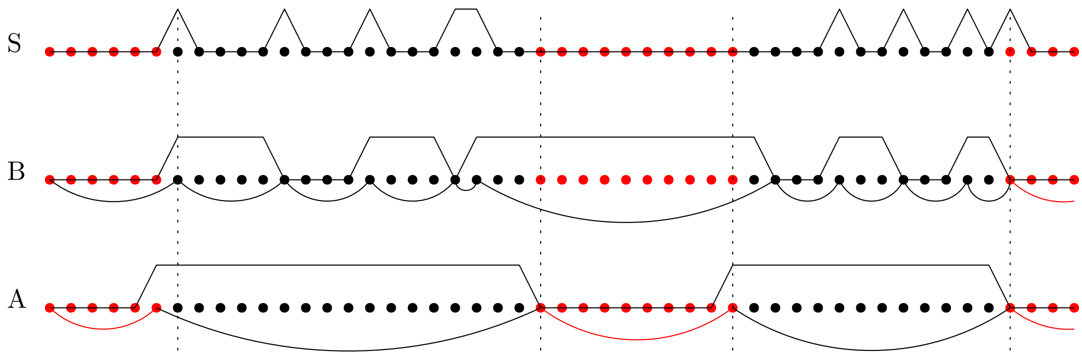


Figure 6.1: Latch example

The set of system requirements is given:

R1 Whenever B is stable in two successive states then the signal S is low in the second of those states.

R2 Whenever B is not stable in two successive states then the signal S is high in the second of those states: i.e. any change to B is signalled by S.

R3 Whenever A is low in two successive states then B is stable across those states.

R4 Whenever A is raised across two successive states then B is stable across those states.

6.2.1 Description in ITL

Consider a single cycle of A from down, through up, and then to down again. This cycle can be described by the ITL formula $\mathsf{halt}(A)\ ;\ \mathsf{halt}(\neg\, A)$. The formula $\mathsf{halt}(A)$ specifies that $\neg\, A$ holds in all states except the final state whereupon A holds. $\mathsf{halt}(\neg\, A)$ specifies the inverse situation. Furthermore, until A holds, B is required to be stable. These conditions can be combined into the following formula describing a cycle of A.

$$(\mathsf{halt}(A) \wedge \mathsf{stable}(B))\ ;\ (\mathsf{halt}(\neg\, A)) \tag{1}$$

To specify finitely many A-cycles, the formula can be repeated:

$$(\ (\mathsf{halt}(A) \wedge \mathsf{stable}(B))\ ;\ (\mathsf{halt}(\neg\, A))\ )^*$$

In a similar way the latching behaviour of B can be specified. Every change in state by B must be accompanied by the raising of the signal S. Each cycle of B consists of a series of states in which B remains stable and a final state in which B's value changes.

$$(B \lessdot\!\!\sim \neg\, B) \wedge (\mathsf{skip}\ ;\ \mathsf{halt}(S)) \tag{2}$$

The operator $\lessdot\!\!\sim$ is padded temporal assignment. The formula $B \lessdot\!\!\sim \neg\, B$ specifies that B is stable until the final state, at which point it changes. The formula $\mathsf{skip}\ ;\ \mathsf{halt}(S)$ specifies that S is low from the second state and raised at the end. Skipping the initial state is necessary because the initial state of each B-cycle (except the first B-cycle) coincides with the final state of the previous B-cycle and in each of these states S holds. The repetition of finitely many B-cycles is given by:

$$(\,(B \lessdot\!\!\sim \neg\, B) \wedge (\mathsf{skip}\ ;\ \mathsf{halt}(S))\,)^*$$

Assume that some terminating condition $STOP$ is specified to align with the end of an A-cycle and a B-cycle, i.e.

$$\mathsf{halt}(STOP) \supset \Diamond(\mathsf{skip} \wedge A \wedge \bigcirc(\neg\, A) \wedge \bigcirc(B) = \neg\, B)$$

$\mathsf{halt}(STOP)$ specifies the interval in which $STOP$ holds only in the final state. This defines the extent of the simulation. When $\mathsf{halt}(STOP)$ holds then there must be a two-state suffix subinterval $(\mathsf{skip} \wedge \ldots)$ which satisfies the final transitions of an A and B-cycle respectively.

Every variable is false in the initial state. The system can be specified as follows:

$$
\begin{array}{ll}
(\mathsf{empty} \wedge \neg A \wedge \neg B \wedge \neg S) \;; & \text{Initial state} \\
(\;\mathsf{halt}(STOP) & \text{Termination condition} \\
\quad \wedge \,(\,(B \lessapprox \neg B) \wedge (\mathsf{skip} \;;\; \mathsf{halt}(S))\,)^* & (1)^* \\
\quad \wedge \,(\,(\mathsf{halt}(A) \wedge \mathsf{stable}(B)) \;;\; (\mathsf{halt}(\neg A))\,)^* & (2)^* \\
) &
\end{array}
$$

The four requirements $(R1 - R4)$ can be derived from this ITL specification. The derivations are presented in B.2.1. Note that $\mathsf{keep}\, f$ requires f to hold over all unit subintervals – i.e. over all subintervals that consist of precisely *two* states.

$$
\begin{array}{ll}
R1: & \mathsf{keep}(\;(\bigcirc(B) \equiv B) \supset \bigcirc(\neg S)\;) \\
R2: & \mathsf{keep}(\;(\bigcirc(B) \not\equiv B) \supset \bigcirc(S)\;) \\
R3: & \mathsf{keep}((\neg A \wedge \bigcirc(\neg A) \supset (B = \bigcirc(B))) \\
R4: & \mathsf{keep}((\neg A \wedge \bigcirc(A) \supset (B = \bigcirc B))
\end{array}
$$

Noting that $(B \lessapprox \neg B) \equiv \triangleright(B \lessapprox \neg B)$, the above ITL specification can be translated into an ITL-Monitor formula:

$$
\begin{array}{lll}
m \;\widehat{=}\; \textbf{GUARD}\,(\neg A \wedge \neg B \wedge \neg S) & & \\
\quad \textbf{THEN} \;\; (& \textbf{HALT}\,(STOP) & \\
 & \textbf{ITERATE} & (\,\textbf{FIRST}\,(B \lessapprox \neg B)\;\textbf{WITH}\;(\mathsf{skip} \;;\; \mathsf{halt}(S))\,) \\
 & \textbf{ITERATE} & (\,(\textbf{HALT}\,(A)\;\textbf{WITH}\;(\mathsf{stable}(B)))\;\textbf{THEN HALT}\,(\neg A)\,) \\
\quad\quad\quad) & &
\end{array}
$$

The structure of this specification exhibits a *managed halt* patern (Section 3.4.2).

6.2.2 Properties expressed in Tempura

The four requirements $(R1-R4)$ can also be written in Tempura for analysis with AnaTempura. As with the LTL specification in the following subsection, the requirements do not require the use of an iteration construct (chopstar). Each of the requirements is expressed as a formula that applies over *all unit intervals*. In Tempura this is achieved using the `keep` operator.

```
define R1(B,S) = { keep( ((next B) = B) implies not next(S) ) }.
define R2(B,S) = { keep( ((next B) = not B) implies next(S) ) }.
define R3(A,B) = { keep( (not A and not next(A)) implies (B = (next B))) }.
```

```
define R4(A,B) = { keep( (not A and next(A)) implies (B = (next B))) }.
```

6.2.3 Properties expressed in LTL

Unlike ITL, LTL does not have an iteration construct and therefore the original ITL specification cannot be translated directly. However, the four requirements (R1 - R4) can be expressed in LTL using the form $\Box(\ldots)$. Each requirement is a formula that holds over pairs of successive states. *Weak-next* is used instead of *strong-next* because the intervals are finite (Section 2.1.2).

$R1$	$\Box((B \Leftrightarrow \textcircled{w}\ (B)) \Rightarrow \textcircled{w}\ (\neg\ S))$
$R2$	$\Box(\neg\ (B \Leftrightarrow \textcircled{w}\ (B)) \Rightarrow \textcircled{w}\ (S))$
$R3$	$\Box((\neg\ A \wedge \textcircled{w}\ (\neg\ A)) \Rightarrow (B \Leftrightarrow \textcircled{w}\ (B))$
$R4$	$\Box((\neg\ A \wedge \textcircled{w}\ (A)) \Rightarrow (B \Leftrightarrow \textcircled{w}\ (B)))$

6.2.4 State machine

The behaviour of the latch example can also be expressed as a deterministic, finite state machine (Figure 6.2). Each transition is implicitly labelled with a 3-tuple consisting of the *next* state of the three variables (A, B, S). The nodes represent each of the eight possible states. In the initial state, s_0, all three flags are down. The system can terminate in either of the states s_1 or s_3 – each of these represents a situation in which a final B-transition has occurred.

6.2.5 Simulation and runtime verification

The latch simulation is written as a Scala program and the full listing appears in Appendix B in two parts as Listing B.4 and Listing B.5. The simulation has been written in such a way that it can be executed using any combination of the following monitoring options:

- Using ITL-Monitor with the ITL specification
- Using AnaTempura with the Tempura specification
- Using TraceContract with the LTL specification
- Using TraceContract with the state machine specification

It is also possible to run the simulation with no monitoring at all. This provides a baseline for performance measurement.

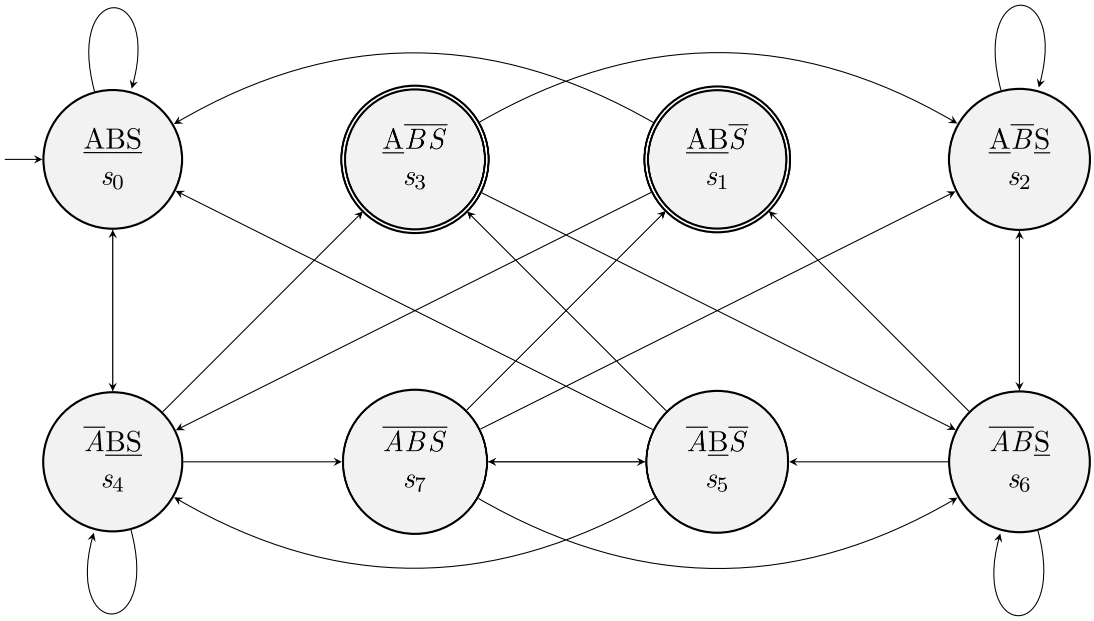


Figure 6.2: Finite state machine for the latch example

Both ITL-Monitor and TraceContract are embedded domain specific libraries written in Scala. In both cases their respective monitors are created dynamically as Scala objects within the simulation and therefore perform *inline monitoring*. Instrumenting the simulation to communicate with these monitors entails internal method calls. In contrast to these, AnaTempura runs externally to the simulation and thus performs *outline monitoring*.[1] In this case the instrumentation is handled by sending specially encoded ASCII messages to `stdout`.

To ensure that the same state data is communicated to all running monitors, the instrumentation for each system is abstracted into a single `verify()` method:

```
def verify() {
  if (runAna)
    println("!PROG: assert Event:"+mu.get(A)+":"+mu.get(B)+":"+mu.get(S)+":!");
  if (runLTL || runStM) nu.verify(TC.Event(mu.get(A), mu.get(B), mu.get(S)))
  if (runITM) mu.!!
  numOfStates = numOfStates + 1
}
```

Each monitoring system is *guarded* by its own flag (`runAna`, `runITL`, `runITM`) and these flags can be set in any combination when the simulation is run.

[1] Inline and outline monitoring was discussed in Section 2.4.

6.2.5.1 ITL monitoring

The simulation contains a mixture of unmonitored and monitored variables. The former are simply standard Scala variables. The latter are represented using the type `Var` from the ITL-Monitor API (cf. Section 5.1.1.1). The three flags are declared thus:

```
object S extends Var[Boolean] { override def toString = "S" }
object A extends Var[Boolean] { override def toString = "A" }
object B extends Var[Boolean] { override def toString = "B" }
```

The ITL specification (6.2.1) is translated into ITL-Monitor. It has been split into subclauses for ease of reading.

```
val initial = (~A and ~B and ~S)
val clause2 = FIRST(B <~ ~B) WITH (skip ';' halt(S))
val clause3 = (HALT(A) WITH (stable(B))) THEN (HALT(~A))
val spec    = (GUARD(initial) THEN (HALT(STOP) ITERATE clause2 ITERATE clause3))
```

A monitor, `mu`, is created to perform a runtime verification using this specification. The constructor requires a name field, and a reference to an Akka actor system (`as`).

```
val mu = RTM(spec, "Latch", as)
```

`mu` encapsulates the state of the monitored variables which can be accessed via *get* and *set* methods. For example, this shows how to assign $B := \neg\, B$ and $S :=$ true:

```
mu.set(B,!mu.get(B)).set(S,true)
```

When an assertion point is reached and the current state is to be added to the interval for analysis, this instruction can be made by the following method call.

```
mu.!!
```

This can result in one of three outcomes:

1. MORE is returned – the verification is inconclusive and more state(s) are required. This is the normal situation while the simulation is being verified, before a definitive judgement can be made.

2. DONE is returned – the execution trace satisfies the formula.

3. An exception is thrown – the execution trace cannot satisfy the formula.

If `mu.!!` occurs within a `try` block then a failure can be handled using a corresponding `catch` block. The simulation demonstrates the potential of this approach.

```
catch {
  case e: RTM.RTVException =>
    println(e)                          // ITM detected a violation
    println("React at Runtime...")      // Alternative action goes here
}
```

6.2.5.2 AnaTempura monitoring

AnaTempura invokes the simulation but both run as separate programs. The specification is contained within a special Tempura file `latch.t` (Listing 6.1).[2]

Listing 6.1: AnaTempura specification

```
load "conversion".
load "exprog".

/* sbt demo.latch.Simulation 2 t off 1 Latch */

set print_states = true.

define get_var(A,B,S,Z) = {
    exists T : {
           get2(T) and
           A = if T[1]="true" then true else false and
           B = if T[2]="true" then true else false and
           S = if T[3]="true" then true else false and
           Z = if T[4]="true" then true else false
     }
}.

define Pass(R) = format("-- Pass R%d\n",R).
define Fail(R) = format("** Fail R%d\n",R).

define R1(B,S) = { keep if ((next B = B) implies (not next S))
                        then Pass(1) else Fail(1)
                 }.
define R2(B,S) = { keep if ((next B = not B) implies (next S))
                        then Pass(2) else Fail(2)
                 }.
define R3(A,B) = { keep if ((not A and not next(A)) implies (B = next B))
                        then Pass(3) else Fail(3)
                 }.
define R4(A,B) = { keep if ((not A and next(A)) implies (B = next B))
                        then Pass(4) else Fail(4)
                 }.
```

[2]Antonio Cau adapted AnaTempura (version 3.5) to run Scala programs within `sbt` (The simple build tool – a command line development environment for Scala projects). The current latch example was the catalyst for this development and the original `latch.t` code was provided by A Cau. The author and A Cau co-developed the file to work with `latch.scala` and AnaTempura.

```
35 /* run */ define test() = {
36  exists A,B,S,Z: {
37    get_var(A,B,S,Z) and
38    format("A=%t, B=%t, S=%t, STOP=%t\n",A,B,S,Z) and
39    keep (get_var(next A,next B,next S,next Z)) and
40    keep format("A=%t, B=%t, S=%t, STOP=%t\n",next A,next B,next S,next Z) and
41    halt(Z) and
42    R1(B,S) and
43    R2(B,S) and
44    R3(A,B) and
45    R4(A,B)
46  }
47 }.
```

Listing 6.1 contains the following features:

line 5 This comment specifies how AnaTempura should call the Scala simulation. It states that the simulation is run within `sbt`, Scala's terminal-based development environment. The simulation path and command line arguments are provided as an `sbt` batch command.

lines 9-17 The `get_var` function parses the string passed to AnaTempura from the simulation for each state. Each state is passed as a string such as:

```
!PROG: assert Event:true:false:true:false:!
```

The components of the string are parsed into `T` where, in this example, `T[1]`, `T[2]`, `T[3]`, and `T[4]` are `true`, `false`, `true`, and `false` respectively. These values are assigned to the state variables `A`, `B`, `S`, and `Z`.

lines 19-20 These definitions specify the messages to be output when monitor R succeeds or fails. The function `format(...)` returns `true` so the formula itself always succeeds. The printed message indicates the truth value associated with R.

lines 22-33 The requirements R1-R4 are expressed as function definitions. Note that in each case the condition `keep cond` has been expressed as (for R1) `keep if cond then Pass(1) else Fail(1)`. This construction allows for the test itself to succeed whether or not `cond` passes or fails. However, the appropriate report message is printed on the output transcript.

lines 35-46 This is the main function `test` which is executed from within AnaTempura.

lines 37 and 38 Read and write the first state.

lines 40 and 41 Read and write all subsequent states.

lines 41-45 The extent of the finite interval is defined using the variable `Z` which becomes `true` when a `STOP` event is received. The remaining lines conjoin the four requirements.

6.2.5.3 TraceContract monitoring

The code required to specify the LTL and state machine specifications in TraceContract is relatively lengthy compared to the ITL-Monitor specification above. Therefore, this code is encapsulated into a separate object, `TC`, (see Listing B.5). These specifications will be discussed further below, but first the construction of a TraceContract monitor, `nu`, as provided within the main simulation program is shown:

```
val nu = if (runLTL && runStM) TC.monitorAll
         else if (runLTL)       TC.monitorLTL
         else if (runStM)       TC.monitorStM
         else                   TC.monitorNil
```

The simulation sets up `nu` to run one of four combinations of TraceContract verification: just LTL, just the state machine, both, or neither. The flags controlling the simulation are derived from command line arguments when the simulation is initiated. When a new state is ready for analysis it needs to be encoded as an `Event` (a type defined within `TC` that consists of the three Boolean values) and then passed to `nu` using the `verify()` method call:

```
nu.verify(TC.Event(mu.get(A), mu.get(B), mu.get(S)))
```

The interplay between (ITL-Monitor) `mu` and (TraceContract) `nu` is important here. The state values are obtained from `mu` using `mu.get` methods and these are used to construct a `TC.Event` to be passed to monitor `nu`. This occurs whether or not `mu` is used for performing a runtime verification, and ensures that the same states are passed to all runtime monitors being used.

The LTL formulae representing R1-R4 are encoded using the TraceContract API. Firstly, a mumber of named propositions are defined and represented as partial functions. Each is a projection function inspecting a particular component of the `TC.Event` state:

```
def aHi: PartialFunction[Event,Boolean] = { case Event(true, _,_) => true }
def aLo: PartialFunction[Event,Boolean] = { case Event(false,_,_) => true }
def bHi: PartialFunction[Event,Boolean] = { case Event(_,true, _) => true }
def bLo: PartialFunction[Event,Boolean] = { case Event(_,false,_) => true }
def sHi: PartialFunction[Event,Boolean] = { case Event(_,_,true ) => true }
def sLo: PartialFunction[Event,Boolean] = { case Event(_,_,false) => true }
```

Each requirement is written as a subclass of a TraceContract `Monitor` and thus inherits all of the monitor behaviour. The code for R1 is shown below. The function `globally` corresponds to the LTL operator $\Box$. Also note the use of `weaknext` which does not fail when applied in the final state of a finite interval.

Listing 6.2: Definition of R1

```
class R1 extends Monitor[Event] {
  /*
   * If B is stable across two adjacent states then S is low in the 2nd state
   *
   * □((B ⇔ Ⓦ (B)) ⇒ Ⓦ (¬ S))
   */

  def bStable = ((matches{bHi}) and weaknext(matches{bHi})) or
                ((matches{bLo}) and weaknext(matches{bLo}))

  property('R1) {
    globally {
      bStable implies (weaknext(matches{sLo}))
    }
  }
}//R1
```

The remaining requirements are encoded similarly and then combined into a single monitor representing the conjunction of all four requirements. [3]

```
class LTLRequirements extends Monitor[Event] {
  monitor( new R1, new R2, new R3_R4 )
}
```

6.2.5.4 State machine with TraceContract

TraceContract supports the encoding of a state machine for runtime verification. To prepare a monitor to behave as a state machine each of the nodes from Figure 6.2 are represented as TraceContract monitors. The method `state` is used in conjunction with a partial function that matches the event (next state) and moves to the next node as appropriate. Two of the nodes, s_0 and s_4, are provided as examples:

```
property('R5) { S0 }

def S0: Formula = state {
                    case Event(true ,false,false) => S4
                    case Event(false,false,false) => S0
                    case _ => error
                  }
def S4: Formula = state {
                    case Event(false,false,false) => S0
                    case Event(false,true ,true ) => S3
                    case Event(true ,true ,true ) => S7
                    case Event(true ,false,false) => S4
                    case _ => error
                  }
```

[3]In the example the requirements R3 and R4 are combined within a single monitor R3_R4. See the code in Listing B.4.

The initial state s_0 is indicated by the construction of a property (R5) which contains s_0. This property forms the `StMRequirements` monitor:

```
class StMRequirements extends Monitor[Event] {
  monitor( new R5 )
}
```

6.2.5.5 Execution timings

The first set of experiements provides a relative performance analysis of the inline monitoring systems ITL-Monitor and TraceContract. Both of these systems perform runtime verification by constructing monitors dynamically via a Scala API and executing these alongside the simulation itself. In this respect, TraceContract follows the same architectural paradigm as ITL-Monitor making TraceContract an excellent candidate for comparative analysis.

For each experiment the time to run the simulation is measured using Java's `nanotime()` system call and the recorded time is the average over ten runs. The experiments are repeated for 20, 40, 60, 80, and 100 A-cycles so that the performance of the verification over intervals with different numbers of states can be compared. The same runtime environment (via `sbt`) is used for each experiment. For each A-cycle length, four experiments are performed: no runtime verification; using ITL-Monitor (ITL); using TraceContract (LTL); and using a TraceContract state machine. The results are listed below (in Table 1 and Table 2).

Table 1

Num of A-cycles	Without monitoring	ITL-Monitor ITL	TraceContract LTL	TraceContract State machine	Num of states*	Time * (s)
20	✓				687	0.011
40	✓				1172	0.016
60	✓				1931	0.038
80	✓				2490	0.057
100	✓				3170	0.096
20		✓			614	0.330
40		✓			1222	0.453
60		✓			1775	0.542
80		✓			2555	0.734
100		✓			3119	0.753
20			✓		616	0.103
40			✓		1249	0.179
60			✓		1765	0.242
80			✓		2631	0.333
100			✓		3162	0.380
20				✓	611	2.447
40				✓	1240	18.336
60				✓	1925	67.325
80				✓	2483	143.435
100				✓	3059	281.840

* Average for ten simulation runs.

The results without monitoring provide a baseline measurement. Runtime verification with ITL-Monitor increases the time taken by one order of magnitude up to about 1.5K states, and then remains at the same order of magnitude up to circa 32K states (Table 2). However, after circa 23K states the experiments run more quickly with ITL-Monitor performing runtime analysis. This counterintuitive result is explained when the CPU loading and number of active threads is inspected. Without monitoring the CPU load for the JVM is maintained around 100% and a single thread is running at any one time. However, when the experiment uses ITL-Monitor the CPU usage for JVM increases to around 300%[4] and the number of active threads increases peaking at eight. ITL-Monitor is written in Akka and its performance will be governed by Akka's threadpool management. Analyis of how to fine tune the threadpool performance for given architectures is left for future work.

The table below summarises a series of experiments in which the ITL-Monitor and TraceContract(LTL) monitors were executed with significantly longer execution traces.

[4]Apple OSX reports up to 100% for each virtual core.

Table 2

Num of A-cycles	Without monitoring	ITL-Monitor ITL	TraceContract LTL	Num of states*	Time * (s)
250	✓			7825	0.588
500	✓			15538	2.191
750	✓			23339	4.916
1000	✓			31513	9.042
2000	✓			62706	35.640
250		✓		7836	1.759
500		✓		15676	3.168
750		✓		23525	4.661
1000		✓		31823	6.318
2000		✓		63341	12.008
250			✓	7874	1.177
500			✓	15946	3.393
750			✓	23843	6.659
1000			✓	31441	11.069
2000			✓	62801	40.978

* Average for ten simulation runs.

Each of the TraceContract monitors `R1`, `R2`, and `R3_R4` are of the form `globally(..)`. TraceContract forks a new monitor for each matching event within a `globally` clause, and each monitor processes events until its formula has been satisfied. Monitors `R1` and `R2` have complementary antecedants and therefore one or the other of these must be triggered in every state. Furthermore, in every state when A is low `R3_R4` is triggered. Each of these monitors' lifetime is only two states but there is nonetheless an extra overhead in creating and disposing of monitors with every new event.

TraceContract performance exhibits exponential growth over the simulation lengths from 0.5K states to 3K states with LTL monitoring being better than the state machine performance. Monitoring of the CPU load during the TraceContract experiments shows that multiple threads are being utilised across the cores but to a significantly less extent with the state machine than the LTL specification. This accords with the sequential flow through the deterministic state machine rather than the forking of multiple monitors within LTL.

The ITL-Monitor specification analyses the data in subintervals according to cycles of A or B. Each fusion point between cycles is the cause of the disposal of one monitor and the creation of a new monitor. Thus the number of monitors created is proportional to the number of A and B cycles and not to the number of states. This is a significant reduction in performance overhead as demonstrated by the timing data.

Considering the AnaTempura verification it is significant to note that the monitoring is performed outline. The simulation does not synchronise with any AnaTempura process and will therefore run independently. However, it is relevant to consider how quickly AnaTempura can process the stream of incoming state data.

Any timing analysis of the AnaTempura monitoring does not affect the time taken to run the simulation because the runtime verification is performed outline. Thus the reported time by AnaTempura is different from that reported by `sbt`. The following table illustrates the difference by showing some sample runs with reported timings (rounded to whole seconds). Each row in the following table is based upon average values from five similar experiments.

Num of A-cycles	Num of states	Elapsed time (s) AnaTempura	Elapsed time (s) sbt
20	612	16	3
40	1228	21	3
60	1863	21	5
80	2557	23	6
100	3088	25	8
250	7828	60	26
500	15938	116	54
1000	31789	224	109

The timings are similar to those for TRACECONTRACT (LTL). It is noticable that the simulation completes significantly before the analysis in each case.

ITL-Monitor revisited

The initial ITL-Monitor specification was based upon subintervals that aligned with cycles of A and B. The resulting ITL modelled the operation of the latches very closely. The four requirements, R1 - R4, were derived from the original ITL specification and it was demonstrated that these were being verified implicitly. However, it is also possible to verify these requirements directly within ITL-Monitor:

```
val R1 = SKIP WITH ((Next(B)'='B) implies not(Next(S)))
val R2 = SKIP WITH ((Next(B)'/='B) implies Next(S))
val R3 = SKIP WITH ((not(A) and not(Next(A))) implies (B'='Next(B)))
val R4 = SKIP WITH ((not(A) and Next(A)) implies (B'='Next(B)))

val spec2  = (GUARD(initial) THEN (HALT(STOP) ITERATE (R1 AND R2 AND R3 AND R4)))
```

Here, formulae with the form keep f have been expressed equivalently as (skip $\wedge f)^*$. There is an associated cost with this approach: all of the subintervals used for generating ITL-Monitors are of unit length. This means that a monitor will need to be created for every state. In this way the verification approach is now closer to TRACECONTRACT (LTL) in its operation. The table below shows the timings for running ITL-Monitor using this revised formula. The timings are higher than the equivalent ones for the original ITL-Monitor specification and approach the TRACECONTRACT (LTL) timings at the higher numbers of states. This observation is consistent with the fact that the revised `spec2` spawns monitors at every state as does TraceContract (LTL).

Num of A-cycles	ITL-Monitor ITL	Num of states*	Time * (s)
20	✓	691	0.697
40	✓	1310	0.921
60	✓	1842	1.135
80	✓	2599	1.555
100	✓	3226	1.811
250	✓	7906	4.169
500	✓	15440	7.792
750	✓	23803	12.348
1000	✓	31511	15.418
2000	✓	63068	31.445

* Average for ten simulation runs.

6.2.5.6 Reporting and recovery

In this section the runtime behaviour of each of the monitoring systems, TraceContract, ITL-Monitor, and AnaTempura, will be analysed. The discussion takes place within the context of the latch example and will address how each system provides a running commentary, reports a successful verification, reports a failure, and supports error recovery.

Displaying progress Each of the monitoring systems can display progress as it receives states from the simulation. TraceContract prints out a report each time a monitor is satisfied. Thus, monitor 'R1 is satisfied by the first two states:

```
Monitor: TC$LTLRequirements.TC$R1
Property 'R1 succeeds

Succeeding event number 2: Event(false,false,false)
Trace:
  1=Event(false,false,false)
  2=Event(false,false,false)
```

AnaTempura similarly reports on each individual state:

```
State   0: A=false, B=false, S=false, STOP=false
State   0: -- Pass R1
State   0: -- Pass R2
State   0: -- Pass R3
State   0: -- Pass R4
State   1: A=false, B=false, S=false, STOP=false
State   1: -- Pass R1
State   1: -- Pass R2
```

```
State   1: -- Pass R3
State   1: -- Pass R4
```

For ITL-Monitor each generated state is printed with a judgement. The first two states of a simulation are shown below – in each case the judgement is MORE meaning that no violation has been detected at this stage.

```
( 0.061 sec): RTM (Latch1) More     0 A -> false  B -> false  S -> false  STOP -> false
( 0.062 sec): RTM (Latch1) More     1 A -> false  B -> false  S -> false  STOP -> false
```

Reporting success All of the systems provide a report when the simulation has completed – i.e., when the finite execution trace has ended. TRACECONTRACT lists each monitor and reports on the number of violations detected. The simulation ran to completion and no violations were detected.

```
Monitor TC$LTLRequirements.TC$R1 property 'R1 violations: 0
Monitor TC$LTLRequirements.TC$R2 property 'R2 violations: 0
Monitor TC$LTLRequirements.TC$R3_R4 property 'R3_R4 violations: 0
```

AnaTempura simply reports Done! and provides some statistical information about the computation. For example:

```
Done! Computation length: 40. Total Passes: 43.
Total reductions: 8061 (7984 successful). Maximum reduction depth: 15.
Time elapsed: 10.696765
```

Finally, ITL-Monitor reports Done and lists the elements of the final state. The [INFO] message indicates that the Akka actor system has shut down the monitor.

```
Done(List((B,false), (STOP,true), (S,true), (A,false)))
[INFO] [03/13/2019 22:17:04.641] [run-main-e]
       [Monitor$Runtime$RTM(akka://LatchActorSystem)]
       Stop: Monitor Latch1 has been stopped.
```

Reporting failure TRACECONTRACT continuously reports on the status of each of its monitors until the end of the simulation. The simulation allows for a random failure to be introduced. In this case an error was introduced at the fourth state:

```
Monitor: TC$LTLRequirements.TC$R2
Property 'R2 violated

Violating event number 4: Event(true,true,false)
Trace:
  3=Event(true,false,false)
  4=Event(true,true,false)
```

AnaTempura reports violations associated with monitors and states. In this case the nature of the reporting is controlled within the `latch.t` program itself by side effecting the Pass/Fail messaging within the monitor formulae (Listing 6.1 lines 22-33).

```
State  10: A=false, B=true, S=false, STOP=false
State  10: -- Pass R1
State  10: ** Fail R2
State  10: -- Pass R3
State  10: -- Pass R4
```

In ITL-Monitor a failure causes the verification to terminate with a message on the transcript. An example of the type of message that appears on the transcipt is shown below. Extraneous Akka messages have been removed.

Listing 6.3: ITM failure detection

```
( 0.025 sec): RTM (Latch1) More     3 A -> true  B -> false  S -> true  STOP -> false
( 0.027 sec): RTM (Latch1) Failed   4 A -> true  B -> true  S -> false  STOP -> false
RTVException Failure Latch1
React at Runtime...
================================
Terminating monitor Monitor:
================================

[WARN] (anon)THEN: RHS failed
[WARN] (anon)WITH: RHS failed
[WARN] (anon)ITERATE: LHS failed
[WARN] (anon)ITERATE: RHS failed
```

The message describes a path through a monitor formula to assist in locating the source of the failure. In this case the failure appears to have occurred in the subformula to the right of `WITH` in `clause2`.

```
val initial = (~A and ~B and ~S)
val clause2 = FIRST(B <~ ~B) WITH (skip ';' halt(S))
val clause3 = (HALT(A) WITH (stable(B))) THEN (HALT(~A))
val spec    = (GUARD(initial)
               THEN (HALT(STOP)
                     ITERATE clause2 ITERATE clause3))
```

The `WARN` messages are output using Akka's asynchronous logging mechanism and, as such, their order is non-deterministic. In the example there is no ambiguity about the location of the failure: the only path that satisfies `THEN RHS`, `ITERATE LHS`, `ITERATE RHS`, `WITH RHS` leads to the subexpression skip ; halt(S). However, it is possible to label any of the ITL-Monitor subformulae to assist in locating the source of a failure or to resolve any potential ambiguity. For example, consider a monitor formula of the form `(a ITERATE b) ITERATE (c ITERATE d)`; in such a case a failure report consisting of {`ITERATE LHS`, `ITERATE RHS`} is ambiguous.

The operator `::` is provided by the ITL-Monitor API so that individual subformulae can be labelled. If the following changes were made to the previous example:

```
val clause2 = "clause2"::(FIRST(B <~ ~B) WITH (skip ‘;‘ halt(S)))
val spec    = "spec"::(GUARD(initial)
                 THEN "loop"::(HALT(STOP)
                          ITERATE clause2 ITERATE clause3))
```

then the error reporting would include the subformula labels (anon is the default for unlabelled formulae).

```
[WARN] (spec)THEN: RHS failed
[WARN] (clause2)WITH: RHS failed
[WARN] (loop)ITERATE: LHS failed
[WARN] (anon)ITERATE: RHS failed
```

Error recovery

Both TraceContract and AnaTempura only provide a report on the output transcript. There is no message passing back to the simulation. The communication of states from the simulation to AnaTempura is via `printf` messages on `stdout`. In TraceContract each state is evaluated using the `verify()` method but, because this method returns `()`[5], it cannot report a failure back to the simulation. In contrast, ITL-Monitor verification returns judgements (of type `Reply`) to the program under test. The methods provided are:

```
def verify: Reply            // returns a judgement (PASS, FAIL, or MORE)
def !                        // a synonym for verify
def !!(e: Exception): Reply // as above but throws e on failure
def !! : Reply               // as above but throws default exception
```

These methods communicate synchronously with the monitor. This design ensures that the calling program can react at runtime as soon as a failure is reported.

```
if (mu.verify.isFail)
  ...
else
  ...
```

However, when there are many such assertion points within a block of code, it may be preferable to associate all of them with the same recovery action. The `!!` methods are designed to be used with the `try/catch` pattern. This is illustrated by the current example – see Listing B.5. The output displayed in Listing 6.3, line 4, demonstrates that recovery code within the `catch` block has been executed following a failure detection. In this example the recovery code is simply a placeholder message, but the principle has been established.

[5] `()` is the only element of the `Unit` type in Scala. It is used similarly to `void` in Java and C to indicate that the method is called only for its side effect.

6.3 Checkout system

In this section a larger example is described which simulates a self-service checkout of the type currently popular in many large retail outlets. It is designed to model a single day's trading. The checkout is a reactive system designed to operate for a finite length of time. It models a realistic system and is capable of generating a large volume of data spanning the full range of possible interactions that can take place during each customer transaction. It will enable ITL-Monitor to be analysed with large volumes of data.

The simulation is comprised of the following components.

Customers The simulation generates customers with randomised shopping baskets

Attendant The attendant is responsible for a given number of terminals. Each customer is assigned to a free terminal by the attendant and the attendant reacts to situations during a transaction such as "assistance required" or "unexpected item in bagging area" or "check customer age" whenever an age-restricted product is scanned.

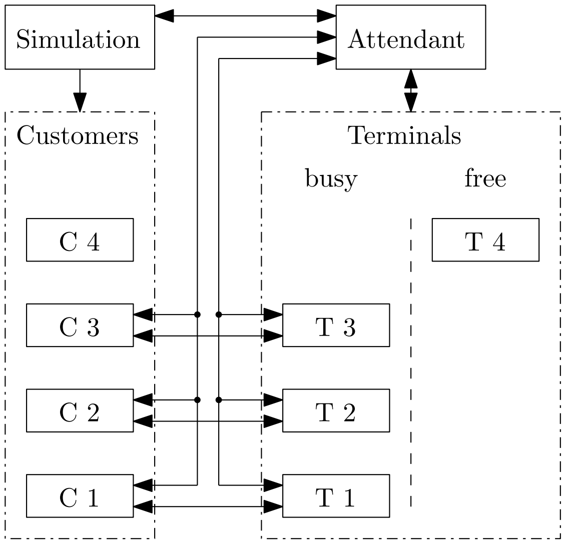
Terminals The simulation models several terminals all of which are managed by the attendant. Each terminal interacts with one customer at a time. The interactions include scanning products and placing them in the bagging area and finally ensuring payment for the goods. When items are placed on the scale in the bagging area their respective weights are checked against the product DB to see if they are correct within a given tolerance. When the customer requests assistance or an intervention is required the terminal alerts the attendant and awaits instruction.

Product DB The product DB maintains details of the products including their price and weight. Within the context of the simulation it is also responsible for generating random shopping baskets of products for each newly generated customer.

The simulation The main simulation is responsible for initialising, running, and finalising the components. It is fundamentally a loop which repeatedly generates a new customer and "offers" the customer to the attendant to be shown to a free terminal. The simulation polls the attendant at given time intervals and generates a new customer once the current one has been "accepted". This ensures a constant supply of new customers until the store closes.

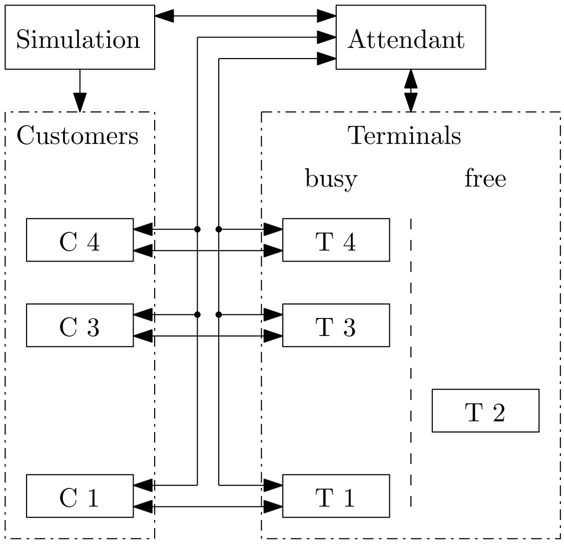
A diagram showing the interaction of the principal actors in the simulation is shown in Figure 6.3.

If customer C2 were to complete their transaction next then the corresponding terminal T2 would return to the pool of free terminals and the associated customer actor would be removed



In this example the attendant looks after four terminals but this number is configurable upon creation of the attendant. The simulation creates customers and passes them (in turn) to the attendant. The attendant allocates a free terminal to a customer. In the state depicted the simulation has just passed the reference to newly created C4 to the attendant who is about to allocate the new customer to the free terminal T4. Whenever there are no free terminals then the customer must wait. The attendant then maintains contact with all the customers and all the terminals reacting to situations that arise. The diagram does not show the product database.

Figure 6.3: The principal actors in the simulation



Customer C2 has completed their transaction and the now-obsolete actor has been garbage-collected; terminal T2 is returned to the free pool ready to be assigned by the attendant to the next customer.

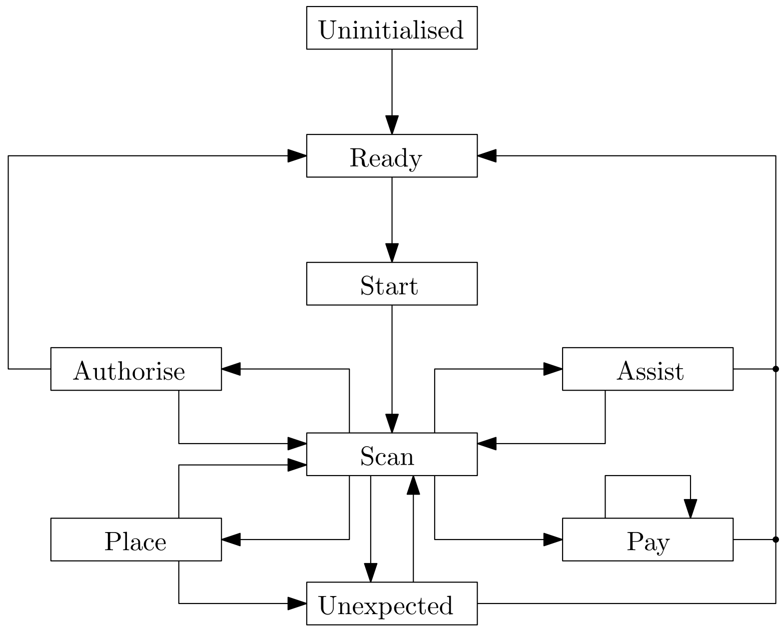
Figure 6.4: The state after customer C2 has completed their transaction

from the simulation. The actor would be garbage collected. The situation in which customer C4 has been allocated to the previously free terminal T4, and in which C2 has completed their transaction, is depicted in Figure 6.4.

Communication between the actors in the simulation takes the form of message passing with immutable data. Each actor represents its own state-transition system that governs its behaviour. Figure 6.5 shows the behaviour of a terminal. The system described makes a number of assumptions and simplifications in order to manage its complexity in this context. For example, it is assumed that the customer will not seek assistance when paying for the goods; and the range of assistance that may be sought and provided is restricted to a small number of representative transactions. Notwithstanding these simplifications, the system generates a sufficiently varied range of traces by which the behaviour of ITL-Monitor can be demonstrated and analysed. Figure 6.6 shows a part of a runtime trace filtered to show only the messages received by, and sent from a single terminal.

6.3.1 Modelling the terminal class

In this section the emphasis is solely upon the implementation of the Terminal class so that it can later be analysed with runtime verification. The principles apply to any of the components in the simulation and it is equally possible to attach monitors to the attendant, or to each



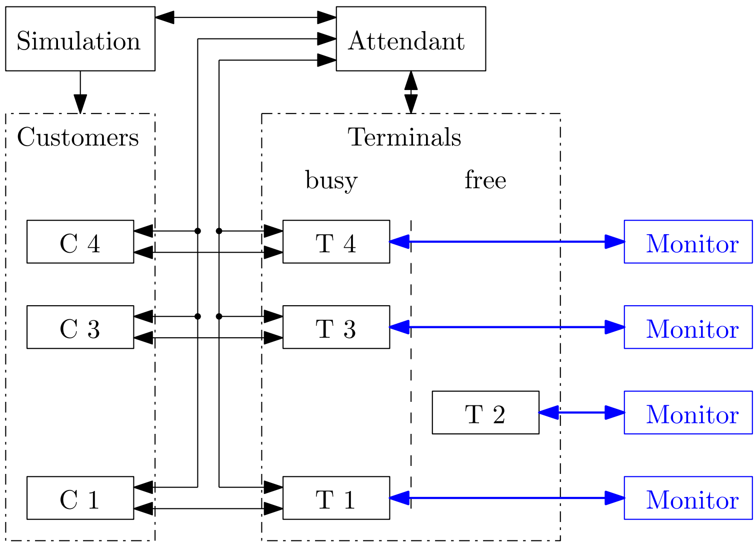
State	**Waiting for**	**Action**	**Next state**
Uninitialised	Internal state set-up	Initialise (handshake)	Ready
Ready	New customer arrival	New customer assigned	Start
Start	Customer to start	Ask customer to scan	Scan
Scan	Customer activity	Scans non-restricted item	Place
		Scans restricted item	Authorise
		Seeks assistance	Assist
		Places item	Unexpected
		Finish and pay	Pay
Authorise	Authorisation	Granted	Place
		Refused / within permitted attempts	Scan
		Refused / too many attempts → end transaction	Ready
Place	Item placed	Weight within tolerance → accept	Scan
		Weight not within tolerance → reject	Unexpected
Pay	Customer to insert card	Payment accepted / transaction complete	Ready
		Payment rejected / ok to try again	Pay
		Payment rejected / too many attempts → end transaction	Ready
Assist	Attendant to help	Help provided → carry on	Scan
		Unwanted item → item removed	Scan
		Abandon transaction	Ready
Unexpected	Attendant to adjudicate	Ok → remove and carry on	Scan
		Not ok → end transaction	Ready

Figure 6.5: State transition diagram for a Terminal

```
[T 1/C247] [scan      ]  ----> T2C_ScanNextItem
[T 1/C247] [scan      ]  <---- C2T_Scan: (921)
[T 1/C247] [scan      ]  - - - Reported: TotalPrice: $\$$ 13.67 NbrOfItems: 5
[T 1/C247] [scan      ]  ----> T2A_AuthorisationRequired: 1014987439
[T 1/C247] [authorise ]  <---- A2T_AuthorisationGranted
[T 1/C247] [authorise ]  ----> T2C_PlaceItemOnScale
[T 1/C247] [place     ]  <---- C2T_Put: (494)
[T 1/C247] [scan      ]  ----> T2C_ClearScanner
[T 1/C247] [scan      ]  ----> T2C_ScanNextItem
[T 1/C247] [scan      ]  <---- C2T_FinishAndPay
[T 1/C247] [scan      ]  ----> T2C_PayWithCard
[T 1/C247] [pay       ]  <---- C2T_InsertCard
[T 1/C247] [pay       ]  ----> T2C_PaymentAccepted
[T 1/C247] [pay       ]  ----> T2A_TransactionComplete
[T 1/C247] [ready     ]  <---- A2T_Assign: -664662046: (248)
[T 1/C248] [ready     ]  - - - Reported: Sending to customer 72140258
[T 1/C248] [ready     ]  ----> T2C_Welcome
[T 1/C248] [start     ]  <---- C2T_Start
[T 1/C248] [start     ]  ----> T2C_ScanNextItem
[T 1/C248] [scan      ]  <---- C2T_Scan: (901)
[T 1/C248] [scan      ]  - - - Reported: TotalPrice: $\$$ 0.00 NbrOfItems: 0
[T 1/C248] [scan      ]  ----> T2A_AuthorisationRequired: -664662046
[T 1/C248] [authorise ]  <---- A2T_AuthorisationRefused
[T 1/C248] [authorise ]  ----> T2C_ClearScanner
[T 1/C248] [authorise ]  ----> T2C_ScanNextItem
[T 1/C248] [scan      ]  <---- C2T_Scan: (109)
[T 1/C248] [scan      ]  - - - Reported: TotalPrice: $\$$ 0.00 NbrOfItems: 0
[T 1/C248] [scan      ]  ----> T2C_PlaceItemOnScale
[T 1/C248] [place     ]  <---- C2T_Put: (645)
[T 1/C248] [scan      ]  ----> T2C_ClearScanner
[T 1/C248] [scan      ]  ----> T2C_ScanNextItem
[T 1/C248] [scan      ]  <---- C2T_Put: (640)
[T 1/C248] [scan      ]  ----> T2A_UnexpectedItemInBaggingArea
[T 1/C248] [unexpected]  <---- A2T_ResetForNextCustomer
[T 1/C248] [unexpected]  ----> T2C_TransactionTerminated
[T 1/C248] [unexpected]  ----> T2A_TransactionComplete
[T 1/C248] [ready     ]  <---- A2T_Assign: -1076536131: (249)
```

This short extract from an execution trace shows a sequence of messages received by, and sent from, terminal T1 during the end of a transaction with customer C247 and at the start of a transaction with new customer C248.

Figure 6.6: Extract from the messages received by, sent from, a terminal



Each terminal has its own runtime monitor running concurrently with it. If there are four terminals then there are four concurrent monitors. It is possible to monitor the state of any class/actor and another could be added to, for example, the attendant. Since these are Akka actors the Akka dispatcher will distribute the processes across the available cores although the total monitoring load necessarily becomes part of the overall computation.

Figure 6.7: Each terminal is associated with its own runtime monitor

customer, or to the simulation driver itself. The terminals have been selected because they represent static machines that would most likely be candidates for such analysis and also because they appear as multiple instances. This permits simultaneous monitoring across multiple, available cores. Figure 6.7 illustrates a situation in which the simulation has four terminals, each with its own independent runtime monitor.

A terminal is an Akka actor that maintains local state and updates this state in reaction to messages received from other actors in the system. When a state change occurs the partial function can be substituted for another in situ: this context switching technique is achieved using the Akka become method – "become the following state". Not all of the private state of a terminal will be monitored and it is useful to partition the state space into monitored and non-monitored variables.

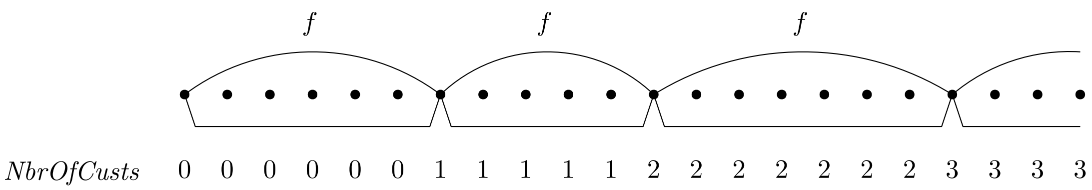
The following code extract shows the definition of the Terminal class and the subsequent state variables partitioned accordingly. The unmonitored variables are modelled using Scala variable types and the monitored variables use the parameterised `Var[T]` type imported from the ITL API.

```
class Terminal( attendant: ActorRef,        // attendant to which this terminal belongs
```

```
                tid: Int,                  // terminal ID
                productDB: ActorRef        // product database
              ) extends Actor with ActorLogging {

/* ****************************************************************************
* Unmonitored state variables
*****************************************************************************/

private val scanned = ListBuffer[(Int, Int)]()    // items scanned (bagging area)
private var offered: Product = Product.nothing    // item just scanned

/* ****************************************************************************
* Monitored state variables
*****************************************************************************/

object CID           extends Var[Int]      {override def toString = "CID"         }
object TotalPrice    extends Var[Int]      {override def toString = "TotalPrice"  }
object NbrOfItems    extends Var[Int]      {override def toString = "NbrOfItems"  }
object NbrOfCusts    extends Var[Int]      {override def toString = "NbrOfCusts"  }
object NbrOfPayments extends Var[Int]      {override def toString = "NbrOfPayments"}
object NbrOfRefusals extends Var[Int]      {override def toString = "NbrOfRefusals"}
object IsClosed      extends Var[Boolean]  {override def toString = "IsClosed"    }
object HelpLight     extends Var[Boolean]  {override def toString = "HelpLight"   }
object Incoming      extends Var[Msg.Value]{override def toString = "Incoming"    }
object Outgoing      extends Var[Msg.Value]{override def toString = "Outgoing"    }
```

6.3.2 Specifications

A list of required temporal properties that must be satisfied by a terminal is provided below. Its execution trace can be divided into a sequence of *transactions* as illustrated in Figure 6.8.



A terminal processes customers sequentially. The variable *NbrOfCusts* is incremented at the start of each new customer transaction. Certain temporal formulae, indicated by f in the figure, apply to subintervals that correspond to individual transactions.

Figure 6.8: Terminal 'lifetime' as a fusion of individual customer transactions

The temporal requirements R1-R5 apply *within* a single transaction – i.e., per customer. These properties apply iteratively over the whole execution trace: in ITL this is written f^*. Given that the overall execution trace is finite, a condition representing the final state is necessary. This reflects the fact that the terminal will eventually be shut down. The variable

IsClosed serves this purpose, and its rôle within an iterative formula representing the whole interval, is given by the following template:

$$\mathsf{halt}(\mathit{IsClosed}) \wedge \\ \quad (\triangleright(\mathit{NbrOfCusts} \lessapprox \mathit{NbrOfCusts} + 1 \vee \mathsf{fin}(\mathit{IsClosed})) \wedge (\mathsf{fin}(\mathit{IsClosed}) \vee f))^*$$

Figure 6.9: Template formula for a customer transaction with requirement f

This is an example of the *exceptional termination* pattern (Section 3.4.3). A Scala function for embedding *single-transaction* formulae within this ITL-Monitor pattern is given below:

Listing 6.4: Single transaction formula template

```
def ByCust(f: Formula): Monitor =
  HALT(IsClosed) ITERATE (
    FIRST(NbrOfCusts <~ NbrOfCusts+1 or fin(IsClosed)) WITH (
      fin(IsClosed) or f
    )
  )
```

This structure emphasises the interval-based properties of these requirements. Each of the *per-transaction* requirements (R1 - R5 below) is embedded within the above template to form the ITL-Monitors: i.e., `ByCust(R1)`, `ByCust(R2)`, etc. As discussed in Section 2.2, aside from individual propositions, LTL cannot restrict the scope of temporal formulae to apply over subintervals.

Requirements

The requirements are presented in ITL alongside their translation into ITL-Monitor. Each of R1-R5 will be substituted into the single transaction formula (see Listing 6.4 and Figure 6.9) to construct the required monitor for the whole execution trace. Requirements R4 and R5 also have related TRACECONTRACT properties defined alongside for comparison.

R1 Whenever the number of (authorisation) refusals exceeds the maximum allowed, then the transaction must be terminated.[6] This property must hold for every transaction.

$$\Diamond(\mathit{NbrOfRefusals} > \mathit{NUM\_OF\_REFUSALS\_ALLOWED}) \supset \\ \quad \Diamond(\mathit{Outgoing} = \mathit{T2C\_TransactionTerminated})$$

```
val R1 = eventually (NbrOfRefusals > NUM_OF_REFUSALS_ALLOWED) implies
         eventually (Outgoing `=` Msg.T2C_TransactionTerminated)
```

[6] This is an example of an obligation property [MP95].

R2 Whenever the help light is illuminated it will eventually be switched off. The help light state is used here as a proxy for help being provided, which is denoted by the light being switched off.[7] This property must hold for every transaction.

$$\boxed{m}\ (HelpLight \supset \Diamond(\neg\ HelpLight))$$

The operator $\boxed{m}$ specifies behaviour in all suffix intervals except the final state. This weaker form of $\Box$ is used because in the final state the implication is a contradiction. This property must hold for every transaction.

```
val R2 = bm (HelpLight implies eventually(~HelpLight))
```

R3 If the number of (failed) payment attempts reaches its limit, then there must have been precisely that number of payment rejections previously within the current transaction. This property must hold for every transaction.

$$\Diamond(NbrOfPayments = PAYMENT\_ATTEMPTS\_ALLOWED) \supset$$
$$\Diamond\!\!\!\!\diamond\ (\bigcirc(\mathsf{halt}(Outgoing = T2C\_PaymentRejected)))^{PAYMENT\_ATTEMPTS\_ALLOWED}$$

```
val R3 = eventually(NbrOfPayments '='PAYMENT_ATTEMPTS_ALLOWED) implies
         di(next(halt(Outgoing '='Msg.T2C_PaymentRejected)) times
                                            PAYMENT_ATTEMPTS_ALLOWED)
```

The formula $\Diamond\!\!\!\!\diamond\ (\bigcirc(\mathsf{halt}(w)))^k$ states that there must be some prefix interval that satisfies k iterations of $\bigcirc(\mathsf{halt}(w))$. For example, if $k = 3$, the formula would be satisfied by an interval with the following pattern: $[\bullet\bullet\ w \bullet\bullet\bullet\ w \bullet\ w]$.

R4 Whenever there is an unexpected item in the bagging area then the next outgoing message must report either a terminated transaction or a removed item. This property must hold for every transaction.

$$\begin{aligned}
&\Box(\ Outgoing = T2A_UnexpectedItemInBaggingArea \supset \\
&\quad \Diamond\!\!\!\!\diamond\ (\ \triangleright(\neg\ (\mathsf{stable}(Outgoing))) \wedge \\
&\qquad \mathsf{fin}(Outgoing = T2C_TransactionTerminated \vee \\
&\qquad\quad Outgoing = T2C_RemoveSelectedItem)\ )\ )
\end{aligned}$$

```
val R4 = always(
           (Outgoing '=' Msg.T2A_UnexpectedItemInBaggingArea) implies
              di( fst(not(stable(Outgoing))) and
                  fin( (Outgoing '=' Msg.T2C_TransactionTerminated) or
```

[7] This is an example of an response property [MP95].

```
                         (Outgoing `=` Msg.T2C_RemoveSelectedItem) ) ) )
```

This reads as follows. Whenever an unexpected item occurs then, from that point, there must be some prefix interval whose final state contains the first update to *Outgoing* which must be either a transaction terminated or a remove selected item message.

It is useful to consider the use of TraceContract for monitoring this behaviour. While it is natural for the ITL-based approaches (ITL-Monitor and Tempura) to treat execution traces as sequences of states, TraceContract considers a trace to contain events. In the earlier latch case study (Section 6.2) states were treated as events. In this example, the ingoing and outgoing messages alone can be sent to TraceContract rather than complete states. The type of data that would be generated is illustrated below:

1 A2T_Assign	:
2 T2C_Welcome	97 T2C_ClearScanner
3 C2T_Start	98 T2C_ScanNextItem
4 T2C_ScanNextItem	99 C2T_FinishAndPay
5 C2T_Scan	100 T2C_PayWithCard
6 T2C_PlaceItemOnScale	101 C2T_InsertCard
7 C2T_Put	102 T2C_PaymentAccepted
:	103 T2A_TransactionComplete

However, such data is generated per-customer and it is possible to adapt the simulation to start a new TraceContract monitor *for each* customer transaction. Requirement R4 can be rewritten in terms of traces of input/output events. If an unexpected item appears in the bagging area then, there must not be another unexpected item in the bagging area until the transaction is terminated or the original unexpected item has been removed.

$$\begin{aligned}&\Box(\mathit{T2A\_UnexpectedItemInBaggingArea} \Rightarrow\\&\quad \bigcirc((\neg\ \mathit{T2A\_UnexpectedItemInBaggingArea})\ \mathcal{U}\\&\qquad (\mathit{T2C\_TransactionTerminated} \vee \mathit{T2C\_RemoveSelectedItem}))\end{aligned}$$

The translation into TraceContract takes advantage of Scala's implicit definitions. In this case messages are lifted to TraceContract LTL formulae automatically – this simplifies the presentation of property 'R4 and aligns it with the LTL formula above.

```
class TCR4 extends tracecontract.Monitor[Msg.Value] {
  import tracecontract._

  implicit def msgToFormula(msg: Msg.Value): Formula = matches {
      case m if m == msg => true
```

```
      case _              => false
    }

  property('TCR4) {
    globally {
      Msg.T2A_UnexpectedItemInBaggingArea implies (
        strongnext(
          not(Msg.T2A_UnexpectedItemInBaggingArea) until (
            Msg.T2C_TransactionTerminated or
            Msg.T2C_RemoveSelectedItem
          )
        )
      )
    }
  }
}
```

R5 A payment rejected message should only occur if a pay with card message has been sent previously within the same transaction. This property must hold for every transaction.

$$\boxdot\big(\ \mathsf{fin}(Outgoing = T2C\_PaymentRejected) \supset \\ \Diamond(Outgoing = T2C\_PayWithCard)\ \big)$$

```
val R5 = bi ( fin (Outgoing `=` Msg.T2C_PaymentRejected) implies
              eventually (Outgoing `=` Msg.T2C_PayWithCard) )
```

The operator $\boxdot$ specifies *all initial intervals*. The requirement states that any prefix interval ending with *PaymentRejected* must contain a state in which *PayWithCard* has taken place: i.e. that the latter has occurred *previously* within the interval.

Transaction subintervals are specified with `ByCust(R5)` (cf. 6.4): thus the subformula $\Diamond(Outgoing = T2C\_PayWithCard)$ is restricted to the 'current transaction'.

It is possible to compare this specification with TRACECONTRACT using its support for LTL with past time events. As in R4 above, the TRACECONTRACT monitor will be defined on a per-transaction basis. TRACECONTRACT supports LTL with past-time events by maintaining a 'facts database' [BH11]. These facts persist so it is up to the monitor to add and remove facts at the right times. Unlike ITL where the scope of past values is delimited by the extent of a subinterval, the database approach relies upon inserting code into the monitor formula to add and remove facts at the correct times.

Such a facts database has been utilised in the following TRACECONTRACT specification of requirement R5. The postfix operators `+`, `?`, and `~`, cause their associated facts to be added, and queried for presence, and absence respectively.

```
class TCR5 extends tracecontract.Monitor[Msg.Value] {
  import tracecontract._

  case object CardPaymentRequested extends Fact
  property('TCR5) {
    require {
      case Msg.T2C_PayWithCard => CardPaymentRequested +
      case Msg.T2C_PaymentRejected if CardPaymentRequested ? => ok
      case Msg.T2C_PaymentRejected if CardPaymentRequested ~ => error
    }
  }
}
```

R6 One transaction in each group of ten must complete a successful payment.

$$\textsf{HALT}\ (\mathit{IsClosed})\ \textsf{ITERATE}\ (\\ \quad \textsf{FIRST}\ (\mathit{NbrOfCusts} \lessapprox \mathit{NbrOfCusts} + 1 \vee \mathsf{fin}(\mathit{IsClosed}))\ \textsf{TIMES}\ 10\ \textsf{SOMETIME}\ (\\ \quad\quad (\mathit{Outgoing} = \mathit{T2C\_PaymentAccepted})\ )\ )$$

This specification highlights the use of nested, iterative monitor composition. The monitor iterates as long as the simultation is running (**HALT** (*IsClosed*)). Each iteration is a sequence of ten transactions. Within that ten-transaction interval, at least one payment accepted message must hold. The use of **SOMETIME** means that the proposition is monitored continually.

```
def R6 = HALT(IsClosed) ITERATE (
            FIRST(NbrOfCusts <~ NbrOfCusts+1 or
                                fin(IsClosed)) TIMES 10 SOMETIME
              (Outgoing `=` Msg.T2C_PaymentAccepted)
          )
```

Each of the requirements R1-R5 are transaction-based and thus can be embedded within the `ByCust(f)` template. R6 spans transactions and thus stands alone. The combined monitor expression is presented below. Combining these monitors with **AND** requires that they all satisfy the same execution trace.

Each submonitor is labelled to enable identification of component formulae in logfile data.

```
val specification = "R1"::ByCust(R1) AND
                    "R2"::ByCust(R2) AND
                    "R3"::ByCust(R3) AND
                    "R4"::ByCust(R4) AND
                    "R5"::ByCust(R5) AND
                    "R6"::R6
```

PAYMENT_ATTEMPTS_ALLOWED	2	number of payment attempts allowed before the transaction is terminated
NUM_OF_REFUSALS_ALLOWED	1	number of times a restricted item can be refused before underage customer is evicted
PROBLEM_SOLVED_LIKELIHOOD	95	percentage likelihood of solving a general enquiry and continuing with shopping
CUSTOMER_TRUST_LIKELIHOOD	95	percentage likelihood that the attendant trusts the customer following mistake
HELP_BUTTON_PRESSED_LIKELIHOOD	10	percentage likelihood of customer pressing the help button
GENERAL_ASSIST_LIKELIHOOD	60	percentage likelihood of customer requiring general assistance
UNDO_PREV_ITEM_LIKELIHOOD	30	percentage likelihood of customer wishing to undo the last scanned item
PLACE_WRONG_ITEM_LIKELIHOOD	10	percentage likelihood of customer placing the wrong item in the bagging area
UNDERAGE_LIKELIHOOD	10	percentage likelihood of customer being underage
CARD_ACCEPTED_LIKELIHOOD	90	percentage likelihood of customer card being accepted (by machine/bank/etc.)
PUT_NOT_SCAN_LIKELIHOOD	5	percentage likelihood of customer putting an item down before scanning it

Figure 6.10: Simulation constants for the checkout system

6.3.3 Timing data

The simulation is controlled using a set of constants which is presented as Figure 6.10. The following values may also be set for a given run:

- The number of terminals available to serve customers.
- The number of shopping items selected randomly for each customer.
- The number of customers to be processed. In terms of the scenario, this determines the trading period. It therefore controls the length of a simulation run.

The first set of results is used for analysing the time it takes to run sequences of transactions. The requirements R1-R5 are written on a per-transaction basis and, as such, cause the monitor to verify the requirements one transaction at a time. The five requirements are monitored simultaneously. For the purpose of the timing analysis each simulation run completes and all of the requirements succeed. The simulation generates random events in accordance with the simulation parameters (Figure 6.10) and these are recorded via instrumentation points (`verify`) placed within the `Terminal` class.

Figure 6.11 shows the outputs for a number of relatively short runs of 100 transactions (representing 100 customers). The number of items in each customer's shopping cart has been set at 10, 50, 100, and 200 respectively. For each run the total number of states generated by the simulation is recorded along with the total time spent analysing the transactions. These times are calculated using the system timer within the verification method. Each transaction has its own interval length depending upon the combination of events that occur. Some intervals may be significantly shorter than the number of items if, for example, the transaction is terminated early. However, the interval lengths are averaged over 100 transactions. The maximum length of any interval is also captured by the system and reported. The interval lengths and times are averaged over five runs.

These relatively short runs generate interval lengths up to on average c. 350 states. Some of the maximum values are significantly larger. The average time for simultaneously checking requirements R1-R5 is less than a third of a second. The interval lengths are based upon realistic amounts of shopping (even though 200 items is perhaps a little extreme) and thus demonstrate that checking these requirements is very feasible. In the type of system this is modelling, these verification times are well within what would be required.

In the first experiment, the time taken to evaluate each of the requirements was dependent upon the length of the interval and the particular values assigned by the simulation. Figure 6.12 shows the outputs for verification of formulae with guaranteed evaluation worst-case complexities of $\mathcal{O}(n^3)$ and $\mathcal{O}(n^4)$.

The $\mathcal{O}(n^3)$ formula $\square(\square((\diamond(\mathsf{empty}))))$ equates to $\neg\ (\diamond(\diamond(\neg\ (\diamond(\mathsf{empty})))))$ which, in turn, is equal to $\neg\ (\mathsf{true}\ ;\ (\mathsf{true}\ ;\ (\neg\ (\mathsf{true}\ ;\ (\mathsf{empty})))))$. Evaluation will require the examination of every possible fusion point for each $;$. The results show that for $\mathcal{O}(n^3)$ formulae, the evaluation times are within a tenth of a second for intervals up to c. 300 states. However, once the $\mathcal{O}(n^4)$ formula is used there is a noticable exponential rise in evaluation times. For interval lengths up to around 200 states even $\mathcal{O}(n^4)$ formulae can be verified in under a second.

The third experiment demonstrates scalability. The computational load on each requirement is kept low by maintaining average interval lengths of 80 states. This is testing that a constant performance for the individual interval verifications is maintained as the overall number of states rises. This occurs as the number of states rises from 80K to just under 1m.

6.3.4 Running with TraceContract

It was shown during the development of R4 and R5 that these requirements could be adapted for use with TraceContract. Although these monitors do not capture the whole execution trace, the behaviour of individual intervals is checked by creating and destroying the TraceContract monitors within the simulation when each new customer arrives.

R1-R5, 1 monitor						
No. of intervals	No. items per intvl	Total states monitored	Avg intvl length	Max intvl length	Total time (s)	Avg intvl time (s)
100	10	8009	80	131	6.376	0.06376
100	10	7671	77	139	5.189	0.05189
100	10	7938	79	136	5.064	0.05064
100	10	8033	80	121	5.751	0.05751
100	10	8251	83	146	6.378	0.06378
avg			**80**			**0.05752**
100	50	20963	210	479	36.151	0.36151
100	50	23655	237	485	20.313	0.20313
100	50	23135	231	468	28.292	0.28292
100	50	23565	236	474	18.660	0.18660
100	50	21398	214	476	20.352	0.20352
avg			**226**			**0.24754**
100	100	32651	327	900	26.267	0.26267
100	100	26427	264	879	19.728	0.19728
100	100	27148	227	854	19.856	0.19856
100	100	31672	317	892	25.150	0.25150
100	100	28906	289	889	23.289	0.23289
avg			**285**			**0.22858**
100	200	33155	332	1655	24.453	0.24453
100	200	31916	319	1673	28.308	0.28308
100	200	34595	346	1492	29.392	0.29392
100	200	32798	328	1553	27.699	0.27699
100	200	34772	348	1644	31.943	0.31943
avg			**335**			**0.28359**

Runs for shopping baskets with 10, 50, 100, and 200 items. Experimental output data in Section B.3.1.

Figure 6.11: Experiment 1

1 monitor						
No. of intervals	No. items per intvl	Total states monitored	Avg intvl length	Max intvl length	Total time (s)	Avg intvl time (s)
Formula for each transaction is $\Box(\Box((\Diamond(\mathsf{empty}))))$ – $\mathcal{O}(n^3)$						
100	10	7795	78	121	2.298	0.02298
100	40	21165	212	383	6.118	0.06118
100	70	24703	247	628	8.565	0.08565
100	100	29038	290	872	9.795	0.09795
100	1000	37913	379	1758	12.844	0.12844
Formula for each transaction is $\Box(\Box(\Box(\Diamond(\mathsf{empty})))))$ – $\mathcal{O}(n^4)$						
100	10	8503	85	132	7.051	0.70705
100	40	20628	206	379	78.349	0.78349
100	70	24025	240	610	200.102	2.00102
100	100	28461	285	824	367.922	3.67922

Illustrating worst-case performance characteristics. Experimental output data in Section B.3.2.

Figure 6.12: Experiment 2

TraceContract can be set to log event data and to report successful verifications to the standard output. For example, in a sample transaction, monitor TCR4 reports that Monitor: Terminal$TCMonitor.Terminal$TCR4

```
Property 'TCR4 succeeds

Succeeding event number 58: T2C_RemoveSelectedItem
Trace:
  56=T2A_UnexpectedItemInBaggingArea
  58=T2C_RemoveSelectedItem
```

The TraceContract event numbers do not align with the ITL-Monitor trace because the latter is being monitored for the whole simulation whereas the former only for a given transaction. However, it is possible to locate the corresponding states in the ITL-Monitor trace for comparison. The `More` judgement in state 74 reflects the fact that requirement R4 has passed this sequence successully (some of the variable mappings have been removed for brevity):

```
( 0.214 sec): RTM (T_1) More    72 CID -> 1
                                   Incoming -> C2T_Put
                                   Outgoing -> T2A_UnexpectedItemInBaggingArea
```

R1-R5, 1 monitor						
No. of intervals	No. items per intvl	Total states monitored	Avg intvl length	Max intvl length	Total time (s)	Avg intvl time (s)
1000	10	79949	79	162	41.575	0.04158
2000	10	159838	80	157	80.466	0.04023
3000	10	238610	80	154	112.501	0.03750
12000	10	955706	80	176	468.546	0.03905

The experiment demonstrates that the monitor performance scales linearly, and is capable of handling large data sets. Experimental output data in Section B.3.3.

Figure 6.13: Experiment 3

```
( 0.214 sec): RTM (T_1) More    73 CID -> 1
                                   Incoming -> A2T_ClearMostRecentItem
                                   Outgoing -> T2A_UnexpectedItemInBaggingArea

( 0.215 sec): RTM (T_1) More    74 CID -> 1
                                   Incoming -> A2T_ClearMostRecentItem
                                   Outgoing -> T2C_RemoveSelectedItem
```

It is more difficult with TCR5 because TraceContract cannot identify the relevant states – the check is made against the 'facts database'. However, by inspecting the event log for each customer it is possible to locate an example. Below is the tail of a TraceContract log showing two trigger events, 69 and 71. The required earlier event is number 67. When event 67 occurred the fact `T2C_PayWithCard` was added to the database. When events 69 and 71 occurred the fact was checked. Note that it is not necessary to clear the fact database because the TraceContract monitor is terminated at the end of the transaction.

```
66 C2T_FinishAndPay
67 T2C_PayWithCard
68 C2T_InsertCard
69 T2C_PaymentRejected
70 C2T_InsertCard
71 T2C_PaymentRejected
72 C2T_InsertCard
73 T2C_PaymentAccepted
74 T2A_TransactionComplete
```

The relevant states from the ITL-Monitor trace are extracted below.

```
( 1.895 sec): RTM (T_1) More  3427 CID -> 45  ...  Outgoing -> T2C_PayWithCard
( 1.896 sec): RTM (T_1) More  3428 CID -> 45  ...  Outgoing -> T2C_PayWithCard
( 1.896 sec): RTM (T_1) More  3429 CID -> 45  ...  Outgoing -> T2C_PayWithCard
( 1.897 sec): RTM (T_1) More  3430 CID -> 45  ...  Outgoing -> T2C_PaymentRejected
( 1.898 sec): RTM (T_1) More  3431 CID -> 45  ...  Outgoing -> T2C_PaymentRejected
( 1.898 sec): RTM (T_1) More  3432 CID -> 45  ...  Outgoing -> T2C_PaymentRejected
( 1.898 sec): RTM (T_1) More  3433 CID -> 45  ...  Outgoing -> T2C_PaymentRejected
( 1.899 sec): RTM (T_1) More  3434 CID -> 45  ...  Outgoing -> T2C_PaymentRejected
( 1.899 sec): RTM (T_1) More  3435 CID -> 45  ...  Outgoing -> T2C_PaymentRejected
( 1.899 sec): RTM (T_1) More  3436 CID -> 45  ...  Outgoing -> T2C_PaymentRejected
( 1.899 sec): RTM (T_1) More  3437 CID -> 45  ...  Outgoing -> T2C_PaymentAccepted
( 1.900 sec): RTM (T_1) More  3438 CID -> 45  ...  Outgoing -> T2A_TransactionComplete
```

The example highlights the difference between storing individual events and storing states. In the latter approach certain data is duplicated in successive states while other values change. There is no need to store historical facts because the interval of states for the current transaction delimits the scope of the operators $\boxdot$ and $\Diamond$.

$$\boxdot(\quad \mathsf{fin}(Outgoing = T2C\_PaymentRejected) \supset \\ \qquad \Diamond(Outgoing = T2C\_PayWithCard) \quad)$$

6.4 Summary

The examples in this chapter have demonstrated that ITL-Monitor can be used both to specify and to monitor real word applications. The checkout system delivered performance characteristics in which intervals of length averaging c. 300 states were being verified against ITL formulae with verification time complexity of $\mathcal{O}(n^3)$ in around a tenth of a second. The system also showed that the tool is capable of handling large execution traces. The experimental requirements R1-R5 were verified over a trace consisting of nearly a million states. When the length of the subintervals was maintained at 80 states, ITL-Monitor delivered consistent average interval verification times of c. 0.04 seconds as the number of subintervals was increased from 1000 to 12000. This demonstrates that the underlying architecture is capable of handling this level of throughput.

The first example has demonstrated the advantage of dividing the execution trace into a series of subintervals. For simulation lengths of up to about 300 states both TraceContract and ITL-Monitor were processing the data in similar times. However, as the number of states increased beyond that, ITL-Monitor gained a clear advantage over TraceContract with the latter taking three times longer to complete over 62K states.

ITL-Monitor generates a new monitor each time an A-cycle or a B-cycle completes (about one in every 10 states), whereas TraceContract is generating a new monitor in every state. When the ITL specification was changed to reflect the LTL more closely, the resulting ITL-Monitor performed more slowly – ITL-Monitor took 31 seconds to process 63K states whereas before it only took 12 seconds. TraceContract took 41 seconds.

Both of these systems are Scala DSLs, which have not been especially optimised, running inline with the simulation itself. However, TraceContract has not been written to exploit multi-core architectures, whereas ITL-Monitor has been, by virtue of being implemented as a network of Akka actors. This design decision delegates the distribution of labour across multiple cores to the Akka dispatcher. It is also possible, in principle, to distribute actors across computer systems and networks – Akka already supports this. Exploting this potential for monitor distribution is left for future work.

There is a potential drawback with performing verification after each subinterval has been collected. Faults will be detected at this level of granularity. Runtime verification systems whose monitors check every state against the specification will be able to detect faults immediately. It is a trade-off. ITL-Monitor is a candidate runtime verification tool for systems for which the time to evaluate each verification judgement is required to be of the order of tenths of seconds, or greater, rather than, e.g., miliseconds.

Practice with these runtime verification tools has highlighted the utility of monitor composition. TraceContract provides this using monitor hierarchies (trees) in which siblings are run concurrently. The internals of TraceContract ensures that each event is distributed to all child monitors. The grammar of ITL-Monitor enables monitors to be composed sequentially, iteratively, and in parallel, and the theory developed in Chapter 4 provides an ITL translation of every ITL-Monitor expression. An example of this approach was particularly evident in requirement R6 which exploits all of these features (including the nested iteration of intervals).

$$\begin{array}{l} \textbf{HALT}\ (\mathit{IsClosed})\ \textbf{ITERATE}\ (\\ \quad \textbf{FIRST}\ (\mathit{NbrOfCusts} \lessapprox \mathit{NbrOfCusts} + 1 \vee \mathsf{fin}(\mathit{IsClosed}))\ \textbf{TIMES}\ 10\ \textbf{SOMETIME}\ (\\ \qquad (\mathit{Outgoing} = \mathit{T2C\_PaymentAccepted})\ )\ ) \end{array}$$

The use of Scala DSLs for developing runtime verification libraries is a current research topic. TraceContract is one example and ITL-Monitor is another. Scala's support for higher order functions, pattern matching, and partial function syntax, all contribute to the ease with which such libraries can be developed.

Chapter 7

Conclusion and future work

The use of ITL to specify the ITL-Monitor components required the development of a theory of fixed-length intervals in ITL (Section 3.5.2). Three new ITL operators were introduced: *all strict initial intervals*, $\boxdot$; *some strict initial interval*, $\diamond$, and the *first occurrence operator*, $\triangleright$. An investigation of these operators was conducted which led to the development of a body of laws that has been added to ITL [CMS19]. The theory includes the important law $FstFstChopEqvFstChopFst^{(C.260)}$ $(\vdash \; \triangleright(\triangleright f \;;\; g) \equiv \triangleright f \;;\; \triangleright g)$ which states that the sequential composition of first occurrences is itself a first occurrence. This is a significant result which has an important corollary[1], $FstFixFst^{(C.261)}$ $(\vdash \; \triangleright \triangleright f \equiv \triangleright f)$. This states that a first occurrence of f can have no other first occurrences of f before it. These operators provide the basis for the translations of the ITL-Monitor expressions into ITL (cf. Chapter 4).

The development of ITL-Monitor as a restriction of ITL enabled monitor operators to be investigated mathematically. Their algebraic properties have been discovered and documented [CMS19]. These are particularly useful because, not only does this enable ITL-Monitor expressions to be rewritten and simplified, it also facilitates the transformation of their executable counterparts.

The implementation of ITL-Monitor as a Domain Specific Language in Scala means that monitors exist as dynamic objects within the programs they are verifying and that they can be constructed and executed under program control. This provides flexibility to the software engineer to decide how the runtime verification affects the program's future behaviour. This feature is an implementation of the *react at runtime* pattern discussed in [LS09].

Experimental results were carried out using a case study capable of generating large volumes of simulation data. The performance of ITL-Monitor was found to be scalable – constant interval processing times of a 0.04 seconds/interval were observed as the size of the execution trace increased from 80,000 states to just under a million. The average interval length was 80

[1] Set $g \equiv \mathsf{empty}$.

states. A second experiment showed that verifying ITL formulae with evaluation complexity of $\mathcal{O}(n^3)$ over interval sizes of around 300 states ran at approximately 0.1 seconds/interval. These results indicate that ITL-Monitor is a practical runtime verification tool with real-world potential application.

7.1 Comparison with related work

Interest in using Scala as a language for building runtime monitor DSLs (cf. 2.4.1) [AHKY15] has increased in recent years. The syntactic features available in Scala which facilitate the construction of DSLs, coupled with its compilation into Java bytecode, make it an attractive implementation target. The DSL approach to monitor construction is one of the considerations discussed recently in [BHK16] in which the authors propose a move towards tighter integration of specification logics and programming languages. ITL-Monitor has been developed as an internal DSL in Scala.

ITL-Monitor shares with AnaTempura the use of manually instrumented checkpoints. However, unlike AnaTempura, ITL-Monitor provides a closer coupling between the program checkpoint and the monitor because it uses an internal DSL rather than an external monitor. This embedded approach benefits from the use of the compiler to type-check the data passed to and from each monitor. In principle, the instrumentation of ITL-Monitor could be automated using an approach similar to AspectJ [Asp17] and JavaMOP [JMLR12] with the trigger events being associated with updates to variables. However, the current manual approach affords greater control when a new state is added to the trace. In particular it permits multiple updates at a time to occur to monitored variables (see Section 2.1 in [RH16]).

A key aspect of ITL-Monitor is its compositional construction which it derives from the underlying ITL. Compositionality in ITL has provided the motivation for both theoretical [Mos94, Mos98, Mos13, Mos14b], and practical work [Mos96a, JNWC15, Dim00, Dim02, SCZ03a, SCZ03b, Sie05, JCS+06, JCSZ13, STE+14]. An exciting area of current research involves temporal projection [MG17]. It is envisaged that ITL-Monitor will develop to exploit temporal projection to perform simultaneous runtime verification at different time granularities.

7.2 Limitation

The instrumentation of ITL-Monitor currently utilises a manual approach in which the software engineer inserts code to pass a new state to the monitor. Although this provides a fine level of control over when states are passed, it is incumbent upon the software engineer to insert checkpoints at exactly the right places in the code. This approach is similar to both of the other runtime verification tools that have been used in this thesis (TraceContract and AnaTempura).

7.3 Future work

Distributed monitors. It was a deliberate design decision to use Akka, an industrial strength, distributed actor system, as the basis for the monitor implementation. This already enables monitors to exploit multiple cores courtesy of the Akka dispatcher. This is particularly useful given that ITL-Monitor is an inline monitoring architecture (cf. 2.4). However, Akka actors can also be distributed across networks and it is intended to explore how to configure ITL-Monitor to perform distributed runtime verification.

Time reversal. The underlying theory expressed in ITL has focused on time flowing in a forwards direction. The insight is that the monitor is concerned with consuming new states as they arrive in real time just until the evolving interval satisfies the required condition. Thus the *first occurrence* operator was introduced to facilitate the expression of such requirements. In recent work [Mos14b] time reversal has been introduced into ITL and has been shown to provide new insights and, in some cases, to lead to simpler proofs.

An evolving execution trace is extended at only one end. However, a completed interval can be viewed from either end. It is natural to consider the reflections of the newly added ITL operators ($\boxed{\mathrm{t}}$, $\diamond_{\mathrm{t}}$, $\lhd$) and how a symmetrical theory of first occurrence could be developed. It is expected that this will provide a set of transformations that could provide greater insight into future applications of these new operators.

Temporal projection. ITL-Monitors may be composed sequentially and in parallel. These are built, fundamentally, upon the ITL operators (; and $\wedge$). However, it can be useful to view intervals at different levels of temporal abstraction. This can be achieved using temporal projection, f_1 proj f_2, [MG17, Mos86, BT00], The translation of ITL-Monitor into ITL will need to be adapted to take into account this new feature.

At a practical level, some exploratory work has already been undertaken, and an initial implementation of projection has already been added to the `Monitor.scala` API. In this experimental version, the projection points must be specified deterministically. This has been tried out practically with a small example and it is clear that useful, multi-level verifications can take place simultaneously on an evolving trace. It is expected that such an enhanced ITL-Monitor will have application in monitoring systems at different temporal granularities simultaneously.

7.4 Potential impact

The theory of first occurrence extends the body of knowledge relating to ITL, particularly in respect of fixed length intervals and their combination. The laws have been checked mechanically by Isabelle/HOL using a translation of ITL into Isabelle/HOL produced by Antonio Cau.

The ITL-Monitor API has potential use in commercial or industrial applications. As mentioned above, the intention is to investigate next how to exploit Akka's distributed actor support so that the existing model can be used to monitor a distributed system. This is seen as a significant potential direction for ITL-Monitor.

Bibliography

[AHKY15] C. Artho, K. Havelund, R. Kumar, and Y. Yamagata. Domain-specific languages with Scala. In Michael Butler, Sylvain Conchon, and Fatiha Zaidi, editors, *LNCS: 17th International Conference on Formal Engineering Methods*, volume 9407, pages 1–16. Springer-Verlag, November 2015.

[Akk17] Akka. Akka homepage. online: `http://akka.io`, 2017.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[Asp17] AspectJ. AspectJ. online: `https://eclipse.org/aspectj/`, 2017.

[BB08] Joachim Baran and Howard Barringer. Forays into sequential composition and concatenation in Eagle. In Martin Leucker, editor, *Runtime Verification: 8th International Workshop*, pages 69–85. Springer, 2008.

[BBC+02] Jonathan P Bowen, Kirill Bogdanov, John A Clark, Mark Harman, Robert M Hierons, and Paul Krause. Fortest: Formal methods and testing. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 91–101. IEEE, 2002.

[BGHS04a] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Program Monitoring with LTL in Eagle. In *IPDPS*, 2004.

[BGHS04b] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, pages 44–57, 2004.

[BH11] Howard Barringer and Klaus Havelund. TraceContract: A Scala DSL for Trace Analysis. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, pages 57–72, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BH14] Dines Bjørner and Klaus Havelund. 40 years of formal methods. In *Proceedings of the 19th International Symposium on FM 2014: Formal Methods - Volume 8442*, pages 42–61, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[BHK16] Manfred Broy, Klaus Havelund, and Rahul Kumar. Towards a unified view of modeling and programming. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, pages 238–257, 2016.

[BHRG09] Howard Barringer, Klaus Havelund, David Rydeheard, and Alex Groce. Rule Systems for Runtime Verification: A Short Tutorial. In Saddek Bensalem and Doron A. Peled, editors, *Runtime Verification*, volume 5779, pages 1–24, Berlin, Heidelberg, 06 2009. Springer Berlin Heidelberg.

[BL13] Michael Bevilacqua-Linn. *Functional Programming Patterns in Scala and Clojure*. The Pragmatic Programmers, 2013.

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Workshop on Runtime Verification (RV'07)*, pages 126–138, 2007.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.

[BRH07] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for runtime monitoring: From Eagle to RuleR. In Oleg Sokolsky and Serdar Taşıran, editors, *Runtime Verification*, pages 111–125, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BT00] Howard Bowman and Simon J. Thompson. A Complete Axiomatization of Interval Temporal Logic with Projection. Technical Report 6-00, Computing Laboratory, University of Kent, Canterbury, Great Britain, January 2000.

[Cau07] A. Cau. AnaTempura. online: http://www.antonio-cau.co.uk/ITL/itlhomepagesu11.html#x15-140003.1, 2007.

[Cau08] Antonio Cau. Interval Temporal Algebra. online: http://www.antonio-cau.co.uk/ITL/itl-atp/index.html, 2008.

[CM16] Antonio Cau and Ben Moszkowski. The ITL homepage. online: http://antonio-cau.co.uk/ITL/, 2016.

[CMS19] Antonio Cau, Ben Moszkowski, and David Smallwood. An encoding of Interval Temporal Logic in Isabelle/HOL. online: http://antonio-cau.co.uk/ITL/itlhomepagesu13.html#x17-220003.3, March 2019. (Version 1.9).

[CR03] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2):108–127, 2003.

[CR07] Feng Chen and Grigore Rosu. Mop: An Efficient and Generic Runtime Verification Framework. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 569–588. ACM, 2007.

[CZCM96] Antonio Cau, Hussein Zedan, Nick Coleman, and Ben Moszkowski. Using ITL and TEMPURA for large scale specification and simulation. In *Proc. of the 4th Euromicro Workshop on Parallel and Distributed Processing*, pages 493–500. IEEE Computer Society Press, 1996. .

[DFM+15] Philip Daian, Yliès Falcone, Patrick O'Neil Meredith, Traian-Florin Serbanuta, Shinichi Shiraishi, Akihito Iwai, and Grigore Rosu. Rv-android: Efficient parametric android runtime verification, a brief tutorial. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*, pages 342–357. Springer, September 2015.

[DGH+16] Philip Daian, Dwight Guth, Chris Hathhorn, Yilong Li, Edgar Pek, Manasvi Saxena, Traian Florin Serbanuta, and Grigore Rosu. Runtime verification at work: A tutorial. In *Runtime Verification - 16th International Conference, RV 2016 Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 46–67. Springer, September 2016.

[DH05] Marcelo DAmorim and Klaus Havelund. Jeagle: a JAVA Runtime Verification Tool. Technical Report 20050082002, NASA Ames Research Center, NASA Ames Research Center; Moffett Field, CA, United States, 2005.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, August 1975.

[Dim00] Jordan Dimitrov. Compositional Reasoning about Events in Interval Temporal Logic. In *Proc. of The Fifth International Conference on Computer Science and Informatics*, 2000.

[Dim02] Jordan Dimitrov. *Formal Compositional Design of Mixed Hardware/Software Systems with semantics of Verilog HDL.* PhD thesis, De Montfort University, 2002.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 27–39, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[Eme90] E. Allen Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and Modal Logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[FHR13] Yliès Falcone, Klaus Havelund, and Giles Reger. A Tutorial on Runtime Verification. In Manfred Broy, Doron Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 141–175. IOS Press, 2013. Summer School Marktoberdorf 2012.

[Fis06] Michael Fisher. Metatem: The story so far. In Rafael H. Bordini, Mehdi M. Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems: Third International Workshop, ProMAS 2005, Utrecht, The Netherlands, July 26, 2005, Revised and Invited Papers*, pages 3–22, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Fis11] M. Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic*. Wiley, 2011.

[For82] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligences*, 19(1):17–37, 1982.

[Hav11] K. Havelund. Closing the Gap Between Specification and Programming: VDM++ and Scala. In Andrei Voronkov and Margarita Korovina, editors, *Higher-Order Workshop on Automated Runtime Verification and Debugging*, volume 1 (first edition). EasyChair Proceedings, December 2011.

[Hav13] K. Havelund. A Scala DSL for Rete-based Runtime Verification. In *LNCS: The 4th International Conference on Runtime Verification (RV 2013)*, volume 8174. Springer Verlag, September 2013.

[Hav14] Klaus Havelund. Monitoring with data automata. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, pages 254–273, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[Hav15] Klaus Havelund. Rule-based runtime verification revisited. *Int. J. Softw. Tools Technol. Transf.*, 17(2):143–170, April 2015.

[Hav19] K. Havelund. Trace Contract. online: `https://github.com/havelund/tracecontract`, January 2019.

[Hie02] Rob Hierons. Editorial: Formal methods and testing. *Software Testing, Verification and Reliability*, 12(2):69–70, 2002.

[Hol04] G. J. Holzmann. *The Spin Model Checker.* Addison-Wesley, 2004.

[Jan10] Helge Janicke. ITL Tracer: Runtime Verification of Properties expressed in ITL (unpublished). 2010.

[Jav17] JavaMOP. JavaMOP4. online: `http://fsl.cs.illinois.edu/index.php/JavaMOP`, 2017.

[JCS+06] Helge Janicke, Antonio Cau, François Siewe, Hussein Zedan, and Kevin Jones. A Compositional Event & Time-Based Policy Model. In *7th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2006), 5-7 June 2006, London, Ontario, Canada*, pages 173–182, 2006. .

[JCSZ13] Helge Janicke, Antonio Cau, François Siewe, and Hussein Zedan. Dynamic Access Control Policies: Specification and Verification. *The Computer Journal*, 56(4):440–463, 2013. .

[Jin12] Dongyun Jin. *Making Runtime Monitoring of Parametric Properties Practical.* PhD thesis, University of Illinois at Urbana-Champaign, August 2012.

[JMLR12] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. JavaMOP: Efficient parametric runtime monitoring framework. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1427–1430. IEEE, 2012.

[JNWC15] Helge Janicke, Andrew Nicholson, Stuart Webber, and Antonio Cau. Runtime-monitoring for industrial control systems. *Electronics*, 4(4):995–1017, December 2015. Open Access.

[JZR+16] O. Javed, Y. Zheng, A. Rosà, H. Sun, and W. Binder. Extended code coverage for AspectJ-Based Runtime Verification Tools. In Y. Falcone and Sánchez C., editors, *Runtime Verification. RV 2016*, volume 10012 of *Lecture Notes in Computer Science*. Springer, 2016.

[Leu12] Martin Leucker. Teaching runtime verification. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, pages 34–48, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[LS09] Martin Leucker and Christian Schallhart. A Brief Account of Runtime Verification. *Journal of Logic and Algebraic Programming (JLAP)*, (78):293–303, 2009.

[MG17] Ben Moszkowski and Dimitar Guelev. An application of temporal projection to interleaving concurrency. *Formal Aspects of Computing*, 29(4):705–750, July 2017.

[MGL14] Ben Moszkowski, Dimitar Guelev, and Martin Leucker. Guest editors' preface to special issue on interval temporal logics. *Annals of Mathematics and Artificial Intelligence*, 71(1-3):1–9, July 2014.

[MM84] Ben Moszkowski and Zohar Manna. Reasoning in interval temporal logic. In Edmund Clarke and Dexter Kozen, editors, *Logics of Programs*, pages 371–382, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.

[Mos82] B. C. Moszkowski. A Temporal Logic for Multi-Level Reasoning About Hardware. Technical Report ADA324174, Stanford University, December 1982.

[Mos83] Ben Moszkowski. *Reasoning about Digital Circuits*. PhD thesis, Department of Computer Science, Stanford University, 1983.

[Mos86] B. C. Moszkowski. *Executing Temporal Logic Programs*. CUP, 1986.

[Mos94] Ben Moszkowski. Some very compositional temporal properties. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, volume A-56 of *IFIP Transactions*, pages 307–326. IFIP, Elsevier Science B.V. (North–Holland), 1994.

[Mos96a] B. C. Moszkowski. Using temporal fixpoints to compositionally reason about liveness. In He Jifeng, John Cooke, and Peter Wallis, editors, *BCS-FACS 7th Refinement Workshop*, electronic Workshops in Computing, London, 1996. BCS-FACS, Springer-Verlag and British Computer Society.

[Mos96b] Ben Moszkowski. The Programming Language Tempura. *J. Symb. Comput.*, 22(5-6):730–733, November 1996.

[Mos96c] Ben Moszkowski. Using temporal fixpoints to compositionally reason about liveness. In He Jifeng, John Cooke, and Peter Wallis, editors, *BCS-FACS 7th*

Refinement Workshop, electronic Workshops in Computing, London, 1996. BCS-FACS, Springer Verlag and British Computer Society. .

[Mos98] B. C. Moszkowski. Compositional Reasoning Using Interval Temporal Logic and Tempura. *Lecture Notes in Computer Science*, 1536:439–464, 1998.

[Mos13] Ben Moszkowski. Interconnections between classes of sequentially compositional temporal formulae. *Information Processing Letters*, 113(9):350 – 353, 2013.

[Mos14a] B. C. Moszkowski. Imperative reasoning in ITL (unpublished). 2014.

[Mos14b] Ben Moszkowski. Compositional reasoning using intervals and time reversal. *Annals of Mathematics and Artificial Intelligence*, 71(1-3):175–250, 2014.

[MP87] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 205–205, New York, NY, USA, 1987. ACM.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Temporal and Reactive Systems: Specification*. Springer-Verlag, 1992.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[oCM18] University of Cambridge and Technische Universität München. Isabelle. online: https://isabelle.in.tum.de, 2018.

[OSV16] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 3 edition, 2016.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:46–57, 1977.

[RH16] Giles Reger and Klaus Havelund. What is a trace? a runtime verification perspective. In *ISoLA 2016: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, Lecture Notes in Computer Science, pages 339–355, 2016.

[sbt19] sbt. Simple Build Tool. online: https://www.scala-sbt.org/documentation.html, 2019.

[Sca] Scala. Scala Parser Combinators. online: https://github.com/scala/scala-parser-combinators.

[Sca17] Scala. The Scala Homepage. online: http://www.scala-lang.org/, 2017.

[SCZ03a] François Siewe, Antonio Cau, and Hussein Zedan. A compositional framework for access control policies enforcement. In *Proceedings of the 2003 ACM Workshop on Formal Methods in Security Engineering*, FMSE '03, pages 32–42, New York, NY, USA, 2003. ACM. .

[SCZ03b] Monika Solanki, Antonio Cau, and Hussein Zedan. Introducing compositionality in webservice descriptions. In *Proceedings of the 3rd International Anwire Workshop on Adaptable Service Provision*. Springer Verlag, 2003. .

[Sie05] François Siewe. *A Compositional Framework for the Development of Secure Access Control Systems*. PhD thesis, De Montfort University, 2005.

[Spi01] J. M. Spivey. Z Notation: A Reference Manual, 2001.

[Spi17] Spin. The Spin Homepage. online: http://spinroot.com/spin/whatispin.html, 2017.

[STE+14] Gerhard Schellhorn, Bogdan Tofan, Gidon Ernst, Jörg Pfähler, and Wolfgang Reif. RGITL: A temporal logic framework for compositional reasoning about interleaved programs. *Annals of Mathematics and Artificial Intelligence*, 71(1-3):131–174, 2014.

[Wol81] Pierre Wolper. Temporal logic can be more expressive. In *Proceedings of the 22Nd Annual Symposium on Foundations of Computer Science*, SFCS '81, pages 340–348, Washington, DC, USA, 1981. IEEE Computer Society.

[Wya13] Derek Wyatt. *Akka Concurrency*. Artima Inc, 2013.

[YAH+16] Yoriyuki Yamagata, Cyrille Artho, Masami Hagiya, Jun Inoue, Lei Ma, Yoshinori Tanabe, and Mitsuharu Yamamoto. Runtime monitoring for concurrent systems. In *16th International Conference on Runtime Verification, RV 2016*, volume 10012 of *Lecture Notes in Computer Science*, pages 386–403, 2016. QC 20170119.

[ZZC05] Shikun Zhou, Hussein Zedan, and Antonio Cau. Run-time Analysis of Time-critical Systems. *Journal of Systems Architecture*, 51(5):331–345, 2005.

Appendices

Appendix A

API listings

A.1 ITL API

Listing A.1: ITL.scala

```
package runtime.analysis

object ITL {
  import scala.language.implicitConversions
  import scala.language.existentials
  import scala.language.postfixOps
  import scala.collection.immutable
  import scala.collection.immutable.ListMap
  import scala.collection.immutable.SortedMap

  trait Eq[T] {
    def EQ(a: T, b: T): Boolean
    def NE(a: T, b: T): Boolean = !(EQ(a,b))
  }

  trait Ord[T] extends Eq[T] {
    def LE(a: T, b: T): Boolean // minimal complete definition
    def LT(a: T, b: T): Boolean = LE(a,b) && NE(a,b)
    def GE(a: T, b: T): Boolean = LE(b,a)
    def GT(a: T, b: T): Boolean = LT(b,a)
  }

  trait Num[T] {
    def Add(a: T, b: T): T
    def Sub(a: T, b: T): T
    def Mul(a: T, b: T): T
    def Neg(a: T): T
    def Abs(a: T): T
    def Sgn(a: T): T
  }

  trait Integral[T] {
    def Div(a: T, b: T): T
    def Mod(a: T, b: T): T
  }

  trait Logical[T] {
    def Ltrue:  T
    def Lfalse: T
    def Listrue(b: T):  Boolean
    def Lisfalse(b: T): Boolean
    def Lnot(b: T): T           = if (Listrue(b)) Lfalse else Ltrue
    def Land( a: T, b: T): T    = if (Listrue(a) && Listrue(b)) Ltrue else Lfalse
```

```
  def Lor(  a: T, b: T): T    = Lnot(Land(Lnot(a),Lnot(b)))
  def Limp( a: T, b: T): T    = Lor(Lnot(a),b)
  def Leqv( a: T, b: T): T    = Land(Limp(a,b), Limp(b,a))
  def Lxor( a: T, b: T): T    = Lnot(Leqv(a,b))
  def Lnand(a: T, b: T): T    = Lnot(Land(a,b))
  def Lnor( a: T, b: T): T    = Lnot(Lor(a,b))
}

/* ***************************************************************************
* Implicit instances for Int
*****************************************************************************/

implicit object RelInt extends Eq[Int] with Ord[Int] {
  override def EQ(a: Int, b: Int): Boolean = a==b
  override def LE(a: Int, b: Int): Boolean = a<=b
}

implicit object NumInt extends Num[Int] {
  def Add(a: Int, b: Int): Int = a+b
  def Sub(a: Int, b: Int): Int = a-b
  def Mul(a: Int, b: Int): Int = a*b
  def Neg(a: Int): Int = -a
  def Abs(a: Int): Int = a.abs
  def Sgn(a: Int): Int = a.signum
}

implicit object IntegralInt extends Integral[Int] {
  def Div(a: Int, b: Int): Int = a/b
  def Mod(a: Int, b: Int): Int = a%b
}

implicit object LogicalInt extends Logical[Int] {
  def Ltrue:  Int = 1
  def Lfalse: Int = 0
  def Listrue(b: Int):  Boolean = b!=0
  def Lisfalse(b: Int): Boolean = b==0
}

/* ***************************************************************************
* Implicit instances for Long
*****************************************************************************/

implicit object RelLong extends Eq[Long] with Ord[Long] {
  override def EQ(a: Long, b: Long): Boolean = a==b
  override def LE(a: Long, b: Long): Boolean = a<=b
}

implicit object NumLong extends Num[Long] {
  def Add(a: Long, b: Long): Long = a+b
  def Sub(a: Long, b: Long): Long = a-b
  def Mul(a: Long, b: Long): Long = a*b
  def Neg(a: Long): Long = -a
  def Abs(a: Long): Long = a.abs
  def Sgn(a: Long): Long = a.signum
}

implicit object IntegralLong extends Integral[Long] {
  def Div(a: Long, b: Long): Long = a/b
  def Mod(a: Long, b: Long): Long = a%b
}

implicit object LogicalLong extends Logical[Long] {
  def Ltrue:  Long = 1
  def Lfalse: Long = 0
  def Listrue(b: Long):  Boolean = b!=0
  def Lisfalse(b: Long): Boolean = b==0
}

/* ***************************************************************************
* Implicit instances for Char
*****************************************************************************/

implicit object RelChar extends Eq[Char] with Ord[Char] {
  override def EQ(a: Char, b: Char): Boolean = a==b
```

```
    override def LE(a: Char, b: Char): Boolean = a<=b
  }

  implicit object LogicalChar extends Logical[Char] {
    def Ltrue:  Char = 'T'
    def Lfalse: Char = 'F'
    def Listrue(b: Char):  Boolean = b!='F'
    def Lisfalse(b: Char): Boolean = b=='F'
  }

  /* ***************************************************************************
  * Implicit instances for Boolean
  ****************************************************************************/

  implicit object RelBoolean extends Eq[Boolean] with Ord[Boolean] {
    override def EQ(a: Boolean, b: Boolean): Boolean = a==b
    override def LE(a: Boolean, b: Boolean): Boolean = a<=b
  }

  implicit object LogicalBoolean extends Logical[Boolean] {
    def Ltrue:  Boolean = true
    def Lfalse: Boolean = false
    def Listrue(b: Boolean):  Boolean = b
    def Lisfalse(b: Boolean): Boolean = !b
  }

  /* ***************************************************************************
  * Implicit conversions
  ****************************************************************************/

  implicit def implicit_T_To_Val_T[T](c: T): Val[T] = Val(c)
  implicit def implicit_T_To_Const_T[T](c: T): Expr[T] = Const(c)
  implicit def implicitVar_T_ToExpr[T] (v: Var[T]): Expr[T] = Ref(v)
  implicit def implicitVal_T_To_T[T](va: Val[T]): T = va match { case Val(a) => a }
  implicit def implicitVal_T_ToConst_T[T](c: Val[T]): Const[T] = Const(c)

  implicit def implicitExprBoolToFormula(x: Expr[Boolean]): Formula = Exp(x)
  implicit def implicitBoolToFormula(a: Boolean): Formula = Exp(Const(a))
  implicit def implicitVarBoolToFormula(v: Var[Boolean]): Formula = Exp(Ref(v))

  /* ***************************************************************************
   * Abstract Values and Variables
   ***************************************************************************/

  abstract trait Value
  abstract trait Variable {
    override def toString(): String = this.getClass.getName
  }

  abstract class Var[T] extends Variable {
    def <~  (x: Expr[T]) (implicit o:Eq[T]) = ptassign(this, x)
    def <-- (x: Expr[T]) (implicit o:Eq[T]) =  tassign(this, x)
    def :=  (x: Expr[T]) (implicit o:Eq[T]) =   assign(this, x)
    def gets(x: Expr[T]) (implicit o:Eq[T]) = ITL.gets(this, x)
  }

  case class Val[T](v: T) extends Value {
    def value: T = v
    override def toString(): String = v.toString()
  }

  type VarUpdate = (Var[T],T) forSome {type T}

  /* ***************************************************************************
   * Intervals
   ***************************************************************************/

  type IntervalRepresentation = immutable.Map[Variable,immutable.Map[Int,Value]]

  class Interval {
    val index: Int = -1
    val sigma: IntervalRepresentation = new immutable.HashMap()

    def firstIndex: Int = 0
```

```
    def lastIndex: Int = index

    def get[T](k: Int, v: Var[T]): Option[Val[T]]
      = sigma.get(v) match {
          case None => None
          case Some(trace) => (k to 0 by -1).toStream.find(trace.isDefinedAt(_)) match {
                                case None     => None
                                case Some(j) => trace.get(j).asInstanceOf[Option[Val[T]]]
                              }//match
          }//match

    def add(updates: VarUpdate* ): Interval = this.add(updates.toList)

    def add(updates: List[VarUpdate] ): Interval = {
      var s = sigma
      val i = index+1
      for ((v,a) <- updates) {
        val newtr: immutable.Map[Int,Value] = s.get(v) match {
                     case None => new immutable.HashMap() + ((i -> Val(a)))
                     case Some(oldtr) => oldtr + ((i -> Val(a)))
        }//match
        s = s + ((v -> newtr))
      }//for
      MakeInterval(s,i)
    }

    def finState: List[VarUpdate] = {
      def finVal[T](v: Var[T]): VarUpdate =
        this.get(lastIndex, v) match {
                                case None          => sys.error("finVal no match!")
                                case Some(Val(a)) => (v,a).asInstanceOf[VarUpdate]
                              }//match

      def getFinVal(v: Variable): VarUpdate =
        finVal(v.asInstanceOf[Var[T] forSome {type T}])

      ((sigma.keys) map getFinVal).toList
    }//finState

    def isEmpty(): Boolean = index==0

    def slice[T](v: Var[T]): String = {
      v.toString + ": " + ((0 to index).map{k => (k, this.get(k, v))}).toString
    }

    override def toString(): String = {
      def order(m: immutable.Map[Int,Value]) = m.toSeq.sortWith(_._1<_._1)
      val tau = sigma.mapValues(m => order(m)).toSeq.sortWith(_._1.toString < _._1.toString)
      tau.toString
    }
  }//Interval

  case class MakeInterval(s: IntervalRepresentation, i: Int) extends Interval {
    override val index: Int = i
    override val sigma: IntervalRepresentation = s
  }

  /* ***************************************************************************
   * Temporal Expressions, Values, Variables, and Formulae
   ****************************************************************************/

  abstract class Expr[T] {
    def unary_-                  (implicit o:Num[T]):       Expr[T]          = Unary(o.Neg,    this       )
    def abs                      (implicit o:Num[T]):       Expr[T]          = Unary(o.Abs,    this       )
    def sgn                      (implicit o:Num[T]):       Expr[T]          = Unary(o.Sgn,    this       )
    def *       (that: Expr[T])(implicit o:Num[T]):       Expr[T]          = Binary(o.Mul,   this, that)
    def /       (that: Expr[T])(implicit o:Integral[T]): Expr[T]          = Binary(o.Div,   this, that)
    def %       (that: Expr[T])(implicit o:Integral[T]): Expr[T]          = Binary(o.Mod,   this, that)
    def +       (that: Expr[T])(implicit o:Num[T]):       Expr[T]          = Binary(o.Add,   this, that)
    def -       (that: Expr[T])(implicit o:Num[T]):       Expr[T]          = Binary(o.Sub,   this, that)
    def `~~`    (that: Expr[T])(implicit o:Eq[T]):        Formula          = ITL.tempeq(     this, that)
    def `=`     (that: Expr[T])(implicit o:Eq[T]):        Expr[Boolean] = Binary(o.EQ,    this, that)
    def `/=`    (that: Expr[T])(implicit o:Eq[T]):        Expr[Boolean] = Binary(o.NE,    this, that)
    def <       (that: Expr[T])(implicit o:Ord[T]):       Expr[Boolean] = Binary(o.LT,    this, that)
```

```
    def >      (that: Expr[T])(implicit o:Ord[T]):       Expr[Boolean] = Binary(o.GT,    this, that)
    def <=     (that: Expr[T])(implicit o:Ord[T]):       Expr[Boolean] = Binary(o.LE,    this, that)
    def >=     (that: Expr[T])(implicit o:Ord[T]):       Expr[Boolean] = Binary(o.GE,    this, that)
    def unary_~                (implicit o:Logical[T]): Expr[T]       = Unary(o.Lnot,   this      )
    def &      (that: Expr[T])(implicit o:Logical[T]):  Expr[T]       = Binary(o.Land,  this, that)
    def &!     (that: Expr[T])(implicit o:Logical[T]):  Expr[T]       = Binary(o.Lnand, this, that)
    def |      (that: Expr[T])(implicit o:Logical[T]):  Expr[T]       = Binary(o.Lor,   this, that)
    def |!     (that: Expr[T])(implicit o:Logical[T]):  Expr[T]       = Binary(o.Lnor,  this, that)
    def |^     (that: Expr[T])(implicit o:Logical[T]):  Expr[T]       = Binary(o.Lxor,  this, that)
    def Implies(that: Expr[T])(implicit o:Logical[T]):  Expr[T]       = Binary(o.Limp,  this, that)
    def Equiv  (that: Expr[T])(implicit o:Logical[T]):  Expr[T]       = Binary(o.Leqv,  this, that)
    def injectInto[U](f: Const[T] => Expr[U])(implicit o:Eq[T])       = With(           this, f   )
  }//Expr

  case class Const[T](c: T)                                             extends Expr[T]
  case class Ref[T](v: Var[T])                                          extends Expr[T]
  case class Unary[T,U](op: T=>U, x: Expr[T])                           extends Expr[U]
  case class Binary[T,U,V](op: (T,U)=>V, x: Expr[T], y: Expr[U])        extends Expr[V]
  case class With[T,U](x: Expr[T], f: Const[T] => Expr[U])              extends Expr[U]
  case class Next[T](v: Var[T])                                         extends Expr[T]
  case class Fin[T](v: Var[T])                                          extends Expr[T]
  case class IntLen()                                                   extends Expr[Int]

  /* *****************************************************************************
   * Expression evaluation
   ******************************************************************************/

  def evalExpr[T](expr: Expr[T], sigma: Interval): Option[Const[T]] =
      evalExprFromTo(expr, sigma, sigma.firstIndex, sigma.lastIndex)

  def evalExprFromTo[T](expr: Expr[T],
                        sigma: Interval, i: Int, j: Int): Option[Const[T]] = {
    expr match {
      case Ref(v)          => sigma.get(i, v) match {
                                case Some(Val(a)) => Some(Const(a))
                                case None         => None
                              }
      case Const(a)        => Some(Const(a))
      case Unary(op,x)     => evalExprFromTo(x, sigma, i, j) match {
                                case None => None // strict!
                                case Some(Const(a)) => Some(Const(op(a)))
                              }
      case Binary(op,x,y) => evalExprFromTo(x, sigma, i, j) match {
                                case None => None // strict!
                                case Some(Const(a)) =>
                                     evalExprFromTo(y, sigma, i, j) match {
                                       case None => None // strict!
                                       case Some(Const(b)) => Some(Const(op(a, b)))
                                     }
                              }
      case With(x,f)       => evalExprFromTo(x, sigma, i, j) match {
                                case None    => None // strict!
                                case Some(k) => evalExprFromTo(f(k), sigma, i,j)
                              }
      case Next(v)         => if ((i+1) > j)
                                 None
                              else
                                sigma.get(i+1, v) match {
                                  case Some(Val(a)) => Some(Const(a))
                                  case None         => None
                                }
      case Fin(v)          => sigma.get(j, v) match {
                                case Some(Val(a)) => Some(Const(a))
                                case None         => None
                              }
      case IntLen()        => Some(Const(j-i))
    }//match
  }//evalExprFromTo

  /* *****************************************************************************
   * Formulae
   ******************************************************************************/

  abstract class Formula {
```

```scala
    def `;`(that: Formula)       = Chop(this,that)
    def chopstar                 = Chopstar(this)
    def times(n: Int)            = Repeat(n, this)  // use  f times 3  or  f.times(3)
    def and(that: Formula)       = And(this,that)
    def or(that: Formula)        = Not(And(not(this),not(that)))
    def implies(that: Formula) = not(this) or that  // does NOT have ITL priority/associativity
    def equiv(that: Formula)     = And(this implies that, that implies this)
    def afb(w: Formula)          = ITL.afb(this, w)
    def fixed(): Option[Int]     = this match {
                                       case Len(k)    => Some(k)              // fixed length
                                       case And(f,g) => f.fixed() match {
                                                          case None => g.fixed()
                                                          case Some(k) => Some(k)
                                                        }
                                       case _         => None
                                     }

    /* ***************************************************************************
     * Formula evaluation
     ****************************************************************************/

    def evalFormulaFromTo[T](sigma: Interval, i: Int, j: Int): Boolean = {
      this match {
        case Label(s, f) =>
             f.evalFormulaFromTo(sigma, i, j)

        case Exp(x) =>
             evalExprFromTo(x, sigma, i, j) match {
               case None => false
               case Some(Const(b)) => b
             }

        case Not(f) =>
             ! (f.evalFormulaFromTo(sigma, i, j))

        case Final(w) =>
             w.evalFormulaFromTo(sigma, j, j)

        case And(f, g) =>
             f.evalFormulaFromTo(sigma, i, j) && g.evalFormulaFromTo(sigma, i, j)

        case Len(n) =>
             Exp(IntLen() `=` Const(n)).evalFormulaFromTo(sigma, i, j)

        case Chop(f, g) => (f.fixed(), g.fixed()) match {
               case (None,None)         => (i to j).toStream.map(k =>
                                               if (f.evalFormulaFromTo(sigma, i, k))
                                                 g.evalFormulaFromTo(sigma, k, j)
                                               else
                                                 false).contains(true)
               case (Some(m),None)      => (i+m >= i && i+m <= j) &&
                                           f.evalFormulaFromTo(sigma, i, i+m) &&
                                           g.evalFormulaFromTo(sigma, i+m, j)
               case (None,Some(n))      => (j-n >= i && j-n <= j) &&
                                           f.evalFormulaFromTo(sigma, i, j-n) &&
                                           g.evalFormulaFromTo(sigma, j-n, j)
               case (Some(m),Some(n)) => (i+m >= i && i+m <= j) &&
                                           (i+m == j-n) &&
                                           f.evalFormulaFromTo(sigma, i, i+m) &&
                                           g.evalFormulaFromTo(sigma, i+m, j)
             }

        case Chopstar(f) => // ChopstarEqv |= f* == (empty \/ ((f /\ more) ; f*))
             if (empty.evalFormulaFromTo(sigma, i, j))
               true
             else // not empty implies more
               Chop((f and more), Chopstar(f)).evalFormulaFromTo(sigma, i, j)

        case Repeat(n, f) =>
             if (n==0)
               empty.evalFormulaFromTo(sigma, i, j)
             else
               Chop(f, Repeat(n-1, f)).evalFormulaFromTo(sigma, i, j)

```

```
        case AllOf(xs, f) => ((xs map f).foldLeft(TRUE) (_ and _)).evalFormulaFromTo(sigma, i, j)
        case AnyOf(xs, f) => ((xs map f).foldLeft(FALSE)(_ or  _)).evalFormulaFromTo(sigma, i, j)
      }//match
    }//evalFormulaFromTo

    def evalFormula(sigma: Interval): Boolean =
        this.evalFormulaFromTo(sigma, sigma.firstIndex, sigma.lastIndex)
  }//Formula

  /* ****************************************************************************
   * Derived formulae
   ****************************************************************************/

  case class Exp      (x: Expr[Boolean])                         extends Formula
  case class Not      (f: Formula)                               extends Formula
  case class Final    (f: Formula)                               extends Formula
  case class And      (f: Formula, g: Formula)                   extends Formula
  case class Len      (n: Int)                                   extends Formula {
    override def toString = "Len("+n+")"
  }
  case class Chop     (f1: Formula, f2: Formula)                 extends Formula
  case class Chopstar(f: Formula)                                extends Formula
  case class Repeat   (n: Int, f: Formula)                       extends Formula
  case class AllOf[T](xs: List[T], f: T => Formula)              extends Formula
  case class AnyOf[T](xs: List[T], f: T => Formula)              extends Formula
  case class Label    (s: String, f: Formula)                    extends Formula {
    override def toString = "Formula(" + s + ")"
  }
  def label(s: String, f: Formula)                      = Label(s, f)
  def anyof[T](xs: List[T], f: T => Formula)            = AnyOf(xs, f)
  def allof[T](xs: List[T], f: T => Formula)            = AllOf(xs, f)
  val TRUE: Formula                                     = Exp(Const(true))
  val FALSE: Formula                                    = Exp(Const(false))
  def len(n: Int): Formula                              = Len(n)
  val empty: Formula                                    = Len(0)
  val Empty: Formula                                    = Len(0)
  val skip: Formula                                     = Len(1)
  def chop(f1: Formula, f2: Formula): Formula           = Chop(f1, f2)
  def repeat(n: Int, f: Formula): Formula               = Repeat(n, f)
  def not(f: Formula): Formula                          = f match { case (Not(g)) => g
                                                                    case _        => Not(f) }
  def next(f: Formula): Formula                         = chop(skip, f)
  def strongnext(f: Formula): Formula                   = next(f)
  def weaknext(f: Formula): Formula                     = empty or strongnext(f)
  val more: Formula                                     = next(TRUE)
  def eventually(f: Formula): Formula                   = chop(TRUE, f)
  def always(f: Formula): Formula                       = not(eventually(not(f)))
  def di(f:    Formula): Formula                        = chop(f, TRUE)
  def bi(f:    Formula): Formula                        = not(di(not(f)))
  def da(f:    Formula): Formula                        = chop(chop(TRUE, f), TRUE)
  def ba(f:    Formula): Formula                        = not(da(not(f)))
  def bs(f:    Formula): Formula                        = empty or chop(bi(f),skip)
  def ds(f:    Formula): Formula                        = not(bs(not(f)))
  def bm(f:    Formula): Formula                        = always(more implies f) // from Ben M
  def dm(f:    Formula): Formula                        = eventually(more and f)  // from Ben M
  def fst(f:   Formula): Formula                        = f and bs(Not(f))
  def fin(f:   Formula): Formula                        = Final(f)
  def halt(f: Formula): Formula                         = always(empty equiv f)
  def keep(f: Formula): Formula                         = ba(skip implies f)
  def afb(f:   Formula,
          w:   Formula): Formula                        = bi(eventually(f) implies fin(w)) // f afb g
  def tempeq[T](x: Expr[T], y: Expr[T])                                                   //x ‘~~‘ y
      (implicit o:Eq[T])                                = always(Exp(Binary(o.EQ, x, y)))
  def tassign[T](v: Var[T], x: Expr[T])                                                   // v <-- x
      (implicit o:Eq[T])                                = Exp(Binary(o.EQ,x,Fin(v)))
  def assign[T](v: Var[T], x: Expr[T])                                                    //  v := x
      (implicit o:Eq[T])                                = Exp(Binary(o.EQ,x,Next(v)))
  def gets[T](v: Var[T], x: Expr[T])                                                      //v gets x
      (implicit o:Eq[T])                                = keep(tassign(v, x))
  def stable[T](v: Var[T])
      (implicit o:Eq[T])                                = gets(v,Ref(v))
  def padded[T](v: Var[T])
      (implicit o:Eq[T])                                = empty or chop(stable(v),skip)
  def ptassign[T](v: Var[T], x: Expr[T])                                                 //  v <~ x
```

```scala
      (implicit o:Eq[T])                        = tassign(v,x) and padded(v)
  def ifThenElse (f: Formula,
                  g: Formula,
                  h: Formula): Formula          = (f and g) or ((not(f) and h))
  def ifThen     (f: Formula,
                  g: Formula): Formula          = ifThenElse(f, g, empty)
  def forDo      (n: Int,
                  f: Formula): Formula          = f.times(n)
  def whileDo    (f: Formula,
                  g: Formula): Formula          = (f and g).chopstar and fin(not(f))
  def repeatUntil(f: Formula,
                  g: Formula): Formula          = chop(f, whileDo(not(g),f))

}//ITL
```

A.2 Monitor API

Listing A.2: Monitor.scala

```scala
package runtime.analysis
/*************************************************************************************/
object Monitor {
  import scala.language.implicitConversions
  import scala.language.postfixOps
  import scala.concurrent.Await
  import scala.concurrent.duration._
  import akka.actor._
  import akka.event.Logging
  import akka.util.Timeout
  import scala.concurrent.Future
  import akka.pattern.ask
  import akka.actor.Status.Failure
  import runtime.analysis.ITL._ // {Formula,Final,Interval,VarUpdate}

  implicit val timeout: Timeout = Timeout(600 seconds)

  object Protocol { /* Communication protocol between monitors and clients */

    abstract class Request

    case class  Step(updates: List[VarUpdate]) extends Request
    case class  Show(indent: Int)              extends Request

    abstract class Reply {                    def isDone: Boolean = false
                                              def isMore: Boolean = false
                                              def isFail: Boolean = false }

    case object Fail extends Reply { override def isFail: Boolean = true }
    case object More extends Reply { override def isMore: Boolean = true }
    case class  Done(updates: List[VarUpdate]) extends Reply {
                                     override def isDone: Boolean = true }

    case object Tick //internal acknowledgement only
  }//Protocol

  object OptimisationFlags {
    class OpTy
    case object ANY_STATE     extends OpTy
    case object ALL_STATES    extends OpTy
    case object ANY_PREFIX    extends OpTy
    case object ALL_PREFIXES  extends OpTy
    case object CHECK_ONCE    extends OpTy
  }//OptimisationFlags

  /* **************************************************************************
  * Abstract monitors link to the client and are expression trees.
  ****************************************************************************/

```

```scala
object Abstr {
  import OptimisationFlags._

  class Monitor {
    /* ***********************************************************************
     * PUBLIC INTERFACE infix binary operators
     ************************************************************************/

    def ::(name:               String): Monitor = Name(name,this)
    def UPTO(that:            Monitor): Monitor = Upto(this,that)
    def THRU(that:            Monitor): Monitor = Thru(this,that)
    def THEN(that:            Monitor): Monitor = Then(this,that)
    def AND(that:             Monitor): Monitor = And(this,that)
    def ITERATE(that:         Monitor): Iterate = Iterate(this,that)
    def WITH(opt : OpTy,  f: Formula): Monitor = With(this,opt,f)
    def WITH(              f :Formula): Monitor = With(this,CHECK_ONCE,f)
    def TIMES(k:                  Int): Monitor = if (k==0) EMPTY
                                                  else this THEN (this TIMES (k-1))
    def ALWAYS(w:             Formula): Monitor = With(this,ALL_STATES,w)
    def SOMETIME(w:           Formula): Monitor = With(this,ANY_STATE,w)
    def WITHIN(f:             Formula): Monitor = With(this,ALL_PREFIXES,
                                                       more implies (not(f) `;` skip))
    // Expirimental

    def INTERRUPT(i: Var[Int], bs: List[Monitor]): Monitor = {
      val interrupts: List[Monitor] =
                (EMPTY::bs).zipWithIndex.map(t => GUARD(i `=` t._2) THEN t._1)
      this UPTO (FIRST(Fin(i)>0)) THEN interrupts.reduceLeft((m,n) => m.UPTO(n))
    }
  }//Abstr.Monitor

  /* ***********************************************************************
   * PUBLIC INTERFACE prefix operators
   ************************************************************************/

  def FIRST(f: Formula)                    = First(ANY_PREFIX, f)
  def LEN(k: Int)                          = FIRST(len(k))
  def SKIP                                 = LEN(1)
  def EMPTY                                = FIRST(empty)
  def FAIL                                 = FIRST(false) WITHIN (empty)
  def HALT(w: Formula)                     = First(ANY_STATE, w)
  def SKIPTO(w: Formula)                   = SKIP THEN HALT(w)
  def GUARD(w: Formula)                    = EMPTY WITH (w)
  def UNTIL (w1 : Formula, w2: Formula)    = HALT(w2) WITH (bm(w1))

  /* ***********************************************************************
   * Abstract monitor representation
   ************************************************************************/

  case class Name    (name: String, a: Monitor)         extends Monitor
  case class First   (opt: OpTy,    f: Formula)         extends Monitor
  case class Upto    (a: Monitor,   b: Monitor)         extends Monitor
  case class Thru    (a: Monitor,   b: Monitor)         extends Monitor
  case class Then    (a: Monitor,   b: Monitor)         extends Monitor
  case class And     (a: Monitor,   b: Monitor)         extends Monitor
  case class Iterate(a: Monitor,    b: Monitor)         extends Monitor {
    def PROJECT(p: Monitor) = Project(this.a, this.b, p)
  }
  case class Project(a: Monitor, b: Monitor, p: Monitor) extends Monitor
  case class With    (a: Monitor, opt: OpTy, f: Formula) extends Monitor
}//Abstr

/* ***********************************************************************
 * Concrete monitors are Akka actors.  Each one is an autonomous agent with its own
 * state sequence (interval). A concrete monitor reacts to state updates.  A cover
 * function, step, is provided in the API which enables the concrete actors to be
 * hidden from the external program.  The user supplies an abstract monitor to
 * API.Monitor which, in turn, creates the underlying concrete monitors on a
 * by-needs basis.  The use of Akka actors enables the concrete monitors to be
 * distributed across cores/nodes.  Synchronous message alreadyPassed is necessary
 * to maintain the interaction with the program being verified - i.e. the main
 * program needs to check the result of the last state change before moving on.
 * (Future work may be able to relax this - returning futures for eg).
 ************************************************************************/
```

```

private object Concr {
  import Protocol._
  import OptimisationFlags._

  abstract class Monitor extends Actor with ActorLogging {
    def indent(i: Int): Int = i + 4
    def tab(i: Int) { for (_ <- 1 to i) print(' ') }

    override def preStart() { log.debug("\nStarting " + this) }
    override def postStop() { log.debug("\nStopping " + this) }

    /* ***************************************************************************
    * zombie represents the state of a monitor that can only be stopped. The
    * stop message comes from a call to context stop ... thus sending an Akka stop
    * message.  The zombie process will react to a Show message by stating
    * that this monitor has been closed down (awaiting stop) and to any other
    * message - that it should not receive - by printing an alarm on the terminal.
    ****************************************************************************/

    def zombie: Receive = {
      case Show(_) => log.error("Trying to Show a zombie process");         sender ! Tick
      case msg => log.error("zombie monitor " + this + "received " + msg); sender ! Tick
    }//zombie

    /* ***************************************************************************
    * rogue represents the state of a monitor that has received a message that is
    * outside the known protocol.  The monitor does not know how to handle the
    * message and it is unsafe to react to it in any way. The monitoring protocol
    * has detected a serious error and cannot continue. The rogue process will
    * react to a Show message by stating that this monitor has "gone rogue" and
    * to any other message by printing an alarm on the terminal. Needless to say,
    * we don't expect to get into this state, but it will help to locate a serious
    * problem if it ever occurs.
    ****************************************************************************/

    def rogue: Receive = {
      case Show(_) =>log.error("Trying to Show a rogue process");          sender ! Tick
      case msg => log.error("rogue monitor " + this + "received " + msg); sender ! Tick
    }//rogue
  }//Concr.Monitor

  /* ***************************************************************************
  * Every abstract monitor has a concrete counterpart.  The abstract monitors form
  * an expression tree and this tree forms the specification (top level monitor)
  * that the client provides.  When an abstract monitor is called upon then its
  * concrete counterpart (an actor) has to be initiated.  Note that subtrees (i.e.
  * subordinate abstract monitors) are passed as parameters to the actors and they
  * are, in turn, initiated on a by-needs basis. It is not possible for an entire
  * abstract expression tree to be represented as actors initially because the tree
  * 'evolves' over time (i.e. the THEN operator) so it is appropriate that the
  * future evaluation is passed in abstract form ready to be interpreted whenever
  * required.
  ****************************************************************************/

  def startUp(mu: Abstr.Monitor, context: ActorContext): ActorRef = mu match {
    case Abstr.Name(n, Abstr.Name(m,a)) => startUp(Abstr.Name(n+"::"+m, a), context)
    case Abstr.Name(n, a)               => startUp1(n,      a,  context)
    case _                              => startUp1("anon", mu, context)
  }//startup

  def startUp1(name: String, mu: Abstr.Monitor, context: ActorContext): ActorRef = mu match {
    case Abstr.First(o,f)     => context.actorOf(Props(classOf[Concr.First],   name, o, f   ))
    case Abstr.Upto(a,b)      => context.actorOf(Props(classOf[Concr.Upto],    name, a, b   ))
    case Abstr.Thru(a,b)      => context.actorOf(Props(classOf[Concr.Thru],    name, a, b   ))
    case Abstr.Then(a,b)      => context.actorOf(Props(classOf[Concr.Then],    name, a, b   ))
    case Abstr.With(a,o,f)    => context.actorOf(Props(classOf[Concr.With],    name, a, o, f))
    case Abstr.And(a,b)       => context.actorOf(Props(classOf[Concr.And],     name, a, b   ))
    case Abstr.Iterate(a,b)   => context.actorOf(Props(classOf[Concr.Iterate], name, a, b   ))
    case Abstr.Project(a,b,p)=> context.actorOf(Props(classOf[Concr.Project], name, a, b, p))
  }//startup1

  /* ***************************************************************************
   * Monitor class: FIRST
```

```
 * This monitor continually checks to see if the formula is satisfied.  As soon
 * as it is then this is the first occurrence of an interval that satisfies the
 * formula and DONE is returned.  For all the preceding initial subintervals the
 * monitor returns MORE indicating that more states are required.  This monitor
 * cannot FAIL because it either finds the first inital subinterval satisfying
 * the formula or it keeps looking.  It is possible for this monitor NOT to
 * terminate if the formula is never satisfied.  This monitor will be shut down
 * by its parent.
 ***************************************************************************/

case class First(name: String, opt: OpTy, f: Formula) extends Monitor {
  // Extend the interval until the first time that sigma |= f

  var sigma: Interval = new Interval  // The interval so far

  override def preStart() { log.debug("\nStarting " + this) }

  override def receive = {

    case Show(i) => tab(i)
                    println("FIRST(" + f + ")" + "sigma = " + sigma)
                    sender ! Tick

    case Step(u) => opt match {
                      case ANY_STATE  => sigma = (new Interval).add(sigma.finState++u)
                      case ANY_PREFIX => sigma = sigma.add(u)
                    }
                    if (f.evalFormula(sigma)) {
                      First.update(log, f, sigma.lastIndex+1)
                      sender ! Done(sigma.finState)
                      context.become(this.zombie)
                    }
                    else
                      sender ! More

    case _       => sender ! Fail;
                    log.error("Unknown request - actor: " + this.toString)
                    context.become(this.rogue)
  }//receive
}//First

/* ******************************************************************************
 * This companion object maintains state common to all First occurrences (like static
 * attributes and methods in Java). Each time a first occurrence terminate successfully
 * this is recorded along with the number of states in the (sub)interval so that average
 * interval length data can be accumulated an reported.
 ********************************************************************************/

object First {
  import scala.math._
  var totStates: Int = 0
  var totFirsts: Int = 0
  var minLen:    Int = Int.MaxValue
  var maxLen:    Int = 0
  var avgLen:    Int = 0
  def update(log: akka.event.LoggingAdapter, f: Formula, numStates: Int) {
    totStates += numStates
    totFirsts += 1
    minLen = min(numStates, minLen)
    maxLen = max(numStates, maxLen)
    if (totFirsts > 0) avgLen  = round (totStates.toFloat / totFirsts.toFloat)
    log.debug(s"LOG FST DONE $totFirsts: this interval has $numStates states: ${f.toString}")
    log.debug(s"LOG FST STATES: AVG($avgLen), TOT($totStates), MIN($minLen), MAX($maxLen)")
  }//update
}//First

/* ****************************************************************************
 * Monitor class: a WITH f
 * This monitor runs monitor a alongside checking f. However, depending upon the
 * supplied optimisation parameter various optimisations may occur. This includes
 * two cases in which previous states don't need to be stored explicitly. Some
 * formulae benefit from being evaluated alongside monitor a whereas others do
 *  not. In the latter case the evaluation of f takes place once when a has
 * completed. The analysis with f adapts for each of the following patterns:
```

```scala
   * (ALL_STATES, w)    ==> Only needs last state. If ~w holds return FAIL
   * (ANY_STATE, w)     ==> Only needs last state. If  w holds return PASS always
   * (ALL_PREFIXES, f)  ==> Needs whole interval.  If ~f holds return FAIL
   * (ANY_PREFIX, f)    ==> Needs whole interval.  If  f holds return PASS always
   * (CHECK_ONCE, f)    ==> Needs whole interval. Only check f if/when a is DONE
   * Mathematically:
   * (ALL_STATES, w)    == sigma |= [i] (fin(w))  or  sigma |= [] w
   * (ANY_STATE, w)     == sigma |= <i> (fin(w))  or  sigma |= <> w
   * (ALL_PREFIXES, f)  == sigma |= [i] f
   * (ANY_PREFIX, f)    == sigma |= <i> f
   * (CHECK_ONCE, f)    == sigma |= f
   ****************************************************************************/

  case class With(name: String, a: Abstr.Monitor, opt: OpTy, f: Formula) extends Monitor {

    var c: ActorRef = _   // c is concrete counterpart to a
    var sigma: Interval = new Interval  // The interval so far
    var alreadyDone = false
    var sigmaSatisfiesF = false

    override def preStart() {
      log.debug("\nStarting " + this)
      c = Concr.startUp(a, context)
    }

    override def receive = {

      case Show(i) => tab(i)
                      println("WITH")
                      Await.result(ask(c, Show(indent(i))), timeout.duration)
                      sender ! Tick

      case Step(u) => if (!alreadyDone) opt match {
                          case ALL_STATES
                             | ANY_STATE => sigma = (new Interval).add(sigma.finState++u)
                          case _         => sigma = sigma.add(u)
                      }//match
                      val cf = ask(c, Step(u))  // copy new state to c
                      Await.result(cf, timeout.duration).asInstanceOf[Reply] match {
                        case Done(s) => if (alreadyDone || f.evalFormula(sigma)) {
                                          sender ! Done(s)
                                          context stop c
                                          context.become(this.zombie)
                                        }
                                        else {
                                          sender ! Fail
                                          log.warning(s"($name)WITH: RHS failed")
                                          context stop c
                                          context.become(this.zombie)
                                        }

                      case More => opt match {
                          case ALL_STATES
                             | ALL_PREFIXES => if (f.evalFormula(sigma))
                                                 sender ! More
                                               else {
                                                sender ! Fail
                                                log.warning(s"($name)WITH(in): Prefix violation")
                                                context stop c
                                                context.become(this.zombie)
                                               }

                          case ANY_STATE
                             | ANY_PREFIX   => if (!alreadyDone) {
                                                 alreadyDone = f.evalFormula(sigma)
                                                 sender ! More
                                               }

                          case CHECK_ONCE   => sender ! More
                          }//match

                      case Fail      => sender ! Fail
                                        log.warning(s"($name)WITH: LHS failed")
                                        context stop c
                                        context.become(this.zombie)
```

```
                        }//match

        case _ => sender ! Fail;
                  log.error("Unknown request - actor: " + this.toString)
                  context.become(this.rogue)
      }//receive
    }//With


    /* ****************************************************************************
     * Monitor class: a UPTO b
     * Either a or b must be satisfied.  The length of the interval consumed is the
     * shortest interval that satisfies a or b (or both).
     *****************************************************************************/

    case class Upto(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor {
      var c: ActorRef = _  // c is concrete counterpart to a
      var d: ActorRef = _  // d is concrete counterpart to b

      override def preStart() {
        log.debug("\nStarting " + this)
        c = Concr.startUp(a, context)
        d = Concr.startUp(b, context)
      }

      override def receive =
      {
        case Show(i) => tab(i)
                        println("UPTO")
                        Await.result(ask(c, Show(indent(i))), timeout.duration)
                        Await.result(ask(d, Show(indent(i))), timeout.duration)
                        sender ! Tick

        case Step(u) =>
          val cf = ask(c, Step(u))  // copy new state to c
          val df = ask(d, Step(u))  // copy new state to d
          Await.result(cf, timeout.duration).asInstanceOf[Reply]
          match {
            case Done(s)
              => Await.result(df, timeout.duration).asInstanceOf[Reply] // redundant   ??? Why
                 sender ! Done(s)
                 context stop c
                 context stop d
                 context.become(this.zombie)

            case More
              => Await.result(df, timeout.duration).asInstanceOf[Reply]
                 match {
                   case Done(s)  => sender ! Done(s)
                                    context stop c
                                    context stop d
                                    context.become(this.zombie)

                   case More     => sender ! More

                   case Fail     => sender ! More
                                    log.warning(s"($name)UPTO: RHS failed")
                                    context stop d
                                    context.become(singleBranchC)
                 }

            case Fail
              => Await.result(df, timeout.duration).asInstanceOf[Reply]
                 match {
                   case Done(s)  => sender ! Done(s)
                                    context stop c
                                    context stop d
                                    context.become(this.zombie)

                   case More     => sender ! More
                                    context stop c
                                    context.become(singleBranchD)

                   case Fail     => sender ! Fail
```

```
                                        log.warning(s"($name)UPTO: LHS & RHS failed")
                                        context stop c
                                        context stop d
                                        context.become(this.zombie)
                    }//match
              }//match

          case _ => sender ! Fail
                    log.error("Unknown request - actor: " + this.toString)
                    context.become(this.rogue)
      }//receive

      def singleBranchC: Receive = {
          case Show(i) => tab(i)
                          println("UPTO-l")
                          Await.result(ask(c, Show(indent(i))), timeout.duration)
                          sender ! Tick

          case Step(u) =>
            val cf = ask(c, Step(u))
            Await.result(cf, timeout.duration).asInstanceOf[Reply]
            match {
              case Done(s)  => sender ! Done(s)
                               context stop c
                               context.become(this.zombie)

              case More     => sender ! More

              case Fail     => sender ! Fail
                               log.warning(s"($name)UPTO: LHS failed")
                               context stop c
                               context.become(this.zombie)
            }//match

          case _ => sender !
                    Fail; log.error("Unknown request - actor: " + this.toString)
                    context.become(this.rogue)
      }//singleBranchC

      def singleBranchD: Receive = {
          case Show(i) => tab(i)
                          println("UPTO-r")
                          Await.result(ask(d, Show(indent(i))), timeout.duration)
                          sender ! Tick

          case Step(u) =>
            val df = ask(d, Step(u))
            Await.result(df, timeout.duration).asInstanceOf[Reply]
            match {
              case Done(s)  => sender ! Done(s)
                               context stop d
                               context.become(this.zombie)

              case More     => sender ! More

              case Fail     => sender ! Fail
                               log.warning(s"($name)UPTO: RHS failed")
                               context stop d
                               context.become(this.zombie)
            }//match

          case _ => sender ! Fail;
                    log.error("Unknown request - actor: " + this.toString)
                    context.become(this.rogue)

      }//singleBranchD
    }//Upto

    /* ****************************************************************************
     * Monitor class: a THRU b
     * Both a and b must be satisfied for some prefix interval.  The length of the
     * interval consumed is the shortest interval that contains both prefixes.
     ******************************************************************************/

```

```
    case class Thru(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor {
      var c: ActorRef = _  // c is concrete counterpart to a
      var d: ActorRef = _  // d is concrete counterpart to b

      override def preStart() {
        log.debug("\nStarting " + this)
        c = Concr.startUp(a, context)
        d = Concr.startUp(b, context)
      }

      override def receive = {
        case Show(i) => tab(i)
                        println("THRU")
                        Await.result(ask(c, Show(indent(i))), timeout.duration)
                        Await.result(ask(d, Show(indent(i))), timeout.duration)
                        sender ! Tick

        case Step(u) =>
          val cf = ask(c, Step(u))  // copy new state to c
          val df = ask(d, Step(u))  // copy new state to d
          (Await.result(cf, timeout.duration).asInstanceOf[Reply],
           Await.result(df, timeout.duration).asInstanceOf[Reply])
          match {
            case (Done(s),  Done(_))  => sender ! Done(s)
                                         context stop c
                                         context stop d
                                         context.become(this.zombie)

            case (More,     More    ) => sender ! More

            case (More,     Done(_))  => sender ! More
                                         context stop d
                                         context.become(singleBranchC)

            case (Done(_),  More    ) => sender ! More
                                         context stop c
                                         context.become(singleBranchD)

            case (Fail,     Fail)    => sender ! Fail
                                         log.warning(s"($name)THRU: LHS & RHS failed")
                                         context stop c
                                         context stop d
                                         context.become(this.zombie)

            case (Fail    ,  _      ) => sender ! Fail
                                         log.warning(s"($name)THRU: LHS failed")
                                         context stop c
                                         context stop d
                                         context.become(this.zombie)

            case (_,        Fail    ) => sender ! Fail
                                         log.warning(s"($name)THRU: RHS failed")
                                         context stop c
                                         context stop d
                                         context.become(this.zombie)

            case (r1,       r2      ) => log.error(s"($name)THRU: unexpected ($r1,$r2)")
          }

        case _ => sender ! Fail;
                  log.error("Unknown request - actor: " + this.toString)
                  context.become(this.rogue)
      }//receive

      def singleBranchC: Receive =
      {
        case Show(i) => tab(i)
                        println("THRU-1")
                        Await.result(ask(c, Show(indent(i))), timeout.duration)
                        sender ! Tick

        case Step(u) =>
          val cf = ask(c, Step(u))
          Await.result(cf, timeout.duration).asInstanceOf[Reply]
```

```
        match {
          case Done(s)  => sender ! Done(s)
                           context stop c
                           context.become(this.zombie)

          case More     => sender ! More

          case Fail     => sender ! Fail
                           log.warning(s"($name)THRU: LHS failed")
                           context stop c
                           context.become(this.zombie)
        }//match

      case _ => sender ! Fail;
                log.error("Unknown request - actor: " + this.toString)
                context.become(this.rogue)
    }//singleBranchC

    def singleBranchD: Receive = {
      case Show(i) => tab(i)
                      println("THRU-r")
                      Await.result(ask(d, Show(indent(i))), timeout.duration)
                      sender ! Tick

      case Step(u) =>
        val df = ask(d, Step(u))
        Await.result(df, timeout.duration).asInstanceOf[Reply]
        match {
          case Done(s) => sender ! Done(s)
                          context stop d
                          context.become(this.zombie)

          case More    => sender ! More

          case Fail    => sender ! Fail
                           log.warning(s"($name)THRU: RHS failed")
                           context stop d
                           context.become(this.zombie)

        }//match

      case _ => sender ! Fail;
                log.error("Unknown request - actor: " + this.toString)
                context.become(this.rogue)
    }//singleBranchD
  }//Thru

  /* ***************************************************************************
   * Monitor class: a AND b
   * Both a and b must be satisfied by the same interval.
   ****************************************************************************/

  case class And(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor {
    var c: ActorRef = _  // c is concrete counterpart to a
    var d: ActorRef = _  // d is concrete counterpart to b

    override def preStart() {
      log.debug("\nStarting " + this)
      c = Concr.startUp(a, context)
      d = Concr.startUp(b, context)
    }

    override def receive = {
      case Show(i) => tab(i)
                      println("AND")
                      Await.result(ask(c, Show(indent(i))), timeout.duration)
                      Await.result(ask(d, Show(indent(i))), timeout.duration)
                      sender ! Tick

      case Step(u) =>
        val cf = ask(c, Step(u))  // copy new state to c
        val df = ask(d, Step(u))  // copy new state to d
        (Await.result(cf, timeout.duration).asInstanceOf[Reply],
         Await.result(df, timeout.duration).asInstanceOf[Reply])
```

```
          match {
            case (Done(s),
                  Done(_)) => sender ! Done(s)
                              context stop c
                              context stop d
                              context.become(this.zombie)

            case (More,
                  More    ) => sender ! More

            case (More,
                  Done(_)) => sender ! Fail
                              log.warning(s"($name)AND: RHS premature")
                              context stop c
                              context stop d
                              context.become(this.zombie)

            case (Done(_),
                  More    ) => sender ! Fail
                              log.warning(s"($name)AND: LHS premature")
                              context stop c
                              context stop d
                              context.become(this.zombie)

            case (r1,
                  r2      ) => sender ! Fail
                              (r1,r2) match {
                                case (Fail,Fail) => log.warning(s"($name)AND: LHS & RHS failed")
                                case (Fail,_    ) => log.warning(s"($name)AND: LHS failed")
                                case (_    ,Fail) => log.warning(s"($name)AND: RHS failed")
                                case (_,    _    ) => log.error(s"($name)AND: unexpected ($r1,$r2)")
                              }
                              context stop c
                              context stop d
                              context.become(this.zombie)
          }//match

        case _ => sender ! Fail;
                  log.error("Unknown request - actor: " + this.toString)
                  context.become(this.rogue)
      }//receive
    }//And

    /* ***************************************************************************
     * Monitor class: a THEN b
     * Once a is satisfied control immediately switches to b. The shared state must
     * be checked (end of a, start of b) when the change over occurs.
     *****************************************************************************/

    case class Then(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor {
      var c: ActorRef = _   // c is concrete counterpart to a (initially) - it may
                            // become the concrete counterpart to b (later)

      override def preStart() {
        log.debug("\nStarting " + this)
        c = Concr.startUp(a, context)
      }

      override def receive = {
        case Show(i) => tab(i)
                        println("THEN")
                        Await.result(ask(c, Show(indent(i))), timeout.duration)
                        sender ! Tick

        case Step(u) =>
          val cf = ask(c, Step(u))  // copy new state to c
          Await.result(cf, timeout.duration).asInstanceOf[Reply]
          match {
            case Fail     => sender ! Fail
                             log.warning(s"($name)THEN: LHS failed")
                             context stop c
                             context.become(this.zombie)

            case More     => sender ! More
```

```

          case Done(s)  => context stop c
                           c = Concr.startUp(b, context) // replace c with concr(b)
                           val cf = ask(c, Step(s))      // copy shared state to c
                           Await.result(cf, timeout.duration).asInstanceOf[Reply]
                           match {
                             case Done(s) => sender ! Done(s)
                                             context stop c
                                             context.become(this.zombie)

                             case More    => sender ! More
                                             context.become(receive2)

                             case Fail    => sender ! Fail
                                             log.warning(s"($name)THEN: RHS failed in 1st state ")
                                             context stop c
                                             context.become(this.zombie)
                           }//match
        }//match

      case _ => sender ! Fail;
                log.error("Unknown request - actor: " + this.toString)
                context.become(this.rogue)
    }//receive

    def receive2: Receive = {
      case Show(i) => tab(i)
                      println("THEN2")
                      Await.result(ask(c, Show(indent(i))), timeout.duration)
                      sender ! Tick

      case Step(u) => val cf = ask(c, Step(u))  // copy new state to c
                      Await.result(cf, timeout.duration).asInstanceOf[Reply]
                      match {
                        case Done(s)  => sender ! Done(s)
                                         context stop c
                                         context.become(this.zombie)

                        case More     => sender ! More

                        case Fail     => sender ! Fail
                                         log.warning(s"($name)THEN: RHS failed")
                                         context stop c
                                         context.become(this.zombie)
                      }//match

      case _       => sender ! Fail;
                      log.error("Unknown request - actor: " + this.toString)
                      context.become(this.rogue)
    }//receive2
  }//Then

  /* ***********************************************************************
   * Monitor class: a ITERATE b
   * Performs a WITH (M(b))*.  However, both a and b are executed as monitors.
   * When a is done then b must also be done – i.e. a finite number of iterations
   * of b must align with a.
   ************************************************************************/

  case class Iterate(name: String, a: Abstr.Monitor, b: Abstr.Monitor) extends Monitor {
    var c: ActorRef = _  // c is concrete counterpart to a
    var d: ActorRef = _  // d is concrete counterpart to b

    override def preStart() {
      log.debug("\nStarting " + this)
      c = Concr.startUp(a, context)
      d = Concr.startUp(b, context)
    }

    override def receive = {
      case Show(i) => tab(i)
                      println("ITERATE")
                      Await.result(ask(c, Show(indent(i))), timeout.duration)
                      Await.result(ask(d, Show(indent(i))), timeout.duration)
```

```
                        sender ! Tick

        case Step(u) =>
          val cf = ask(c, Step(u))  // copy new state to c
          val df = ask(d, Step(u))  // copy new state to d
          (Await.result(cf, timeout.duration).asInstanceOf[Reply],
           Await.result(df, timeout.duration).asInstanceOf[Reply])
          match {
            case (Done(s),
                  Done(_)) => sender ! Done(s)
                              context stop c
                              context stop d
                              context.become(this.zombie)

            case (Done(s),
                  More    ) => sender ! Fail
                              // error because a is the controlling monitor
                              log.warning(s"($name)ITERATE: LHS premature")
                              context stop c
                              context stop d
                              context.become(this.zombie)

            case (More,
                  Done(s)) => // send b round again...
                              context stop d
                              d = Concr.startUp(b, context)
                              val df = ask(d, Step(s))  // copy shared state to d
                              Await.result(df, timeout.duration).asInstanceOf[Reply]
                              match {
                                case Done(_) => // No further progress can be made
                                                // with b, but a hasn't finished, so
                                                sender ! Fail
                                                log.warning(s"($name)ITERATE: RHS empty loop")
                                                context stop c
                                                context stop d
                                                context.become(this.zombie)

                                case More    => sender ! More

                                case Fail    => sender ! Fail
                                                log.warning(s"($name)ITERATE: RHS failed")
                                                context stop c
                                                context stop d
                                                context.become(this.zombie)
                              }//match

            case (More,
                  More    ) => sender ! More

            case (r1,
                  r2      ) => sender ! Fail
                              (r1,r2) match {
                                case (Fail,Fail) => log.warning(s"($name)ITERATE: LHS & RHS failed")
                                case (Fail,_    ) => log.warning(s"($name)ITERATE: LHS failed")
                                case (_    ,Fail) => log.warning(s"($name)ITERATE: RHS failed")
                                case (_,    _
) => log.error(s"($name)ITERATE: unexpected ($r1,$r2)")
                              }
                              context stop c
                              context stop d
                              context.become(this.zombie)
          }//match

        case _ => sender ! Fail;
                  log.error("Unknown request - actor: " + this.toString)
                  context.become(this.rogue)
      }//receive
    }//Iterate

    /* ************************************************************************
     * Monitor class: a ITERATE b PROJ c
     * Performs a WITH (M(b))*.  However, both a and b are executed as monitors.
     * When a is done then b must also be done — i.e. a finite number of iterations
     * of b must align with a.
```

```
   ******************************************************************************/

  case class Project(name: String, a: Abstr.Monitor, b: Abstr.Monitor, p: Abstr.Monitor)
      extends Monitor {
    var c: ActorRef = _  // c is concrete counterpart to a
    var d: ActorRef = _  // d is concrete counterpart to b
    var q: ActorRef = _  // q is concrete counterpart to p

    override def preStart() {
      log.debug("\nStarting " + this)
      c = Concr.startUp(a, context)
      d = Concr.startUp(b, context)
      q = Concr.startUp(p, context)
    }

    override def receive = {
      case Show(i) => tab(i)
                      println("PROJECT")
                      Await.result(ask(c, Show(indent(i))), timeout.duration)
                      Await.result(ask(d, Show(indent(i))), timeout.duration)
                      Await.result(ask(q, Show(indent(i))), timeout.duration)
                      sender ! Tick

      case Step(u) =>
        val cf = ask(c, Step(u))  // copy new state to c
        val df = ask(d, Step(u))  // copy new state to d
        val qf = ask(q, Step(u))  // copy new state to d
        (Await.result(cf, timeout.duration).asInstanceOf[Reply],
         Await.result(df, timeout.duration).asInstanceOf[Reply],
         Await.result(qf, timeout.duration).asInstanceOf[Reply])
        match {
          case (Done(s),
                Done(_),
                Done(_)) => // All three monitors are satisfied by the first state
                            sender ! Done(s)
                            context stop c
                            context stop d
                            context stop q
                            context.become(this.zombie)
          case (More,
                More,
                More)    => // All three monitors need to continue
                            sender ! More
                            context.become(receive2)

          case (r1,
                r2,
                r3)      => sender ! Fail
                            (r1,r2,r3) match {
                              case (Fail,Fail,Fail) => log.warning(s"($name)PROJECT: 1st state all failed")
                              case (Fail,Fail,_
) => log.warning(s"($name)PROJECT: 1st state 1&2 failed")
                              case (Fail,_,
Fail) => log.warning(s"($name)PROJECT: 1st state 1&3 failed")
                              case (_,
Fail,Fail) => log.warning(s"($name)PROJECT: 1st state 2&3 failed")
                              case (Fail,_,    _
) => log.warning(s"($name)PROJECT: 1st state 1 failed")
                              case (_,    Fail,_
) => log.warning(s"($name)PROJECT: 1st state 2 failed")
                              case (_,    _,
Fail) => log.warning(s"($name)PROJECT: 1st state 3 failed")
                              case (_,    _,    _
) => log.error(s"($name)PROJECT: unexpected ($r1,$r2,$r3)")
                            }
                            context stop c
                            context stop d
                            context stop q
                            context.become(this.zombie)
        }//match

      case _ => sender ! Fail;
                log.error("Unknown request - actor: " + this.toString)
                context.become(this.rogue)
```

```
    }//receive

    def receive2: Receive = {
      case Show(i) => tab(i)
                      println("PROJECT")
                      Await.result(ask(c, Show(indent(i))), timeout.duration)
                      Await.result(ask(d, Show(indent(i))), timeout.duration)
                      Await.result(ask(q, Show(indent(i))), timeout.duration)
                      sender ! Tick

      case Step(u) =>
        val cf = ask(c, Step(u))  // copy new state to c
        val df = ask(d, Step(u))  // copy new state to d
        (Await.result(cf, timeout.duration).asInstanceOf[Reply],
         Await.result(df, timeout.duration).asInstanceOf[Reply])
        match {
          case (Done(s),
                Done(_)) => val qf = ask(q, Step(s)) // send s to projection
                            Await.result(qf, timeout.duration).asInstanceOf[Reply]
                            match {
                              case Done(_) => sender ! Done(s)
                              case More    => sender ! Fail
                                              log.warning(s"($name)PROJECT: 1&2 premature")
                              case Fail    => sender ! Fail
                                              log.warning(s"($name)PROJECT: 1 premature; 2 failed")
                            }//match
                            context stop c
                            context stop d
                            context stop q
                            context.become(this.zombie)

          case (Done(s),
                More   ) => sender ! Fail
                            log.warning(s"($name)PROJECT: 1 premature")
                            context stop c
                            context stop d
                            context stop q
                            context.become(this.zombie)

          case (More,
                Done(s)) => // send b round again...
                            context stop d
                            d = Concr.startUp(b, context)
                            val df = ask(d, Step(s))  // copy shared state to d
                            Await.result(df, timeout.duration).asInstanceOf[Reply]
                            match {
                              case Done(_) => // No further progress can be made
                                              // with b, but a hasn't finished, so
                                              sender ! Fail
                                              log.warning(s"($name)PROJECT: 2 empty loop")
                                              context stop c
                                              context stop d
                                              context stop q
                                              context.become(this.zombie)

                              case More    => // Send s to projection
                                              val qf = ask(q, Step(s))
                                              Await.result(qf, timeout.duration).asInstanceOf[Reply]
                                              match {
                                                case More    => sender ! More
                                                case Done(_) => sender ! Fail
                                                                log.warning(s"($name)PROJECT: 3 premature")
                                                                context stop c
                                                                context stop d
                                                                context stop q
                                                                context.become(this.zombie)
                                                case Fail    => sender ! Fail
                                                                log.warning(s"($name)PROJECT: 3 failed")
                                                                context stop c
                                                                context stop d
                                                                context stop q
                                                                context.become(this.zombie)
                                              }//match

```

```
                                case Fail     => sender ! Fail
                                                log.warning(s"($name)PROJECT: 2 failed")
                                                context stop c
                                                context stop d
                                                context stop q
                                                context.become(this.zombie)
                              }//match

            case (More,
                  More    ) => sender ! More

            case (r1, r2 ) => sender ! Fail
                              (r1,r2) match {
                                case (Fail,Fail) => log.warning(s"($name)PROJECT: 1&2 failed")
                                case (Fail,_    ) => log.warning(s"($name)PROJECT: 1 failed")
                                case (_,    Fail) => log.warning(s"($name)PROJECT: 2 failed")
                                case (_,    _
) => log.error(s"($name)PROJECT: unexpected ($r1,$r2)")
                              }//match
                              context stop c
                              context stop d
                              context stop q
                              context.become(this.zombie)
          }//match

        case _ => sender ! Fail;
                  log.error("Unknown request - actor: " + this.toString)
                  context.become(this.rogue)
      }//receive2
    }//Project

  }//Concr

  /* ****************************************************************************************
  * Object Runtime encapsulates the runtime monitoring definitions that are exported for public
  * use.  It imports and re-exports everything in Protocol._ which makes the error messages and
  * other related reporting objects visible.  The key class that this interface exports is RTM.
  *****************************************************************************************/

  object Runtime {
    import scala.collection.immutable
    import scala.collection.immutable.Map
    import Protocol._
//    val system = ActorSystem("MonitorSystem") // An Akka Actor system with a name
//    def stopAllMonitors = system.shutdown     // Shut down the Actor system when done

    /* **************************************************************************************
    * RTMActor.  This private actor implements the actual runtime monitor ... sending updated
    * states to, and receiving replies from, the concrete monitor tree. A public interface to
    * it is provided by the RTM class - below.
    ***************************************************************************************/

    private case class RTMActor(a: Abstr.Monitor) extends Actor with ActorLogging {
      var c: ActorRef = _   // c is concrete counterpart to a

      override def preStart() {
        log.debug("Running " + this + ": analysing abstract monitor")
        c = Concr.startUp(a, context)
      }

      override def postStop() {
        log.debug("Stopping " + this)
      }

      override def receive = {
        case rqst => sender ! ask(c, rqst)
      }
    }//RTMActor

    /* **************************************************************************************
    * RTM: The mutable monitored state takes responsibility for managing the internal actor
    * system associated with the abstract monitor.  The client simply has to define their
    * abstract specification, spec, and pass it to an instance of RTM. For example:
    *
```

```
 *    val spec: Abstract.Monitor = ...
 *    val mu = RTM(spec, "Simulation")
 *
 * Once the monitoring is complete the client should call:
 *    mu.stop
 *
 * External clients can use the MonitoredState ... for example:
 *    object I extends Var[Int] { override def toString = "I" }
 *    object J extends Var[Int] { override def toString = "J" }
 *    mu.set(I, i)
 *    mu.set(J, mu.get(I)+1)
 *    mu.verify
 *    mu.checkWhile { ... statements ... }
 *
 * Known issue: The state is not fully specified.  The variables are, of course, typed since
 * they extend Var[T], but there is not a way, currently, of declaring the names and types
 * of all the variables in the state.  The state is simply a collection of (Var[T],T))
 * forSome {type T} pairs. This uses existential types.
 ***************************************************************************************/
case class RTM(a: Abstr.Monitor, name: String, system: ActorSystem) {
  private val m: ActorRef = system.actorOf(Props(classOf[RTMActor], a), name)
  private var store: immutable.Map[Variable, Value] = new immutable.HashMap()
  private var state: Int                      = 0
  private var updates: List[VarUpdate]        = List()
  private var reply: Reply                    = Done(List())
  private var printEachCheckPoint: Boolean    = false
  private var logEachCheckPoint: Boolean      = false
  private var stopped: Boolean                = false
  private val lock: Object                    = new Object
  private var timer: Long                     = 0
  private val log                             = Logging.getLogger(system, this)
  private var exception: RTM.RTVException     = new RTM.RTVException(f"Failure $name")

  /* ************************************************************************************
   * Methods to manage the store/state
   ***************************************************************************************/

  def set[T](v: Var[T], a: T): RTM = lock.synchronized {
      store   = store + ((v -> Val(a)))
      updates = updates ::: List((v,a))
      this
    }//lock

  def get[T](v: Var[T]): T = lock.synchronized {
      store(v).asInstanceOf[Val[T]]
      match {
        case Val(a) => a
      }
    }//lock

  def getStore = lock.synchronized {
      store.toSeq.sortWith(_._1.toString < _._1.toString)
    }//lock

  def getUpdates = lock.synchronized {
      updates // return the latest updates that were applied
    }//lock

  /* ************************************************************************************
   * To stop a monitor
   ***************************************************************************************/

  def stop = lock.synchronized {
      if (stopped) {
         log.info("Stop: Monitor " + name + " has been stopped.")
      }
      else {
        system stop m
      }
      reply // always return the last reply
    }//lock

  /* ************************************************************************************
   * To print a monitor
```

```
*****************************************************************************/

def showStore = lock.synchronized {
    f"$state%4d " +
    getStore.foldRight(""){case ((i,v),s) => f"${i.toString} -> ${v.toString}  $s"}
  }//lock

override def toString = lock.synchronized {
    "RTM (" + name + ") " +
    ( if (hasFailed) "Failed" else if (hasStopped) "Done! " else "More  " ) + showStore
  }//lock

/* ***************************************************************************
* To show a single monitor's concrete state
*****************************************************************************/

def show: Unit = lock.synchronized {
    if (stopped) println("Show: Monitor " + name + " has been stopped.")
    else          Await.result(ask(m, Show(0)), timeout.duration)
  }//lock

/* ***************************************************************************
* To set/unset the checkpoint printing flag
*****************************************************************************/

def printOn:  RTM = lock.synchronized { printEachCheckPoint = true;  this }//lock
def printOff: RTM = lock.synchronized { printEachCheckPoint = false; this }//lock

/* ***************************************************************************
* To set/unset the checkpoint logging flag
*****************************************************************************/

def logOn:  RTM = lock.synchronized { logEachCheckPoint = true;  this }//lock
def logOff: RTM = lock.synchronized { logEachCheckPoint = false; this }//lock

/* ***************************************************************************
* To set the default exception handler
*****************************************************************************/

def setException(e: RTM.RTVException): RTM = lock.synchronized {
  exception = e
  this
}//lock

/* ***************************************************************************
* To run a verification
*****************************************************************************/

def verify: Reply = lock.synchronized {
    var rf: Future[Reply] = null
    if (stopped) {
       getReply
    }
    else {
      val t0: Long = java.lang.System.nanoTime()
      rf = Await.result(ask(m,Step(updates)),timeout.duration).asInstanceOf[Future[Reply]]
      updates = List() // re-set for next time
      reply = Await.result(rf, timeout.duration).asInstanceOf[Reply]
      val t1: Long = java.lang.System.nanoTime()
      timer = timer + (t1 - t0)
      if (reply.isDone || reply.isFail) {
        system stop m
        stopped = true
      }
      if (printEachCheckPoint) {
        println(f"(${timer.toDouble/1000000000}%6.3f sec): $this")
      }
      if (logEachCheckPoint) {
        log.debug(f"(${timer.toDouble/1000000000}%6.3f sec): $this")
      }
      state = state + 1
      reply
    }
  }//lock
```

```

    def ! = this.verify

    def !! : Reply = this.!!(this.exception)

    def !!(e: Exception): Reply = this.verify match {
      case Fail => throw e
      case r    => r
    }//match

    /* ***************************************************************************
    * To analyse replies
    ***************************************************************************/

    def getNbrOfStates = lock.synchronized { this.state }
    def getTimer       = lock.synchronized { this.timer }
    def getReply       = lock.synchronized { this.reply }
    def hasStopped     = lock.synchronized { this.reply.isDone }
    def hasFailed      = lock.synchronized { this.reply.isFail }

  }//RTM

  object RTM {
    class RTVException(msg: String) extends RuntimeException {
      override def toString() = f"RTVException $msg"
    }
  }//companion object RTM

  /* ****************************************************************************
  * RTMRef Runtime Monitor Reference - for use with RTMC
  *****************************************************************************/

  case class RTMRef(name: String) { override def toString = "RTMRef("+name+")" }

  /* ****************************************************************************
  * RTMC Runtime Monitor Cluster:     [M0, M1, M2, ...]
  *****************************************************************************/

  class RTMC(name: String, system: ActorSystem) {
    private var monitors: immutable.Map[RTMRef, ActorRef] = new immutable.HashMap()
    private var replies:  immutable.Map[RTMRef, Reply]    = new immutable.HashMap()
    private var stopped:  immutable.List[RTMRef]          = List()
    private var failed:   immutable.List[RTMRef]          = List()
    private var store:    immutable.Map[Variable, Value]  = new immutable.HashMap()
    private var state:    Int                             = 0
    private var updates:  List[VarUpdate]                 = List()
    private val lock:     Object                          = new Object  // for synchronization
    private var printEachCheckPoint: Boolean              = false
    private var logEachCheckPoint: Boolean                = false
    private var timer: Long                               = 0
    val log = Logging.getLogger(system, this)

    /* ***************************************************************************
    * Methods to manage the store/state
    ***************************************************************************/

    def set[T](v: Var[T], a: T): RTMC = lock.synchronized {
        store   = store + ((v -> Val(a)))
        updates = updates ::: List((v,a))
        this
      }//lock

    def get[T](v: Var[T]): T = lock.synchronized {
        store(v).asInstanceOf[Val[T]]
        match { case Val(a) => a }
      }//lock

    def getStore = lock.synchronized {
        store.toSeq.sortWith(_._1.toString < _._1.toString)
      }//lock

    def getUpdates = lock.synchronized {
        updates // return the latest updates that were applied
      }//lock
```

```
    /* ************************************************************************************
    * Methods to add/remove/stop monitors to/from the cluster
    * add:       sets up a new RTMActor and associates it with a reference. This pair is then
    *            added to the 'cluster'
    * remove:    stops the monitor identified by its reference and then removes all
    *            references
    * removeAll: removes all monitors from the cluster
    *************************************************************************************/

    def add(a: Abstr.Monitor, name: String): RTMRef = lock.synchronized {
        val mr = RTMRef(name)
        monitors = monitors + (mr -> system.actorOf(Props(classOf[RTMActor], a), name))
        mr
      }//lock

    // Completely remove a monitor from the cluster
    def remove(mr: RTMRef): RTMC = lock.synchronized {
        if (monitors contains mr)
          system.stop(monitors(mr))
        monitors = monitors - mr
        replies  = replies  - mr
        stopped  = stopped.filterNot(_ == mr)
        failed   = failed.filterNot(_ == mr)
        this
      }//lock

    // Remove all monitors from the cluster
    def removeAll: RTMC = lock.synchronized { monitors.keys.foreach (remove(_)); this }//lock

    /* ************************************************************************************
    * To print out the monitors in the cluster
    *************************************************************************************/

    def showStore = lock.synchronized {
        "<" + state + "> " +
        getStore.foldRight(""){case ((i,v),s) => i.toString + "->" + v.toString + " " + s}
      }//lock

    override def toString = lock.synchronized {
        def show(a: RTMRef, s: String): String = a.name + " " + s
        "RTMCluster (" + name +
                    ") {Live: "    + monitors.keys.foldRight("")(show(_,_)) +
                    "} {Stopped: " + stopped.foldRight("")(show(_,_)) +
                    "} {Failed: "  + failed.foldRight("")(show(_,_)) +
                    "} {Store: " + showStore +
                    "}"

      }//lock

    /* ************************************************************************************
    * To show a single monitor's concrete state
    *************************************************************************************/

    def show(mr: RTMRef): Unit = lock.synchronized {
        if (monitors contains mr) Await.result(ask(monitors(mr), Show(0)), timeout.duration)
      }//lock

    /* ************************************************************************************
    * To set/unset the checkpoint printing flag
    *************************************************************************************/

    def printOn:  RTMC = lock.synchronized { printEachCheckPoint = true;  this }//lock
    def printOff: RTMC = lock.synchronized { printEachCheckPoint = false; this }//lock

    /* ************************************************************************************
    * To set/unset the checkpoint logging flag
    *************************************************************************************/

    def logOn:  RTMC = lock.synchronized { logEachCheckPoint = true;  this }//lock
    def logOff: RTMC = lock.synchronized { logEachCheckPoint = false; this }//lock

    /* ************************************************************************************
    * To run a verification
```

```scala
    ***************************************************************************/

    def verify: Unit = lock.synchronized {
        var ns: immutable.Map[RTMRef,Future[Reply]] = new immutable.HashMap()

        val t0: Long = java.lang.System.nanoTime()
        monitors.foreach {
          case (mr: RTMRef, m: ActorRef) =>
                    ns = ns + ((mr, Await.result(ask(m, Step(updates)),
                                       timeout.duration).asInstanceOf[Future[Reply]]))
          }//foreach
        updates = List() // re-set for next time
        val ps = ns.mapValues{rf => Await.result(rf, timeout.duration).asInstanceOf[Reply]}
        val t1: Long = java.lang.System.nanoTime()
        timer = timer + (t1 - t0)
        val (more, no_more) = ps.partition       { case (mr, r) => r.isMore }
        val (done, fail)    = no_more.partition { case (mr, r) => r.isDone }
        stopped  = stopped  ++ (done.keys)
        failed   = failed   ++ (fail.keys)
        monitors = monitors -- (no_more.keys)
        replies  = replies  ++ ps
        if (printEachCheckPoint) {
            println(f"(${timer.toDouble/1000000000}%6.3f sec): $this")
        }
        if (logEachCheckPoint) {
          log.info(s"(${timer.toDouble/1000000000}%6.3f sec): $this")
        }
        state = state + 1
      }//lock

    /* ************************************************************************
    * To analyse replies
    ***************************************************************************/

    def getStoppedMonitors: List[RTMRef]          = lock.synchronized { stopped }
    def getFailedMonitors:  List[RTMRef]          = lock.synchronized { failed  }
    def getLiveMonitors:    List[RTMRef]          = lock.synchronized { monitors.keys.toList }
    def getReplies: immutable.Map[RTMRef,Reply] = lock.synchronized { replies }
    def getReply(mr: RTMRef): Option[Reply] =
      lock.synchronized { if (replies contains mr) Some(replies(mr)) else None }//lock
    def hasStopped(mr: RTMRef): Boolean = lock.synchronized { stopped  contains mr }
    def hasFailed(mr: RTMRef):  Boolean = lock.synchronized { failed   contains mr }
    def isLive(mr: RTMRef):     Boolean = lock.synchronized { monitors contains mr }
    def noneFailed: Boolean = lock.synchronized { replies.values.forall(r => !(r.isFail)) }
  }//RTMC
 }//Runtime object
}//Monitor object
```

Appendix B

Practical examples

B.1 Tennis example

The code listings in this section relate to the tennis example in Chapter 4.5. The three files comprise:

1. The definitions for the simulation including the monitored and non-monitored variables;
2. The ITL-Monitor specifications
3. The main simulation itself

Listing B.1: Tennis example: definitions

```
package demo.tennis

object Defs {
  import runtime.analysis.ITL._                                    // Needed for Var definitions

  /* ************************************************************************************
  * Data types used by the simulation and the specification
  ************************************************************************************/
  class Player {
    def other = if (this==P1) P2 else P1
  }
  case object P1 extends Player
  case object P2 extends Player
  class Score
  case object Love extends Score
  case object Fifteen extends Score
  case object Thirty extends Score
  case object Forty extends Score
  case object Advantage extends Score
  case object Game extends Score
  implicit object RelScore extends Eq[Score] with Ord[Score] {
    override def EQ(a: Score, b: Score): Boolean = a==b
    override def LE(a: Score, b: Score): Boolean = a match {
      case Love      => true
      case Fifteen   => b != Love
      case Thirty    => b != Love && b != Fifteen
      case Forty     => b == Forty || b == Advantage || b == Game
      case Advantage => b == Advantage || b == Game
```

```
      case Game       => b == Game
    }
  }

  /* ****************************************************************************
  * Monitored variables
  *****************************************************************************/
  case class Points(p: Player) extends Var[Score] { override def toString = "Points("+p+")" }
  case class Games(p: Player)  extends Var[Int]   { override def toString = "Games("+p+")"  }
  case class Sets(p: Player)   extends Var[Int]   { override def toString = "Sets("+p+")"   }
}
```

Listing B.2: Tennis example: specification

```
package demo.tennis

object Spec {
  import runtime.analysis.ITL._                              // ITL definitions and operators
  import runtime.analysis.Monitor.Abstr._                    // Runtime Monitor components
  import Defs._                                              // Variable definitions

  /* ****************************************************************************
  * ITL/monitor specification
  *****************************************************************************/

  def nextPoint(p: Player) = ((Points(p)`=`Love)      and (Next(Points(p))`=`Fifteen))   or
                             ((Points(p)`=`Fifteen)   and (Next(Points(p))`=`Thirty))    or
                             ((Points(p)`=`Thirty)    and (Next(Points(p))`=`Forty))     or
                             ((Points(p)`=`Forty)     and (Next(Points(p))`=`Game))      or
                             ((Points(p)`=`Forty)     and (Next(Points(p))`=`Advantage)) or
                             ((Points(p)`=`Advantage) and (Next(Points(p))`=`Forty))     or
                             ((Points(p)`=`Advantage) and (Next(Points(p))`=`Game))

  def winPoint  = skip and (((stable(Points(P1)) and nextPoint(P2))) or
                           ((stable(Points(P2)) and nextPoint(P1))) )

  def validGame  = label("VALID GAME",
                            (Points(P1)`=`Love) and (Points(P2)`=`Love) and
                            (winPoint).chopstar and
                            (((Games(P1) <~ Games(P1) + 1) and stable(Games(P2))) or
                             ((Games(P2) <~ Games(P2) + 1) and stable(Games(P1))))
                        )

  def gameOver  = label("GAME OVER", ((Points(P1))`=`Game) or ((Points(P2))`=`Game) )

  def validSet  = label("VALID SET",
                           ((Games(P1)`=`0) and (Games(P2)`=`0))  and
                           (((Sets(P1) <~ Sets(P1) + 1) and stable(Sets(P2))) or
                            ((Sets(P2) <~ Sets(P2) + 1) and stable(Sets(P1))))
                       )

  def setOver   = label("SET OVER", ((Games(P1)>=6) and (Games(P2)+1 < Games(P1))) or
                                    ((Games(P2)>=6) and (Games(P1)+1 < Games(P2))) )

  def matchOver = label("MATCH OVER", (Sets(P1)`=`3) or (Sets(P2)`=`3) )

  def startMatch = label("START MATCH", (Points(P1)  `=` Love) and (Points(P2)  `=` Love) and
                                        (Games(P1)   `=` 0)    and (Games(P2)   `=` 0   ) and
                                        (Sets(P1)    `=` 0)    and (Sets(P2)    `=` 0   )  )

/*****************************************************************************
 * Analysis granularity = one game
 *****************************************************************************/

  def bygame = GUARD(startMatch) THEN HALT(matchOver) ITERATE (
                  (SKIP THEN HALT(setOver) ITERATE (
                                 SKIP THEN HALT(gameOver) WITH (skip `;` validGame)
                              )
                  ) WITH (skip `;` validSet)
              )

/*****************************************************************************
```

```
 * Analysis granularity = one set
 ***************************************************************************************/

  def byset = GUARD(startMatch) THEN HALT(matchOver) ITERATE (
                                  (SKIP THEN HALT(setOver)) WITH
                                  ((skip ‘;‘ (halt(gameOver) and validGame)).chopstar and
                                   (skip ‘;‘ validSet))
                                )

/****************************************************************************************
 * Analysis granularity = one match - i.e. the whole interval checked once at the end
 ***************************************************************************************/

  def validMatch = (  (skip ‘;‘ (halt(gameOver) and validGame)).chopstar and
                      (skip ‘;‘ (halt(setOver)  and validSet))
                   ).chopstar

  def bymatch   = GUARD(startMatch) THEN HALT(matchOver) WITH validMatch

/****************************************************************************************
 * Analysis granularity = one game / adding projection
 ***************************************************************************************/

  def setsIncr(p: Player) = (keep((Next(Sets(p)) - Sets(p)) <= 1))

  def bygamep= GUARD(startMatch) THEN HALT(matchOver) ITERATE (
                   (SKIP THEN HALT(setOver) ITERATE (
                                  SKIP THEN HALT(gameOver) WITH (skip ‘;‘ validGame)
                                )
                   ) WITH (skip ‘;‘ validSet)
               ) PROJECT
                    ( SKIP THEN
                      HALT(matchOver) WITH (setsIncr(P1)) WITH (setsIncr(P2))
                    )
}
```

Listing B.3: Tennis example: simulation

```
package demo.tennis
/*
Example of 'Tennis Score' pattern
scalac demo/tennis/Simulation.scala

scala  demo.tennis.Simulation bygame
scala  demo.tennis.Simulation bygameproj
scala  demo.tennis.Simulation bysafegame
scala  demo.tennis.Simulation byset
scala  demo.tennis.Simulation bymatch
*/

object Simulation {
  import akka.actor.ActorSystem
  import runtime.analysis.Monitor.Runtime._
  import Defs._                          //  Variable definitions
  import Spec._                          //  ITL and Runtime Monitor specifcation

  /* **********************************************************************************
  * Simulation / Program to be monitored
  ***************************************************************************************/

  def playMatch(mu: RTM)
  {
    def matchOver(p: Player) = mu.get(Sets(p)) == 3
    def setOver(p: Player) = (mu.get(Games(p)) >= 6) &&
                             ((mu.get(Games(p.other))+1) < mu.get(Games(p)))
    def gameOver = (mu.get(Points(P1)) == Game) || (mu.get(Points(P2)) == Game)

    val r = scala.util.Random
    var winner: Player = P1 //P1 is a placeholder initial value only

    /* *******************************************************************************
    * Play a match
```

```
  ***********************************************************************************/

  mu.set(Points(P1), Love).set(Points(P2), Love)
    .set(Games(P1),  0)    .set(Games(P2),  0)
    .set(Sets(P1),   0)    .set(Sets(P2),   0)
    .verify
  do
  {
      /* ****************************************************************************
      * Play a set
      ******************************************************************************/
      //println("New Set")
      mu.set(Games(P1), 0)
        .set(Games(P2), 0)

      do
      {
          /* ************************************************************************
          * Play a game
          **************************************************************************/
          //println("New Game - init")
          mu.set(Points(P1),Love)
            .set(Points(P2),Love)
            .verify
          //println("New Game - start")
          do
          {
              winner = if (r.nextInt(2)==0) P1 else P2 // random: 0==P1 win, 1==P2 win
              mu.get(Points(winner))
              match
              {
                case Love      => mu.set(Points(winner), Fifteen).verify
                case Fifteen   => mu.set(Points(winner), Thirty).verify
                case Thirty    => mu.set(Points(winner), Forty).verify          // the correct line
                //case Thirty      => mu.set(Points(winner), Game).verify      // insert a bug
                case Forty     => if (mu.get(Points(winner.other)) == Forty)
                                     mu.set(Points(winner), Advantage).verify
                                  else if (mu.get(Points(winner.other)) == Advantage)
                                     mu.set(Points(winner.other), Forty).verify
                                  else
                                     mu.set(Points(winner), Game)
                case Advantage => mu.set(Points(winner), Game)
              }
              //println("Winner: " + winner + ", (P1,P2) = " +
              //                                    (mu.get(Points(P1)), mu.get(Points(P2))))
          }
          while (!gameOver) /* ***************************************************/

          mu.set(Games(winner), mu.get(Games(winner)) + 1)
          if (setOver(winner))
            mu.set(Sets(winner), mu.get(Sets(winner)) + 1)
          mu.verify

      }
      while (!setOver(winner)) /* ****************************************************/

  }
  while (!matchOver(winner)) /* ******************************************************/

  println("Match over. Winner is " + winner)
}

/* ***********************************************************************************
* Simulation thread - starting, and then awaiting, the simulation and run-time monitor
*************************************************************************************/
def runSimulation(args: Array[String])
{
  val system = ActorSystem("Ex3ActorSystem")
  val mu = RTM(args(0) match
               {
                 case "bygame"  => bygame
                 case "bygamep" => bygamep
                 case "byset"   => byset
                 case "bymatch" => bymatch
```

```
                  },
                  "Tennis",
                  system).printOn

    playMatch(mu)
    mu.stop
    Thread sleep 2000
    system.terminate
  }

  def main(args: Array[String]) {
    runSimulation(args)
  }

}
```

B.2 Latch example

The Scala code for the latch example is separated into two objects. One is `TC` into which the TraceContract specifications have been placed. The second is `Simulation` which contains the ITL-Monitor specification and the program that generates sample execution traces for analysis. The latter distinguishes between the the monitored and non-monitored variables, both of which are used within the simulation irrespective of whether or not any monitoring is carried out. The integration of monitored variables into the program under test performs the instrumentation used by ITL-Monitor.

Listing B.4: TraceContract definitions for the latch example

```
package demo.latch

object TC {
  import tracecontract._

  /*
   * An event is the construction of a new state consisting of the three
   * flags: a, b, and s.  A trace is a sequence of events (states)
   */
  case class Event(a: Boolean, b: Boolean, s: Boolean)

  def aHi: PartialFunction[Event,Boolean] = { case Event(true, _,_) => true }
  def aLo: PartialFunction[Event,Boolean] = { case Event(false,_,_) => true }
  def bHi: PartialFunction[Event,Boolean] = { case Event(_,true, _) => true }
  def bLo: PartialFunction[Event,Boolean] = { case Event(_,false,_) => true }
  def sHi: PartialFunction[Event,Boolean] = { case Event(_,_,true ) => true }
  def sLo: PartialFunction[Event,Boolean] = { case Event(_,_,false) => true }

  class R1 extends Monitor[Event] {
    /*
     * If B is stable across two adjacent states then S is low in the 2nd state
     *
     * □((B ⇔ ⓦ (B)) ⇒ ⓦ (¬ S))
     */

    def bStable   = ((matches{bHi}) and weaknext(matches{bHi})) or
                    ((matches{bLo}) and weaknext(matches{bLo}))

    property('R1) {
      globally {
        bStable implies (weaknext(matches{sLo}))
      }
    }
  }//R1
```

```
class R2 extends Monitor[Event] {
  /*
   * If B is unstable across two adjacent states then S is high in the 2nd state
   *
   * □(¬ (B ⇔ ⓦ (B)) ⇒ ⓦ (S))
   */

  def bStable   = ((matches{bHi}) and weaknext(matches{bHi})) or
                  ((matches{bLo}) and weaknext(matches{bLo}))

  def bUnstable = ((matches{bHi}) and weaknext(matches{bLo})) or
                  ((matches{bLo}) and weaknext(matches{bHi}))

  property('R2) {
    globally {
     bUnstable implies (weaknext(matches{sHi}))
    }
  }
}//R2

class R3_R4 extends Monitor[Event] {
  /*
   * R3  Whenever A is stable across two adjacent states then B is stable
   *
   *      □((¬ A ∧ ⓦ (¬ A)) ⇒ (B ⇔ ⓦ (B))
   *
   * R4  Whenever A is low across two adjacent states then B is stable
   *
   *      □((¬ A ∧ ⓦ (A)) ⇒ (B ⇔ ⓦ (B)))
   */

  def bStable   = ((matches{bHi}) and weaknext(matches{bHi})) or
                  ((matches{bLo}) and weaknext(matches{bLo}))

  def aStableLo = (matches{aLo}) and weaknext(matches{aLo})

  def aRises    = (matches{aLo}) and weaknext(matches{aHi})

  property('R3_R4) {
    globally { (aStableLo implies bStable) and (aRises implies bStable) }
  }
}//R3_R4

class R5 extends Monitor[Event] {
  /*
   *  A state machine representing the latch behaviour
   *  Event                      State  ABS   ==>  valid moves:
   *  Event(false,false,false)   S0     ___   ==>  S0, S4
   *  Event(false,false,true )   S1     __-   ==>  S0, S4
   *  Event(false,true ,false)   S2     _-_   ==>  S2, S6
   *  Event(false,true ,true )   S3     _--   ==>  S2, S6
   *  Event(true ,false,false)   S4     -__   ==>  S0, S3, S4, S7
   *  Event(true ,false,true )   S5     -_-   ==>  S0, S3, S4, S7
   *  Event(true ,true ,false)   S6     --_   ==>  S1, S2, S5, S6
   *  Event(true ,true ,true )   S7     ---   ==>  S1, S2, S5, S6
   */

  property('R5) { S0 }

  def S0: Formula = state {
                      case Event(true ,false,false) => S4
                      case Event(false,false,false) => S0
                      case _ => error
                    }

  def S2: Formula = state {
                      case Event(true ,true ,false) => S6
                      case Event(false,true ,false) => S2
                      case _ => error
                    }

  def S4: Formula = state {
                      case Event(false,false,false) => S0
```

```
                          case Event(false,true ,true ) => S3
                          case Event(true ,true ,true ) => S7
                          case Event(true ,false,false) => S4
                          case _ => error
                        }

    def S6: Formula = state {
                          case Event(false,true ,false) => S2
                          case Event(false,false,true ) => S1
                          case Event(true ,false,true ) => S5
                          case Event(true ,true ,false) => S6
                          case _ => error
                        }

    def S3: Formula = state {
                          case Event(false,true ,false) => S2
                          case Event(true ,true ,false) => S6
                          case _ => error
                        }

    def S1: Formula = state {
                          case Event(false,false,false) => S0
                          case Event(true ,false,false) => S4
                          case _ => error
                        }

    def S7: Formula = state {
                          case Event(true ,true ,false) => S6
                          case Event(true ,false,true ) => S5
                          case Event(false,true ,false) => S2
                          case Event(false,false,true ) => S1
                          case _ => error
                        }

    def S5: Formula = state {
                          case Event(true ,false,false) => S4
                          case Event(true ,true ,true ) => S7
                          case Event(false,false,false) => S0
                          case Event(false,true ,true ) => S3
                          case _ => error
                        }
  }//R5

  class LTLRequirements extends Monitor[Event] {
    /*
     * All the LTL requirements are conjoined in the following monitor
     */

    monitor( new R1, new R2, new R3_R4 )
  }

  class StMRequirements extends Monitor[Event] {
    /*
     * The state machine requirement becomes a monitor
     */
    monitor( new R5 )
  }

  class AllRequirements extends Monitor[Event] {
    /*
     * A monitor representing the conjunction of the LTL and state machine
     */
    monitor( new LTLRequirements, new StMRequirements )
  }

  /*
   * Convenient covers for exporting each of the combinations
   */
  def monitorLTL = new LTLRequirements
  def monitorStM = new StMRequirements
  def monitorAll = new AllRequirements
  def monitorNil = new Monitor[Event]

}//TC
```

Listing B.5: Latch example simulation

```scala
package demo.latch

/*
 * Simulation of the latch example in which runtime verification may use any
 * combination of ITM(ITL), TraceContract(LTL), and TraceContract(state machine).
 */
object Simulation {
  import akka.actor.ActorSystem
  import runtime.analysis.ITL._
  import runtime.analysis.Monitor.Runtime._
  import runtime.analysis.Monitor.Abstr._

  var as: ActorSystem = _

  /*
   * ITM-monitored variables are integral to the simulation irrespective
   * of whether or not ITM monitoring is performed.
   */
  object S    extends Var[Boolean] { override def toString = "S" }
  object A    extends Var[Boolean] { override def toString = "A" }
  object B    extends Var[Boolean] { override def toString = "B" }
  object STOP extends Var[Boolean] { override def toString = "STOP" }

  /*
   * The ITM (ITL) specification:
   *
   * initial: The initial state condition in which all the flags are low.
   *
   * clause2: Satisfied by a subinterval from this point up to the first
   * state in which B changes value. Throughout this interval S can be
   * high or low in the first state; then S must stay low until the final
   * state when S must be high. (In an extreme case it is possible for this
   * subinterval to consist of only two states in which (B != O(B)) & O(S)
   * holds.
   *
   * clause3: Satisfied by a subinterval from this point up to the first
   * state in which A is raised followed by the first state in which A is
   * lowered. Within the first part of this subinterval B must remain
   * stable.
   *
   * spec: The initial state must be fused with an interval that continues
   * until the first state in which HALT holds. Over this interval the
   * cycles represented by clause2 and clause3 are repeated.
   */
  val initial = (~A and ~B and ~S)

  val clause2 = FIRST(B <~ ~B) WITH (skip `;` halt(S))

  val clause3 = (HALT(A) WITH (stable(B))) THEN (HALT(~A))

  val spec    = (GUARD(initial)
                 THEN (HALT(STOP)
                       ITERATE clause2 ITERATE clause3))

  /*
   * The purpose of the simulation is to demonstrate and compare the different
   * runtime verification approaches. Flags to the simulation control which of
   * these is set/unset. The length of the simulation (the number of verified
   * states) is returned.
   */
  def runSimulation(iter:    Int,      // Iteration number (for multiple runs)
                    aCycles: Int,      // Number of A cycles to simulate
                    runITM:  Boolean,  // ITM monitoring on/off
                    runLTL:  Boolean,  // LTL monitoring on/off
                    runStM:  Boolean,  // State Machine monitoring on/off
                    runAna:  Boolean,  // AnaTempura monitoring on/off
                    printOn: Boolean,  // Stdout continuous commentary on/off
                    errorOn: Int       // Error on given cycle (0 = off)
                   ): Int = {
    /*
     * A number of constants control the simulation:
     * rand:       A random number generator
     * aStayLow:   Generates a random number of states (1-20) for A to stay low
```

```
   * bFlips:      Randomly determines if B flips state (50%)
   * aIsLowered: Randomly determines if A is lowered (5%)
   */
  val rand = scala.util.Random
  def aStaysLow = 1+rand.nextInt(20)
  def bFlips = rand.nextInt(100)<50
  def aIsLowered = rand.nextInt(100)<5

  /*
   * mu is the ITM monitor.
   *    -- associated with an Akka actor system and an ITL specification
   *
   * nu is the TraceContract monitor.
   *    -- runs LTL and/or state machine monitor combinations as required
   */
  val mu = RTM(spec, "Latch"+iter, as)
  val nu = if (runLTL && runStM) TC.monitorAll
           else if (runLTL)       TC.monitorLTL
           else if (runStM)       TC.monitorStM
           else                   TC.monitorNil

  // Initialise logging and printing
  if (printOn) { mu.printOn; nu.setSuccess(true) }
  nu.setEventLog("log/Latch.log")

  /*
   * A counter to measure the length of a simulation run
   */
  var numOfStates = 0

  /*
   * verify() is invoked at each assertion point within the simulation.
   * This performs the instrumentation connecting the program to the
   * monitors.
   *
   * The monitored variables are maintainted within the monitor (mu)
   * irrespective of whether or not ITM verification is invoked. All of
   * the monitoring systems used by the simulation use the same values
   * taken from these state variables. This facilitates a fair comparison
   * of the different monitoring systems to be made.
   *
   * The AnaTempura instrumentation is handled via an output on stdout.
   *
   * The TraceContract instrumentation requires the combination of the
   * monitored variables into a TC.Event. This event is transmitted to
   * the TC monitor (nu).
   *
   * The ITM variables are maintainted within the monitor (mu) itself. The
   * instruction mu.!! instructs the monitor to process the current state.
   * Any violation will result in an exception being thrown.
   */
  def verify() {
    if (runAna)
      println("!PROG: assert Event:"+
              mu.get(A)+":"+mu.get(B)+":"+mu.get(S)+":"+mu.get(STOP)+":!")
    if (runLTL || runStM) nu.verify(TC.Event(mu.get(A), mu.get(B), mu.get(S)))
    if (runITM) mu.!!
    numOfStates = numOfStates + 1
  }

  /*
   * Initialise the monitored state variables.
   * This is not an assertion point.
   */
  mu.set(STOP,false).set(A,false).set(B,false).set(S,false)

  /*
   * The ITM monitor mu raises an exception if violation is encountered.
   * This represents a react-at-runtime behaviour. The alternative
   * behaviour is placed within the corresponding catch block.
   */
  try {
    for (i <- 1 to aCycles) {                // Simulate this many A cycles

```

```
        verify()

        if (mu.get(S))                          // A has been lowered (at start
          mu.set(S,false)                       // of cycle) - ensure S is low

        for (j <- 1 to aStaysLow)               // Generate states for analysis
          verify()                              // while A remains low

        mu.set(A,true)                          // A is now raised

        while (mu.get(A)) {                     // While A is raised...

          verify()                              // Assertion point

          if (bFlips)                           // Randomly, B may flip
            mu.set(B,!mu.get(B)).set(S,true)    // If so,  raise S
          else
            mu.set(S,false)                     // If not, lower S

          if (i==errorOn)                       // The simulation allows for a
            mu.set(B,!mu.get(B))                // deliberate error to occur
                                                // at the i-th iteration

          if (aIsLowered)                       // Randomly, A may be lowered
            mu.set(A,false)

        }//while
      }//for-i

      if (!mu.get(S))                           // Ensure that the simulation
        mu.set(B,!mu.get(B)).set(S,true)        // ends with a B-transition

      mu.set(STOP,true)                         // Set STOP in the final state
      verify()

    } catch {
      case e: RTM.RTVException =>
        println(e)                              // ITM detected a violation
        println("React at Runtime...")          // Alternative action goes here

    } finally {
      println(mu.getReply)                      // Print final ITM judgement.
      nu.end()                                  // nu prints final summary by
      mu.stop                                   // default. Close both monitors
    }
    numOfStates                                 // Return the sumulation length
  }//runSimulation

  /*
   * The main program analyses the command line arguments to determine
   * how to call run the simulation. If the simulation type (args(1))
   * contains 'a' then the simulation will be repeated ten times and an
   * average timing analysis printed. Otherwise the simulation runs once.
   * AnaTempura is only run once so the inclusion of flag 't' suppresses
   * flag 'a'.
   */
  def main(args: Array[String]) {
     // args(0)  Number of A cycles to run the simulation
     // args(1)  A string i=ITM l=LTL s=StM t=AnaTempura (a=run averages)
     // args(2)  A string: "on" means printing is on (anything else is "off")
     // args(3)  A number indicating which A-cycle to introduce an error
    val aCycles:  Int      =  args(0).toInt
    val runITM:   Boolean  =  (args.length > 1) && args(1).contains('i')
    val runLTL:   Boolean  =  (args.length > 1) && args(1).contains('l')
    val runStM:   Boolean  =  (args.length > 1) && args(1).contains('s')
    val runAna:   Boolean  =  (args.length > 1) && args(1).contains('t')
    val average:  Boolean  =  !runAna &&
                              (args.length > 1) && args(1).contains('a')
    val printOn:  Boolean  =  (args.length > 2) && args(2)=="on"
    val errorOn:  Int      =  if (args.length > 3) args(3).toInt else 0

    as = ActorSystem("LatchActorSystem")

    if (average) {
```

```
        var times: List[Long] = List()
        var states: List[Int] = List()
        val N = 10 // Run N experiments
        (1 to N) foreach { i =>
          val t0: Long = java.lang.System.nanoTime()
          val s = runSimulation(i, aCycles,
                                runITM, runLTL, runStM, runAna, printOn, errorOn)
          val t1: Long = java.lang.System.nanoTime()
          times = (t1 - t0) :: times
          states = s :: states
        }
        val min: Double = times.min.toDouble/1000000000
        val max: Double = times.max.toDouble/1000000000
        val avg: Double = times.sum.toDouble/N/1000000000
        val avs: Double = states.sum.toDouble/N
        print(f"A-cycles: $aCycles, sim: ${args(1)}, ")
        println(f"Times: min: $min%6.3f, max: $max%6.3f, ")
        println(f"avg: $avg%6.3f, avg: $avg%6.3f, avs: ${Math.round(avs)}")
      }
      else // run once
        runSimulation(1, aCycles, runITM, runLTL, runStM, runAna, printOn, errorOn)

      as.terminate // Close down the actor system
    }
}
```

B.2.1 Latch example - derived formula

In Section 6.2.1 the following ITL formula was presented.

$$
\begin{array}{lr}
(\mathsf{empty} \wedge \neg\, A \wedge \neg\, B \wedge \neg\, S)\ ; & \text{Initial state} \\
(\ \ \mathsf{halt}(STOP) & \text{Termination condition} \\
\quad \wedge\ (\ (B \mathbin{<\!\sim} \neg\, B) \wedge (\mathsf{skip}\ ;\ \mathsf{halt}(S))\ )^{*} & (1)^{*} \\
\quad \wedge\ (\ (\mathsf{halt}(A) \wedge \mathsf{stable}(B))\ ;\ (\mathsf{halt}(\neg\, A))\ )^{*} & (2)^{*} \\
) &
\end{array}
$$

From this specification, four requirements and one further derivation were calculated. The analysis is presented below.

Firstly consider formula (1). $B \mathbin{<\!\sim} \neg\, B$ is equivalent to $\mathsf{keep}(\bigcirc(B) \equiv B)\ ;\ (\mathsf{skip} \wedge (\bigcirc(B) \not\equiv B))$ and $\mathsf{skip}\ ;\ \mathsf{halt}(S)$ is equivalent to $\mathsf{keep}(\bigcirc(\neg\, S))\ ;\ (\mathsf{skip} \wedge \bigcirc(S))$. Thus:

$$
\begin{array}{lr}
(B \mathbin{<\!\sim} \neg\, B) \wedge (\mathsf{skip}\ ;\ \mathsf{halt}(S)) & \text{from } (1) \\
\equiv (\ \ \mathsf{keep}(\bigcirc(B) \equiv B)\ ;\ (\mathsf{skip} \wedge (\bigcirc(B) \not\equiv B))\ ) \wedge (\ \ \mathsf{keep}(\bigcirc(\neg\, S))\ ;\ (\mathsf{skip} \wedge \bigcirc(S))\ ) & \\
\equiv (\ \ \mathsf{keep}((\bigcirc(B) \equiv B) \wedge \bigcirc(\neg\, S))\ )\ ;\ (\ \ \mathsf{skip} \wedge (\bigcirc(B) \not\equiv B) \wedge \bigcirc(S)\ ) & \\
 & RFixedAndDistr^{(C.152)} \\
\supset (\ \ \mathsf{keep}((\bigcirc(B) \equiv B) \equiv \bigcirc(\neg\, S))\ )\ ;\ (\ \ \mathsf{skip} \wedge (\bigcirc(B) \not\equiv B) \equiv \bigcirc(S)\ ) & \text{logic} \\
\equiv (\ \ \mathsf{keep}((\bigcirc(B) \equiv B) \not\equiv \bigcirc(S))\ )\ ;\ (\ \ \mathsf{skip} \wedge (\bigcirc(B) \not\equiv B) \equiv \bigcirc(S)\ ) & \text{logic} \\
\equiv \mathsf{keep}(\ \ (\bigcirc(B) \not\equiv B) \equiv \bigcirc(S)\ ) & \text{logic, ITL } (\mathsf{keep})\quad (3)
\end{array}
$$

Secondly, consider formula (2).

$(\mathsf{halt}(A) \wedge \mathsf{stable}(B))\ ;\ \mathsf{halt}(\neg\, A)$ from (2)

$\equiv \mathsf{keep}((\neg\, A \wedge \bigcirc\neg\, A) \wedge (B = \bigcirc B))\ ;\ (\mathsf{skip} \wedge \neg\, A \wedge \bigcirc(A) \wedge \bigcirc(B))\ ;\ \mathsf{keep}(A)$ ITL (4)

Then

$(4) \supset \mathsf{keep}((\neg\, A \wedge \bigcirc(\neg\, A) \supset (B = \bigcirc(B)))$ (5)

and

$(4) \supset \mathsf{keep}((\neg\, A \wedge \bigcirc(A) \supset (B = \bigcirc B))$ (6)

Finally, the four requirements can be presented in ITL.

$R1:$	$\mathsf{keep}(\ (\bigcirc(B) \equiv B) \supset \bigcirc(\neg\, S)\ )$	from (3)
$R2:$	$\mathsf{keep}(\ (\bigcirc(B) \not\equiv B) \supset \bigcirc(S)\ )$	from (3), contrapositive
$R3:$	$\mathsf{keep}((\neg\, A \wedge \bigcirc(\neg\, A) \supset (B = \bigcirc(B)))$	from (5)
$R4:$	$\mathsf{keep}((\neg\, A \wedge \bigcirc(A) \supset (B = \bigcirc B))$	from (6)
$D5:$	$\mathsf{keep}((\neg\, A \wedge \bigcirc(\neg\, A) \supset \bigcirc(\neg\, S))$	from R3 and R1, transitivity

B.3 Checkout example - experiments

The results of running each of the checkout experiments (6.3.3) are listed below. The runs generate a large volume of output, so only the concluding lines, containing the statistical data, are included for each run.

B.3.1 Experiment 1

```
Experiment 1 R1-R5, ITM only, one monitor, 100 intervals

runMain demo.marvin.Simulation 100 1 10
******** LOG FST STATES: AVG(80), TOT(40344), MIN(1), MAX(131)
=======> Monitor T_1 completes with success. 8009 states, 100 custs, in   6.376 sec.
=======> Customers[100], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 100 1 10
******** LOG FST STATES: AVG(76), TOT(38554), MIN(1), MAX(139)
=======> Monitor T_1 completes with success. 7671 states, 100 custs, in   5.189 sec.
=======> Customers[100], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 100 1 10
******** LOG FST STATES: AVG(79), TOT(40195), MIN(1), MAX(136)
=======> Monitor T_1 completes with success. 7938 states, 100 custs, in   5.064 sec.
```

```
=======> Customers[100], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 100 1 10
******** LOG FST STATES: AVG(80), TOT(40606), MIN(1), MAX(121)
=======> Monitor T_1 completes with success. 8033 states, 100 custs, in   5.751 sec.
=======> Customers[100], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 100 1 10
******** LOG FST STATES: AVG(82), TOT(41598), MIN(1), MAX(146)
=======> Monitor T_1 completes with success. 8251 states, 100 custs, in   6.378 sec.
=======> Customers[100], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 100 5 10
******** LOG FST STATES: AVG(206), TOT(104911), MIN(1), MAX(479)
=======> Monitor T_1 completes with success. 20963 states, 100 custs, in  36.151 sec.
=======> Customers[100], Items[5 x 10 = 50], Terminals(1)

runMain demo.marvin.Simulation 100 5 10
******** LOG FST STATES: AVG(233), TOT(118185), MIN(1), MAX(485)
=======> Monitor T_1 completes with success. 23655 states, 100 custs, in  20.313 sec.
=======> Customers[100], Items[5 x 10 = 50], Terminals(1)

runMain demo.marvin.Simulation 100 5 10
******** LOG FST STATES: AVG(229), TOT(116130), MIN(1), MAX(468)
=======> Monitor T_1 completes with success. 23135 states, 100 custs, in  28.982 sec.
=======> Customers[100], Items[5 x 10 = 50], Terminals(1)

runMain demo.marvin.Simulation 100 5 10
******** LOG FST STATES: AVG(233), TOT(117976), MIN(1), MAX(474)
=======> Monitor T_1 completes with success. 23565 states, 100 custs, in  18.660 sec.
=======> Customers[100], Items[5 x 10 = 50], Terminals(1)

runMain demo.marvin.Simulation 100 5 10
******** LOG FST STATES: AVG(211), TOT(106984), MIN(1), MAX(476)
=======> Monitor T_1 completes with success. 21398 states, 100 custs, in  20.352 sec.
=======> Customers[100], Items[5 x 10 = 50], Terminals(1)

runMain demo.marvin.Simulation 100 10 10
******** LOG FST STATES: AVG(321), TOT(163026), MIN(1), MAX(900)
=======> Monitor T_1 completes with success. 32651 states, 100 custs, in  26.267 sec.
=======> Customers[100], Items[10 x 10 = 100], Terminals(1)

runMain demo.marvin.Simulation 100 10 10
```

```
******** LOG FST STATES: AVG(261), TOT(132332), MIN(1), MAX(879)
=======> Monitor T_1 completes with success. 26427 states, 100 custs, in  19.728 sec.
=======> Customers[100], Items[10 x 10 = 100], Terminals(1)

runMain demo.marvin.Simulation 100 10 10
******** LOG FST STATES: AVG(268), TOT(135805), MIN(1), MAX(854)
=======> Monitor T_1 completes with success. 27148 states, 100 custs, in  19.856 sec.
=======> Customers[100], Items[10 x 10 = 100], Terminals(1)

runMain demo.marvin.Simulation 100 10 10
******** LOG FST STATES: AVG(312), TOT(158715), MIN(1), MAX(892)
=======> Monitor T_1 completes with success. 31672 states, 100 custs, in  25.150 sec.
=======> Customers[100], Items[10 x 10 = 100], Terminals(1)

runMain demo.marvin.Simulation 100 10 10
******** LOG FST STATES: AVG(285), TOT(144695), MIN(1), MAX(889)
=======> Monitor T_1 completes with success. 28906 states, 100 custs, in  23.289 sec.
=======> Customers[100], Items[10 x 10 = 100], Terminals(1)

runMain demo.marvin.Simulation 100 20 10
******** LOG FST STATES: AVG(327), TOT(166220), MIN(1), MAX(1655)
=======> Monitor T_1 completes with success. 33155 states, 100 custs, in  24.453 sec.
=======> Customers[100], Items[20 x 10 = 200], Terminals(1)

runMain demo.marvin.Simulation 100 20 10
******** LOG FST STATES: AVG(314), TOT(159699), MIN(1), MAX(1673)
=======> Monitor T_1 completes with success. 31916 states, 100 custs, in  28.308 sec.
=======> Customers[100], Items[20 x 10 = 200], Terminals(1)

runMain demo.marvin.Simulation 100 20 10
******** LOG FST STATES: AVG(339), TOT(172323), MIN(1), MAX(1492)
=======> Monitor T_1 completes with success. 34595 states, 100 custs, in  29.392 sec.
=======> Customers[100], Items[20 x 10 = 200], Terminals(1)

runMain demo.marvin.Simulation 100 20 10
******** LOG FST STATES: AVG(323), TOT(164205), MIN(1), MAX(1553)
=======> Monitor T_1 completes with success. 32798 states, 100 custs, in  27.699 sec.
=======> Customers[100], Items[20 x 10 = 200], Terminals(1)

runMain demo.marvin.Simulation 100 20 10
******** LOG FST STATES: AVG(342), TOT(173893), MIN(1), MAX(1644)
=======> Monitor T_1 completes with success. 34772 states, 100 custs, in  31.943 sec.
=======> Customers[100], Items[20 x 10 = 200], Terminals(1)
```

B.3.2 Experiment 2

```
Experiment 2 true, ITM only, one monitor, 100 intervals

runMain demo.marvin.Simulation 100 1 10
ByCust(always(always(eventually(empty)))) // n^3
******** LOG FST STATES: AVG(77), TOT(7896), MIN(1), MAX(121)
=======> Monitor T_1 completes with success. 7795 states, 100 custs, in   2.298 sec.
=======> Customers[100], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 100 4 10
ByCust(always(always(eventually(empty)))) // n^3
******** LOG FST STATES: AVG(208), TOT(21266), MIN(1), MAX(383)
=======> Monitor T_1 completes with success. 21165 states, 100 custs, in   6.118 sec.
=======> Customers[100], Items[4 x 10 = 40], Terminals(1)

runMain demo.marvin.Simulation 100 7 10
ByCust(always(always(eventually(empty)))) // n^3
******** LOG FST STATES: AVG(243), TOT(24804), MIN(1), MAX(628)
=======> Monitor T_1 completes with success. 24703 states, 100 custs, in   8.565 sec.
=======> Customers[100], Items[7 x 10 = 70], Terminals(1)

runMain demo.marvin.Simulation 100 10 10
ByCust(always(always(eventually(empty)))) // n^3
******** LOG FST STATES: AVG(286), TOT(29139), MIN(1), MAX(872)
=======> Monitor T_1 completes with success. 29038 states, 100 custs, in   9.795 sec.
=======> Customers[100], Items[10 x 10 = 100], Terminals(1)

runMain demo.marvin.Simulation 100 100 10
******** LOG FST STATES: AVG(373), TOT(38014), MIN(1), MAX(1758)
=======> Monitor T_1 completes with success. 37913 states, 100 custs, in  12.844 sec.
=======> Customers[100], Items[100 x 10 = 1000], Terminals(1)

runMain demo.marvin.Simulation 100 1 10
ByCust(always(always(always(eventually(empty))))) // n^4
******** LOG FST STATES: AVG(84), TOT(8604), MIN(1), MAX(132)
=======> Monitor T_1 completes with success. 8503 states, 100 custs, in   7.051 sec.
=======> Customers[100], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 100 4 10
ByCust(always(always(always(eventually(empty))))) // n^4
******** LOG FST STATES: AVG(203), TOT(20729), MIN(1), MAX(379)
=======> Monitor T_1 completes with success. 20628 states, 100 custs, in  78.349 sec.
=======> Customers[100], Items[4 x 10 = 40], Terminals(1)
```

```
runMain demo.marvin.Simulation 100 7 10
ByCust(always(always(always(eventually(empty))))) // n^4
******** LOG FST STATES: AVG(237), TOT(24126), MIN(1), MAX(610)
=======> Monitor T_1 completes with success. 24025 states, 100 custs, in 200.102 sec.
=======> Customers[100], Items[7 x 10 = 70], Terminals(1)

runMain demo.marvin.Simulation 100 10 10
ByCust(always(always(always(eventually(empty))))) // n^4
******** LOG FST STATES: AVG(280), TOT(28562), MIN(1), MAX(824)
=======> Monitor T_1 completes with success. 28461 states, 100 custs, in 367.922 sec.
=======> Customers[100], Items[10 x 10 = 100], Terminals(1)
```

B.3.3 Experiment 3

```
Experiment 3 R1-R5, ITM only, one monitor, 1000 intervals
runMain demo.marvin.Simulation 1000 1 10
******** LOG FST STATES: AVG(81), TOT(403722), MIN(1), MAX(162)
=======> Monitor T_1 completes with success. 79949 states, 1000 custs, in  41.575 sec.
=======> Customers[1000], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 2000 1 10
******** LOG FST STATES: AVG(81), TOT(807671), MIN(1), MAX(157)
=======> Monitor T_1 completes with success. 159838 states, 2000 custs, in  80.466 sec.
=======> Customers[2000], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 3000 1 10
******** LOG FST STATES: AVG(80), TOT(1205241), MIN(1), MAX(154)
=======> Monitor T_1 completes with success. 238610 states, 3000 custs, in 112.501 sec.
=======> Customers[3000], Items[1 x 10 = 10], Terminals(1)

runMain demo.marvin.Simulation 12000 1 10
******** LOG FST STATES: AVG(81), TOT(4827361), MIN(1), MAX(176)
=======> Monitor T_1 completes with success. 955706 states, 12000 custs, in 468.546 sec.
=======> Customers[12000], Items[1 x 10 = 10], Terminals(1)
```

Appendix C

List of laws

All of the laws relating to ITL and ITL-Monitor along with their mechanically checked proofs in Isabelle/HOL appear in [CMS19]. Please refer to page vii for information on how to download a copy of that report.

This appendix contains a list of the laws used in this thesis. The laws are annotated as follows:

[CAU] Law from Antonio Cau.

[DRS] An existing ITL law (from Cau or Moszkowski) for which a proof was constructed as part of this thesis, or an original law developed as part of this thesis.

[ITL] Law from [CM16].

[MOS] Law from Ben Moszkowski.

C.1 First order ITL

C.1.1 ITL definitions, derived constructs, axioms and rules

C.1.1.1 Semantic exists

SemanticExists $_{[DRS]}$

$$\mathcal{F}[\![\exists\, v \bullet f]\!](\sigma) = \mathsf{tt} \quad \text{iff} \quad \text{exists}\, \sigma' \text{ s.t. } \sigma \sim_v \sigma',\ \mathcal{F}[\![f]\!](\sigma') = \mathsf{tt} \tag{C.1}$$

C.1.1.2 Frequently-used non-temporal derived constructs

FalseDef [*ITL*]

$$\mathsf{false} \quad \widehat{=} \quad \neg\ \mathsf{true} \tag{C.2}$$

OrDef [*ITL*]

$$f_1 \vee f_2 \quad \widehat{=} \quad \neg\,(\neg\, f_1 \wedge \neg\, f_2) \tag{C.3}$$

ImpDef [*ITL*]

$$f_1 \supset f_2 \quad \widehat{=} \quad \neg\, f_1 \vee f_2 \tag{C.4}$$

EqvDef [*ITL*]

$$f_1 \equiv f_2 \quad \widehat{=} \quad (f_1 \supset f_2) \wedge (f_2 \supset f_1) \tag{C.5}$$

ExistsDef [*ITL*]

$$\exists\, v \bullet f \quad \widehat{=} \quad \neg\ \forall\, v \bullet \neg\, f \tag{C.6}$$

C.1.1.3 Frequently-used temporal derived constructs

StrongNextDef [*ITL*]

$$\bigcirc f \quad \widehat{=} \quad \mathsf{skip}\ ;\ f \tag{C.7}$$

MoreDef [*ITL*]

$$\mathsf{more} \;\widehat{=}\; \bigcirc\,\mathsf{true} \tag{C.8}$$

EmptyDef [*ITL*]

$$\mathsf{empty} \;\widehat{=}\; \neg\ \mathsf{more} \tag{C.9}$$

DiamondDef [*ITL*]

$$\Diamond f \;\widehat{=}\; \mathsf{true}\ ;\ f \tag{C.10}$$

BoxDef [*ITL*]

$$\Box f \;\widehat{=}\; \neg \Diamond \neg f \tag{C.11}$$

WeakNextDef [*ITL*]

$$\textcircled{w}\, f \;\widehat{=}\; \neg \bigcirc \neg f \tag{C.12}$$

DiDef [*ITL*]

$$\textcircled{i}\!\Diamond f \;\widehat{=}\; f\ ;\ \mathsf{true} \tag{C.13}$$

BiDef [*ITL*]

$$\boxed{i}\, f \;\widehat{=}\; \neg\, \textcircled{i}\!\Diamond \neg f \tag{C.14}$$

DaDef [ITL]

$$\diamond_{a} f \;\widehat{=}\; \mathsf{true} \;;\; f \;;\; \mathsf{true} \tag{C.15}$$

BaDef [ITL]

$$\boxed{a}\, f \;\widehat{=}\; \neg \diamond_{a} \neg f \tag{C.16}$$

C.1.1.4 Frequently-used concrete derived constructs

IfThenElseDef [ITL]

$$\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2 \;\widehat{=}\; (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2) \tag{C.17}$$

IfThenDef [ITL]

$$\mathsf{if}\ f_0\ \mathsf{then}\ f_1 \;\widehat{=}\; \mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else\ empty} \tag{C.18}$$

FinDef [ITL]

$$\mathsf{fin}\, f \;\widehat{=}\; \Box(\mathsf{empty} \supset f) \tag{C.19}$$

HaltDef [ITL]

$$\mathsf{halt}\, f \;\widehat{=}\; \Box(\mathsf{empty} \equiv f) \tag{C.20}$$

KeepDef [ITL]

$$\mathsf{keep}\, f \;\widehat{=}\; \boxed{a}\,(\mathsf{skip} \supset f) \tag{C.21}$$

KeepNowDef [ITL]

$$\mathsf{keepnow}\, f \quad \widehat{=} \quad \diamondsuit\!\!\!\cdot\, (\mathsf{skip} \wedge f) \tag{C.22}$$

IterZeroDef [ITL]

$$f^0 \quad \widehat{=} \quad \mathsf{empty} \tag{C.23}$$

IterDef [ITL]

$$f^{n+1} \quad \widehat{=} \quad f \;;\; f^n, \quad [n \geq 0] \tag{C.24}$$

ForDef [ITL]

$$\mathsf{for}\; n \;\mathsf{do}\; f \quad \widehat{=} \quad f^n \tag{C.25}$$

WhileDef [ITL]

$$\mathsf{while}\; f_0 \;\mathsf{do}\; f_1 \quad \widehat{=} \quad (f_0 \wedge f_1)^* \wedge \mathsf{fin}(\neg\, f_0) \tag{C.26}$$

RepeatDef [ITL]

$$\mathsf{repeat}\; f_0 \;\mathsf{until}\; f_1 \quad \widehat{=} \quad f_0 \;;\; \mathsf{while}(\neg\, f_1) \;\mathsf{do}\; f_0 \tag{C.27}$$

C.1.1.5 Frequently-used derived constructs relating to expressions

AssignDef [ITL]

$$A := e \quad \widehat{=} \quad (\bigcirc A) = e \tag{C.28}$$

TemporalEqualityDef [ITL]

$$A \approx e \quad \widehat{=} \quad \Box(A = e) \tag{C.29}$$

TemporalAssignDef [ITL]

$$A \leftarrow e \quad \widehat{=} \quad \mathsf{fin}\, A = e \tag{C.30}$$

GetsDef [ITL]

$$A\, \mathsf{gets}\, e \quad \widehat{=} \quad \mathsf{keep}(A \leftarrow e) \tag{C.31}$$

StableDef [ITL]

$$\mathsf{stable}\, A \quad \widehat{=} \quad A\, \mathsf{gets}\, A \tag{C.32}$$

PaddedDef [ITL]

$$\mathsf{padded}\, A \quad \widehat{=} \quad (\mathsf{stable}\, A \ ; \ \mathsf{skip}) \vee \mathsf{empty} \tag{C.33}$$

PaddedTemporalAssignDef [ITL]

$$A <\!\!\sim e \quad \widehat{=} \quad (A \leftarrow e) \wedge \mathsf{padded}\, A \tag{C.34}$$

LenDef [ITL]

$$\mathsf{len}(n) \quad \widehat{=} \quad \mathsf{skip}^n \tag{C.35}$$

C.1.1.6 Propositional axioms and rules for ITL

ChopAssoc [*ITL*]

$$\vdash \ (f_0 \ ; \ f_1) \ ; \ f_2 \equiv f_0 \ ; (f_1 \ ; \ f_2) \tag{C.36}$$

OrChopImp [*ITL*]

$$\vdash \ (f_0 \lor f_1) \ ; \ f_2 \supset (f_0 \ ; \ f_2) \lor (f_1 \ ; \ f_2) \tag{C.37}$$

ChopOrImp [*ITL*]

$$\vdash \ f_0 \ ; (f_1 \lor f_2) \supset (f_0 \ ; \ f_1) \lor (f_0 \ ; \ f_2) \tag{C.38}$$

EmptyChop [*ITL*]

$$\vdash \ \mathsf{empty} \ ; \ f \equiv f \tag{C.39}$$

ChopEmpty [*ITL*]

$$\vdash \ f \ ; \ \mathsf{empty} \equiv f \tag{C.40}$$

BiBoxChopImpChop [*ITL*]

$$\vdash \ (\boxed{\mathsf{i}}\,(f_0 \supset f_1) \land \Box(f_2 \supset f_3)) \ \supset \ ((f_0 \ ; \ f_2) \supset (f_1 \ ; \ f_3)) \tag{C.41}$$

StateImpBi [*ITL*]

$$\vdash \ w \supset \boxed{\mathsf{i}}\, w \tag{C.42}$$

NextImpNotNextNot [ITL]

$$\vdash \bigcirc f \supset \neg \bigcirc \neg f \tag{C.43}$$

KeepnowImpNotKeepnowNot [ITL]

$$\vdash \mathsf{keepnow}(f) \supset \neg \mathsf{keepnow}(\neg f) \tag{C.44}$$

BoxInduct [ITL]

$$\vdash f \wedge \Box(f \supset ⓦ f) \supset \Box f \tag{C.45}$$

ChopStarEqv [ITL]

$$\vdash f^* \equiv (\mathsf{empty} \vee ((f \wedge \mathsf{more}) ; f^*)) \tag{C.46}$$

MP [ITL]

$$\frac{\vdash f_0 \supset f_1, \quad \vdash f_0}{\vdash f_1} \tag{C.47}$$

BoxGen [ITL]

$$\frac{\vdash f}{\vdash \Box f} \tag{C.48}$$

BiGen [ITL]

$$\frac{\vdash f}{\vdash \boxed{i} f} \tag{C.49}$$

C.1.1.7 First order axioms and rules for ITL

ExistsChopRight [*ITL*]

$$(\exists\, v \bullet (f_1\ ;\ f_2)) \ \supset\ (\exists\, v \bullet f_1)\ ;\ f_2 \quad \text{[where } v \text{ not free in } f_2\text{]} \tag{C.50}$$

ExistsChopLeft [*ITL*]

$$(\exists\, v \bullet (f_1\ ;\ f_2)) \ \supset\ f_1\ ;\ (\exists\, v \bullet f_2) \quad \text{[where } v \text{ not free in } f_1\text{]} \tag{C.51}$$

ForallGen [*ITL*]

$$\frac{\vdash\ f}{\vdash\ \ \forall\, v \bullet f}\ \text{[for any variable } v\text{]} \tag{C.52}$$

C.1.2 Time reversal

C.1.2.1 Time reversal definitions and laws

ReflectionRule [*MOS*]

$$\models f \quad \text{iff} \quad \models f^r \tag{C.53}$$

TRTrue [*MOS*]

$$\vdash\ \ \mathsf{true}^r \equiv \mathsf{true} \tag{C.54}$$

TRSkip [*MOS*]

$$\vdash\ \ \mathsf{skip}^r \equiv \mathsf{skip} \tag{C.55}$$

TRState $_{[MOS]}$

$$\vdash \; w^r \equiv \mathsf{fin}\, w \tag{C.56}$$

TRNot $_{[MOS]}$

$$\vdash \; (\neg f)^r \equiv \neg f^r \tag{C.57}$$

TROr $_{[MOS]}$

$$\vdash \; (f \lor g)^r \equiv f^r \lor g^r \tag{C.58}$$

TRChop $_{[MOS]}$

$$\vdash \; (f \;;\; g)^r \equiv g^r \;;\; f^r \tag{C.59}$$

TRChopstar $_{[MOS]}$

$$\vdash \; (f^*)^r \equiv (f^r)^* \tag{C.60}$$

TRAnd $_{[DRS]}$

$$\vdash \; (f \land g)^r \equiv f^r \land g^r \tag{C.61}$$

TRImp $_{[DRS]}$

$$\vdash \; (f \supset g)^r \equiv f^r \supset g^r \tag{C.62}$$

TREqv [*DRS*]

$$\vdash \ (f \equiv g)^r \equiv (f^r \equiv g^r) \tag{C.63}$$

TRMore [*MOS*]

$$\vdash \ \mathsf{more}^r \equiv \mathsf{more} \tag{C.64}$$

TREmpty [*DRS*]

$$\vdash \ \mathsf{empty}^r \equiv \mathsf{empty} \tag{C.65}$$

TRDi [*DRS*]

$$\vdash \ (\diamondsuit_i f)^r \equiv \Diamond f^r \tag{C.66}$$

TRBi [*MOS*]

$$\vdash \ (\boxdot_i f)^r \equiv \Box f^r \tag{C.67}$$

TRDiamond [*DRS*]

$$\vdash \ (\Diamond f)^r \equiv \diamondsuit_i f^r \tag{C.68}$$

TRBox [*DRS*]

$$\vdash \ (\Box f)^r \equiv \boxdot_i f^r \tag{C.69}$$

C.1.3 Definitions and laws related to exportable commitments

BmDef [MOS]

$$\boxed{\mathsf{m}}\; f \quad \widehat{=} \quad \Box(\mathsf{more} \supset f) \tag{C.70}$$

DmDef [MOS]

$$\mathrlap{\Diamond}\,{\scriptstyle\mathsf{m}}\; f \quad \widehat{=} \quad \neg\, \boxed{\mathsf{m}}\; \neg f \tag{C.71}$$

BaEqvBmAndFin [MOS]

$$\vdash \;\; \boxed{\mathsf{a}}\; f \equiv \boxed{\mathsf{m}}\; f \wedge \mathsf{fin}\, f \tag{C.72}$$

BmDiFixCS [MOS]

$$\vdash \;\; \boxed{\mathsf{m}}\; \mathrlap{\Diamond}\,{\scriptstyle\mathsf{i}}\, f \equiv (\boxed{\mathsf{m}}\; \mathrlap{\Diamond}\,{\scriptstyle\mathsf{i}}\, f)^* \tag{C.73}$$

FixDiInfBmDIFixCS [MOS]

$$DI \equiv \mathrlap{\Diamond}\,{\scriptstyle\mathsf{i}}\; DI \vdash \;\; \boxed{\mathsf{m}}\;\; DI \equiv (\boxed{\mathsf{m}}\;\; DI)^* \tag{C.74}$$

FixDaInfBmDAFixCS [MOS]

$$DA \equiv \mathrlap{\Diamond}\,{\scriptstyle\mathsf{a}}\; DA \vdash \;\; \boxed{\mathsf{m}}\;\; DA \equiv (\boxed{\mathsf{m}}\;\; DA)^* \tag{C.75}$$

StateImpDiamondStateFixDi [MOS]

$$\vdash \;\; (w \supset \Diamond w') \equiv \mathrlap{\Diamond}\,{\scriptstyle\mathsf{i}}\,(w \supset \Diamond w') \tag{C.76}$$

BmStateImpDiamondStateFixCS [MOS]

$$\vdash \boxed{\mathsf{m}}\ (w \supset \Diamond w') \equiv (\boxed{\mathsf{m}}\ (w \supset \Diamond w'))^* \qquad \text{(C.77)}$$

StateImpDAFixDi [MOS]

$$DA \equiv \diamondsuit\!\!\!\!\cdot_{a}\ DA \vdash\ (w \supset DA) \equiv \diamondsuit\!\!\!\!\cdot_{i}\,(w \supset DA) \qquad \text{(C.78)}$$

BmStateImpDAFixCS [MOS]

$$DA \equiv \diamondsuit\!\!\!\!\cdot_{a}\ DA \vdash\ \boxed{\mathsf{m}}\ (w \supset DA) \equiv (\boxed{\mathsf{m}}\ (w \supset DA))^* \qquad \text{(C.79)}$$

C.1.4 Always-followed-by

AfbDef [ITL]

$$f \mapsto w \quad \widehat{=} \quad \boxed{\mathsf{a}}\,(f \supset \mathsf{fin}\, w) \qquad \text{(C.80)}$$

SafbDef [ITL]

$$f \leftrightarrow w \quad \widehat{=} \quad \boxed{\mathsf{a}}\,(f \equiv \mathsf{fin}\, w) \qquad \text{(C.81)}$$

AltAfbDef [ITL]

$$f \mapsto w \quad \widehat{=} \quad \boxed{\mathsf{a}}\,(f \supset \mathsf{fin}\, w) \qquad \text{(C.82)}$$

AltSafbDef [ITL]

$$f \leftrightarrow w \quad \widehat{=} \quad \boxed{\mathsf{a}}\,(f \equiv \mathsf{fin}\, w) \qquad \text{(C.83)}$$

C.1.5 Commonly used ITL laws

C.1.5.1 Box, Diamond, Now

NowImpDiamond [MOS]

$$\vdash \; f \supset \Diamond f \tag{C.84}$$

BoxImpNow [DRS]

$$\vdash \; \Box f \supset f \tag{C.85}$$

C.1.5.2 State, skip, true, false, empty, more with chop

StateAndChop [MOS]

$$\vdash \; (w \wedge f) \; ; \; g \equiv w \wedge (f \; ; \; g) \tag{C.86}$$

MoreEqvTrueChopSkip [DRS]

$$\vdash \; \mathsf{more} \equiv \mathsf{true} \; ; \; \mathsf{skip} \tag{C.87}$$

SkipTrueEqvTrueSkip [DRS]

$$\vdash \; \mathsf{skip} \; ; \; \mathsf{true} \equiv \mathsf{true} \; ; \; \mathsf{skip} \tag{C.88}$$

BiChopImpChop [MOS]

$$\vdash \; \boxdot\,(f \supset f') \supset ((f \; ; \; g) \supset (f' \; ; \; g)) \tag{C.89}$$

BoxChopImpChop [MOS]

$$\vdash \ \Box(g \supset g') \supset ((f \ ; \ g) \supset (f \ ; \ g')) \tag{C.90}$$

DiIntro [MOS]

$$\vdash \ f \supset \diamondsuit\!\!\!\!\cdot\; f \tag{C.91}$$

BiElim [MOS]

$$\vdash \ \boxdot f \supset f \tag{C.92}$$

StateEqvBi [MOS]

$$\vdash \ \boxdot w \equiv w \tag{C.93}$$

TrueEqvTrueChopTrue [MOS]

$$\vdash \ \mathsf{true} \equiv \mathsf{true} \ ; \ \mathsf{true} \tag{C.94}$$

MoreEqvSkipChopTrue [MOS]

$$\vdash \ \mathsf{more} \equiv \mathsf{skip} \ ; \ \mathsf{true} \tag{C.95}$$

MoreEqvNotEmpty [CAU]

$$\vdash \ \mathsf{more} \equiv \neg \ \mathsf{empty} \tag{C.96}$$

MoreEqvMoreChopTrue [CAU]

$$\vdash \ \mathsf{more} \equiv \mathsf{more}\ ;\ \mathsf{true} \tag{C.97}$$

ChopFalseEqvFalse [DRS]

$$\vdash \ f\ ;\ \mathsf{false} \equiv \mathsf{false} \tag{C.98}$$

FalseChopEqvFalse [DRS]

$$\vdash \ \mathsf{false}\ ;\ f \equiv \mathsf{false} \tag{C.99}$$

BoxImpFinImpDiamondImpFin [DRS]

$$\Box(f \supset \mathsf{fin}\ w) \ \supset\ (\Diamond f \supset \mathsf{fin}\ w) \tag{C.100}$$

C.1.5.3 Implication and equivalence through chop

LeftChopImpChop [MOS]

$$\vdash\ f \supset f' \quad \Rightarrow \quad \vdash\ (f\ ;\ g) \supset (f'\ ;\ g) \tag{C.101}$$

RightChopImpChop [MOS]

$$\vdash\ g \supset g' \quad \Rightarrow \quad \vdash\ (f\ ;\ g) \supset (f\ ;\ g') \tag{C.102}$$

LeftChopEqvChop [DRS]

$$\vdash\ f \equiv f' \quad \Rightarrow \quad \vdash\ (f\ ;\ g) \equiv (f'\ ;\ g) \tag{C.103}$$

RightChopEqvChop [*MOS*]

$$\vdash \ g \equiv g' \quad \Rightarrow \quad \vdash \ (f \ ; \ g) \equiv (f \ ; \ g') \tag{C.104}$$

LeftAndChopImp [*DRS*]

$$\vdash \ (f \wedge f') \ ; \ g \supset f \ ; \ g \wedge f' \ ; \ g \tag{C.105}$$

RightAndChopImp [*DRS*]

$$\vdash \ f \ ; (g \wedge g') \supset f \ ; \ g \wedge f \ ; \ g' \tag{C.106}$$

ChopOrEqv [*MOS*]

$$\vdash \ f \ ; (g \vee g') \equiv f \ ; \ g \vee f \ ; \ g' \tag{C.107}$$

OrChopEqv [*MOS*]

$$\vdash \ (f \vee f') \ ; \ g \equiv f \ ; \ g \vee f' \ ; \ g \tag{C.108}$$

C.1.5.4 Initial intervals

DiImpDi [*MOS*]

$$\vdash \ f \supset g \quad \Rightarrow \quad \vdash \ \Diamond_{\mathrm{i}} f \supset \Diamond_{\mathrm{i}} g \tag{C.109}$$

DiEqvDi [*MOS*]

$$\vdash \ f \supset g \quad \Rightarrow \quad \vdash \ \Diamond_{\mathrm{i}} f \supset \Diamond_{\mathrm{i}} g \tag{C.110}$$

BiImpBiRule [*MOS*]

$$\vdash\ f \supset g \qquad \Rightarrow \qquad \vdash\ \boxed{i}\, f \supset \boxed{i}\, g \tag{C.111}$$

DiState [*MOS*]

$$\vdash\ \Diamond\!\!\!\!\!\;\;i\ w \equiv w \tag{C.112}$$

DiNotEqvNotBi [*MOS*]

$$\vdash\ \Diamond\!\!\!\!\!\;\;i\ \neg f \equiv \neg\, \boxed{i}\, f \tag{C.113}$$

NotDiEqvBiNot [*MOS*]

$$\vdash\ \neg\, \Diamond\!\!\!\!\!\;\;i\ f \equiv \boxed{i}\, \neg f \tag{C.114}$$

DiEqvNotBiNot [*MOS*]

$$\vdash\ \Diamond\!\!\!\!\!\;\;i\ f \equiv \neg\, \boxed{i}\, \neg f \tag{C.115}$$

ChopImpDi [*MOS*]

$$\vdash\ f\ ;\ g \supset \Diamond\!\!\!\!\!\;\;i\ f \tag{C.116}$$

DiEqvDiDi [*MOS*]

$$\vdash\ \Diamond\!\!\!\!\!\;\;i\ f \equiv \Diamond\!\!\!\!\!\;\;i\ \Diamond\!\!\!\!\!\;\;i\ f \tag{C.117}$$

BiEqvBiBi [*MOS*]

$$\vdash \ \boxed{i}\, f \equiv \boxed{i}\,\boxed{i}\, f \tag{C.118}$$

DiEmpty [*MOS*]

$$\vdash \ \diamondsuit_{i}\ \mathsf{empty} \tag{C.119}$$

NotBiMore [*DRS*]

$$\vdash \ \neg\ \boxed{i}\ \mathsf{more} \tag{C.120}$$

DiOrEqv [*MOS*]

$$\vdash \ \diamondsuit_{i}\,(f \lor g) \equiv \diamondsuit_{i}\, f \lor \diamondsuit_{i}\, g \tag{C.121}$$

BiAndEqv [*DRS*]

$$\vdash \ \boxed{i}\,(f \land g) \equiv \boxed{i}\, f \land \boxed{i}\, g \tag{C.122}$$

AndChopA [*MOS*]

$$\vdash \ (f \land f')\ ;\ g \supset f\ ;\ g \tag{C.123}$$

DiAndImpAnd [*MOS*]

$$\vdash \ \diamondsuit_{i}\,(f \land g) \supset \diamondsuit_{i}\, f \land \diamondsuit_{i}\, g \tag{C.124}$$

BoxStateEqvBiFinState [DRS]

$$\vdash \ \Box w \equiv \boxed{\scriptstyle i}\,(\mathsf{fin}\ w) \tag{C.125}$$

DiamondStateEqvDiFinState [DRS]

$$\vdash \ \Diamond w \equiv \Diamond_{i}\,(\mathsf{fin}\ w) \tag{C.126}$$

C.1.5.5 Induction

EmptyNextInductA [MOS]

$$\vdash \ \mathsf{empty} \supset f, \quad \vdash \ \bigcirc f \supset f \qquad \Rightarrow \qquad \vdash \ f \tag{C.127}$$

EmptyChopSkipInduct [DRS]

$$\vdash \ \mathsf{empty} \supset f, \quad \vdash \ (f\ ;\ \mathsf{skip}) \supset f \qquad \Rightarrow \qquad \vdash \ f \tag{C.128}$$

C.1.5.6 Chop and negation

NotSkipNotChop [DRS]

$$\vdash \ \neg\,(\mathsf{skip}\ ;\ \neg\, g) \equiv \mathsf{empty} \vee (\mathsf{skip}\ ;\ g) \tag{C.129}$$

NotNotChopSkip [CAU]

$$\vdash \ \neg\,(\neg\, f\ ;\ \mathsf{skip}) \equiv \mathsf{empty} \vee (f\ ;\ \mathsf{skip}) \tag{C.130}$$

ChopAndNotChopImp [*MOS*]

$$\vdash \; f \;;\; g \wedge \neg\, (f \;;\; h) \supset f \;;\, (g \wedge \neg\, h) \tag{C.131}$$

RightChopAndNot [*CAU*]

$$\vdash \; f \;;\; g \wedge \neg\, (h \;;\; g) \supset (f \wedge \neg\, h) \;;\; g \tag{C.132}$$

ChopAndNotNotChop [*CAU*]

$$\vdash \; f \;;\; h \wedge \neg\, (\neg\, g \;;\; h) \supset (f \wedge g) \;;\; \mathsf{true} \tag{C.133}$$

SkipNotEmptyOrMoreMore [*CAU*]

$$\vdash \; \mathsf{skip} \equiv \neg\, (\mathsf{empty} \vee \mathsf{more} \;;\; \mathsf{more}) \tag{C.134}$$

C.1.5.7 Strong and weak next

NextEqvMoreAndWeakNext [*MOS*]

$$\vdash \; \bigcirc f \equiv \mathsf{more} \wedge ⓦ\, f \tag{C.135}$$

NotWeakNextNotEqvNext [*DRS*]

$$\vdash \; \neg\, ⓦ\, \neg\, f \equiv \bigcirc f \tag{C.136}$$

WeakNextEqvMoreImpStrongNext [*DRS*]

$$\vdash \; ⓦ\, f \equiv \mathsf{more} \supset \bigcirc f \tag{C.137}$$

C.1.5.8 Existential quantifcation through chop

ExistsChopLeftEqv [DRS]

$$\vdash \ (\exists\, v \bullet (f_1 \ ; \ f_2)) \ \equiv \ f_1 \ ; (\exists\, v \bullet f_2) \ \ \text{[where } v \text{ not free in } f_1] \tag{C.138}$$

C.1.5.9 Chop with empty and more

ChopEmptyAndEmpty [DRS]

$$\vdash \ f \ ; \ g \wedge \mathsf{empty} \equiv f \wedge g \wedge \mathsf{empty} \tag{C.139}$$

ChopSkipAndEmptyEqvFalse [DRS]

$$\vdash \ f \ ; \ \mathsf{skip} \wedge \mathsf{empty} \equiv \mathsf{false} \tag{C.140}$$

ChopSkipImpMore [DRS]

$$\vdash \ (f \ ; \ \mathsf{skip}) \supset \mathsf{more} \tag{C.141}$$

MoreImpImpChopSkipEqv [DRS]

$$\vdash \ \mathsf{more} \supset ((f \supset g) \ ; \ \mathsf{skip} \equiv ((f \ ; \ \mathsf{skip}) \supset (g \ ; \ \mathsf{skip}))) \tag{C.142}$$

C.1.6 Fixed length intervals

C.1.6.1 Properties of interval length

LenZeroEqvEmpty [*DRS*]

$$\vdash \;\; \mathsf{len}(0) \equiv \mathsf{empty} \tag{C.143}$$

LenOneEqvSkip [*DRS*]

$$\vdash \;\; \mathsf{len}(1) \equiv \mathsf{skip} \tag{C.144}$$

LenNPlusOneA [*DRS*]

$$\vdash \;\; \mathsf{len}(n+1) \equiv \mathsf{skip} \;;\; \mathsf{len}(n) \tag{C.145}$$

LenEqvLenChopLen [*DRS*]

$$\vdash \;\; \mathsf{len}(i+j) \equiv \mathsf{len}(i) \;;\; \mathsf{len}(j) \tag{C.146}$$

LenNPlusOneB [*DRS*]

$$\vdash \;\; \mathsf{len}(n+1) \equiv \mathsf{len}(n) \;;\; \mathsf{skip} \tag{C.147}$$

ExistsLen [*DRS*]

$$\vdash \;\; \exists\, k \bullet \mathsf{len}(k) \tag{C.148}$$

AndExistsLen [*DRS*]

$$\vdash\ f \equiv f \wedge \exists\, k \bullet \mathsf{len}(k) \tag{C.149}$$

LenIffModSig [*DRS*]

$$\mathcal{F}[\![\mathsf{len}(k)]\!](\sigma) = \mathsf{tt} \quad \text{iff} \quad |\sigma| = k \tag{C.150}$$

LFixedAndDistr [*DRS*]

$$\vdash\ (f \wedge \mathsf{len}(k))\ ;\ p \wedge (g \wedge \mathsf{len}(k))\ ;\ q \equiv (f \wedge g \wedge \mathsf{len}(k))\ ; (p \wedge q) \tag{C.151}$$

RFixedAndDistr [*DRS*]

$$\vdash\ p\ ; (f \wedge \mathsf{len}(k)) \wedge q\ ; (g \wedge \mathsf{len}(k)) \equiv (p \wedge q)\ ; (f \wedge g \wedge \mathsf{len}(k)) \tag{C.152}$$

LFixedAndDistrA [*DRS*]

$$\vdash\ (f \wedge \mathsf{len}(k))\ ;\ p \wedge (g \wedge \mathsf{len}(k))\ ;\ p \equiv (f \wedge g \wedge \mathsf{len}(k))\ ;\ p \tag{C.153}$$

LFixedAndDistrB [*DRS*]

$$\vdash\ (f \wedge \mathsf{len}(k))\ ;\ p \wedge (f \wedge \mathsf{len}(k))\ ;\ q \equiv (f \wedge \mathsf{len}(k))\ ; (p \wedge q) \tag{C.154}$$

RFixedAndDistrA [*DRS*]

$$\vdash\ p\ ; (f \wedge \mathsf{len}(k)) \wedge p\ ; (g \wedge \mathsf{len}(k)) \equiv p\ ; (f \wedge g \wedge \mathsf{len}(k)) \tag{C.155}$$

RFixedAndDistrB [*DRS*]

$$\vdash\ p\ ;\ (f \wedge \mathsf{len}(k)) \wedge q\ ;\ (f \wedge \mathsf{len}(k)) \equiv (p \wedge q)\ ;\ (f \wedge \mathsf{len}(k)) \tag{C.156}$$

ChopSkipAndChopSkip [*DRS*]

$$\vdash\ f\ ;\ \mathsf{skip} \wedge g\ ;\ \mathsf{skip} \equiv (f \wedge g)\ ;\ \mathsf{skip} \tag{C.157}$$

BiAndChopSkipEqv [*DRS*]

$$\vdash\ \boxed{\scriptstyle i}\,(f \wedge g)\ ;\ \mathsf{skip} \equiv \boxed{\scriptstyle i}\,f\ ;\ \mathsf{skip} \wedge \boxed{\scriptstyle i}\,g\ ;\ \mathsf{skip} \tag{C.158}$$

DiAndChopSkipImp [*DRS*]

$$\vdash\ \Diamond_i\,(f \wedge g)\ ;\ \mathsf{skip} \supset \Diamond_i\,f\ ;\ \mathsf{skip} \wedge \Diamond_i\,g\ ;\ \mathsf{skip} \tag{C.159}$$

NotChopFixed [*DRS*]

$$\vdash\ \neg\,(f\ ;\ h) \equiv \neg\,\Diamond h \vee (\neg\, f\ ;\ h) \quad \mathsf{where}\ \ h \equiv g \wedge \mathsf{len}(k) \tag{C.160}$$

NotFixedChop [*DRS*]

$$\vdash\ \neg\,(h\ ;\ f) \equiv \neg\,\Diamond_i\,h \vee (h\ ;\ \neg\, f) \quad \mathsf{where}\ \ h \equiv g \wedge \mathsf{len}(k) \tag{C.161}$$

C.1.7 Further laws with initial intervals

ImpEqvDi [*DRS*]

$$\vdash\ f \supset (f \equiv \Diamond_i\,f) \tag{C.162}$$

AndDiEqv [*DRS*]

$$\vdash\ f \wedge \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em f \equiv f \tag{C.163}$$

OrDiEqvDi [*DRS*]

$$\vdash\ f \vee \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em f \equiv \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em f \tag{C.164}$$

AndDiOrEqv [*DRS*]

$$\vdash\ f \wedge (\Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em f \vee g) \equiv f \tag{C.165}$$

DiChopImpDiA [*DRS*]

$$\vdash\ \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em f\ ;\ g \supset \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em f \tag{C.166}$$

DiChopImpDiB [*DRS*]

$$\vdash\ \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em (f\ ;\ g) \supset \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em f \tag{C.167}$$

BiNotImpNotBiChop [*DRS*]

$$\vdash\ \boxed{\scriptstyle i}\, \neg f \supset \boxed{\scriptstyle i}\, \neg (f\ ;\ g) \tag{C.168}$$

DiDiAndDiEqvDiAndDi [*DRS*]

$$\vdash\ \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em (\Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em f \wedge \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em g) \equiv \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em f \wedge \Diamond\kern-0.6em{\scriptscriptstyle i}\kern0.35em g \tag{C.169}$$

AndDiAndDiEqvAndDi [*DRS*]

$$\vdash \; f \wedge \Diamond_i (f \wedge \Diamond_i g) \equiv f \wedge \Diamond_i g \tag{C.170}$$

DiAndDiEqvDiAndDiOrDiAndDi [*DRS*]

$$\vdash \; \Diamond_i f \wedge \Diamond_i g \equiv \Diamond_i (f \wedge \Diamond_i g) \vee \Diamond_i (g \wedge \Diamond_i f) \tag{C.171}$$

BiOrBiImpBiOr [*DRS*]

$$\vdash \; \Box_i f \vee \Box_i g \supset \Box_i (f \vee g) \tag{C.172}$$

BiOrBiEqvBiBiOrBi [*DRS*]

$$\vdash \; \Box_i f \vee \Box_i g \equiv \Box_i (\Box_i f \vee \Box_i g) \tag{C.173}$$

MoreAndBiImpBiChopSkip [*DRS*]

$$\vdash \; \mathsf{more} \wedge \Box_i f \supset \Box_i f \; ; \; \mathsf{skip} \tag{C.174}$$

DiEqvOrDiChopSkipA [*DRS*]

$$\vdash \; \Diamond_i f \equiv f \vee \Diamond_i (f \; ; \; \mathsf{skip}) \tag{C.175}$$

DiEqvOrDiChopSkipB [*DRS*]

$$\vdash \; \Diamond_i f \equiv f \vee (\Diamond_i f \; ; \; \mathsf{skip}) \tag{C.176}$$

BiEqvAndEmptyOrBiChopSkip [*DRS*]

$$\vdash \ \boxed{\text{i}}\, f \equiv f \wedge (\mathsf{empty} \vee (\boxed{\text{i}}\, f \ ; \ \mathsf{skip})) \tag{C.177}$$

DiamondEqvOrStrongNextDiamond [*DRS*]

$$\vdash \ \Diamond f \equiv f \vee \bigcirc \Diamond f \tag{C.178}$$

BoxEqvAndWeakNextBox [*DRS*]

$$\vdash \ \Box f \equiv f \wedge \text{\textcircled{w}}\ \Box f \tag{C.179}$$

BiAndDiEqvBiAndDiAndBi [*DRS*]

$$\vdash \ \boxed{\text{i}}\, f \wedge \Diamond_{\text{i}}\, g \equiv \boxed{\text{i}}\, f \wedge \Diamond_{\text{i}}\, (g \wedge \boxed{\text{i}}\, f) \tag{C.180}$$

DiAndBiImpDiAndBi [*DRS*]

$$\vdash \ \Diamond_{\text{i}}\, f \wedge \boxed{\text{i}}\, g \supset \Diamond_{\text{i}}\, (f \wedge \boxed{\text{i}}\, g) \tag{C.181}$$

BiAndEmptyEqvAndEmpty [*DRS*]

$$\vdash \ \boxed{\text{i}}\, f \wedge \mathsf{empty} \equiv f \wedge \mathsf{empty} \tag{C.182}$$

DiAndEmptyEqvAndEmpty [*DRS*]

$$\vdash \ \Diamond_{\text{i}}\, f \wedge \mathsf{empty} \equiv f \wedge \mathsf{empty} \tag{C.183}$$

BiEmptyEqvEmpty [CAU]

$$\vdash \quad \boxed{\text{i}}\,\mathsf{empty} \equiv \mathsf{empty} \tag{C.184}$$

C.1.8 Strict initial intervals

BsDef [DRS]

$$\boxed{\text{s}}\,f \quad \widehat{=} \quad \mathsf{empty} \lor \boxed{\text{i}}\,f \ ; \ \mathsf{skip} \tag{C.185}$$

DsDef [DRS]

$$\diamond\!\!\!\text{s}\; f \quad \widehat{=} \quad \neg\, \boxed{\text{s}}\, \neg\, f \tag{C.186}$$

DsMoreDi [DRS]

$$\vdash \quad \diamond\!\!\!\text{s}\; f \equiv \mathsf{more} \land \diamond\!\!\!\text{i}\; f \ ; \ \mathsf{skip} \tag{C.187}$$

DsDi [DRS]

$$\vdash \quad \diamond\!\!\!\text{s}\; f \equiv \diamond\!\!\!\text{i}\; f \ ; \ \mathsf{skip} \tag{C.188}$$

C.1.8.1 Duality

BsEqvNotDsNot [DRS]

$$\vdash \quad \boxed{\text{s}}\,f \equiv \neg\, \diamond\!\!\!\text{s}\; \neg\, f \tag{C.189}$$

NotBsEqvDsNot [*DRS*]

$$\vdash \neg \boxed{\mathsf{s}}\, f \equiv \Diamond_{\mathsf{s}} \neg f \tag{C.190}$$

NotDsEqvBsNot [*DRS*]

$$\vdash \neg \Diamond_{\mathsf{s}}\, f \equiv \boxed{\mathsf{s}} \neg f \tag{C.191}$$

NotDsAndEmpty [*DRS*]

$$\vdash \neg (\Diamond_{\mathsf{s}}\, f \wedge \mathsf{empty}) \tag{C.192}$$

C.1.8.2 Distribution through conjunction and disjunction

BsMoreEqvEmpty [*DRS*]

$$\vdash \boxed{\mathsf{s}}\ \mathsf{more} \equiv \mathsf{empty} \tag{C.193}$$

BsAndEqv [*DRS*]

$$\vdash \boxed{\mathsf{s}}\, f \wedge \boxed{\mathsf{s}}\, g \equiv \boxed{\mathsf{s}}\, (f \wedge g) \tag{C.194}$$

DsOrEqv [*DRS*]

$$\vdash \Diamond_{\mathsf{s}}\, f \vee \Diamond_{\mathsf{s}}\, g \equiv \Diamond_{\mathsf{s}}\, (f \vee g) \tag{C.195}$$

BsOrImp [*DRS*]

$$\vdash \boxed{\mathsf{s}}\, f \vee \boxed{\mathsf{s}}\, g \supset \boxed{\mathsf{s}}\, (f \vee g) \tag{C.196}$$

DsAndImp [*DRS*]

$$\vdash \Diamond_{s} (f \wedge g) \supset \Diamond_{s} f \wedge \Diamond_{s} g \tag{C.197}$$

DsAndImpElimL [*DRS*]

$$\vdash \Diamond_{s} (f \wedge g) \supset \Diamond_{s} f \tag{C.198}$$

DsAndImpElimR [*DRS*]

$$\vdash \Diamond_{s} (f \wedge g) \supset \Diamond_{s} g \tag{C.199}$$

C.1.8.3 Useful implications

BiImpBs [*DRS*]

$$\vdash \boxed{i}\, f \supset \boxed{s}\, f \tag{C.200}$$

BsImpBsBs [*DRS*]

$$\vdash \boxed{s}\, f \supset \boxed{s}\, \boxed{s}\, f \tag{C.201}$$

DsImpDi [*DRS*]

$$\vdash \Diamond_{s} f \supset \Diamond_{i} f \tag{C.202}$$

BsImpBsRule [*DRS*]

$$\vdash f \supset g \quad \Rightarrow \quad \vdash \boxed{s}\, f \supset \boxed{s}\, g \tag{C.203}$$

DsChopImpDsB [*DRS*]

$$\vdash \ \Diamond\!\!\!\!\!\!\cdot_{s}\ (f \ ; \ g) \supset \Diamond_{s} f \tag{C.204}$$

NotBsImpBsNotChop [*DRS*]

$$\vdash \ \boxed{s}\, \neg f \supset \boxed{s}\, \neg (f \ ; \ g) \tag{C.205}$$

C.1.8.4 Relating strict and non-strict initial intervals

BsOrBsEqvBsBiOrBi [*DRS*]

$$\vdash \ \boxed{s}\, f \vee \boxed{s}\, g \equiv \boxed{s}\, (\boxed{i}\, f \vee \boxed{i}\, g) \tag{C.206}$$

DiOrDsEqvDi [*DRS*]

$$\vdash \ \Diamond_{i} f \vee \Diamond_{s} f \equiv \Diamond_{i} f \tag{C.207}$$

DiAndDsEqvDs [*DRS*]

$$\vdash \ \Diamond_{i} f \wedge \Diamond_{s} f \equiv \Diamond_{s} f \tag{C.208}$$

OrDsEqvDi [*DRS*]

$$\vdash \ f \vee \Diamond_{s} f \equiv \Diamond_{i} f \tag{C.209}$$

AndBsEqvBi [*DRS*]

$$\vdash \ f \wedge \boxed{s}\, f \equiv \boxed{i}\, f \tag{C.210}$$

BsEqvBsBi [*DRS*]

$$\vdash \ \boxed{\mathsf{s}}\, f \equiv \boxed{\mathsf{s}}\, \boxed{\mathsf{i}}\, f \tag{C.211}$$

StateImpBs [*DRS*]

$$\vdash \ w \supset \boxed{\mathsf{s}}\, w \tag{C.212}$$

DsAndDsEqvDsAndDiOrDsAndDi [*DRS*]

$$\vdash \ \Diamond_{\mathsf{s}}\, f \wedge \Diamond_{\mathsf{s}}\, g \equiv \Diamond_{\mathsf{s}}\, (f \wedge \Diamond_{\mathsf{i}}\, g) \vee \Diamond_{\mathsf{s}}\, (g \wedge \Diamond_{\mathsf{i}}\, f) \tag{C.213}$$

BsEqvBiMoreImpChop [*DRS*]

$$\vdash \ \boxed{\mathsf{s}}\, f \equiv \boxed{\mathsf{i}}\, (\mathsf{more} \supset f \ ; \ \mathsf{skip}) \tag{C.214}$$

C.1.8.5 Strict final intervals

BtDef [*DRS*]

$$\boxed{\mathsf{t}}\, f \quad \widehat{=} \quad \mathsf{empty} \vee \mathsf{skip} \ ; \ \Box f \tag{C.215}$$

DtDef [*DRS*]

$$\Diamond_{\mathsf{t}}\, f \quad \widehat{=} \quad \neg\, \boxed{\mathsf{t}}\, \neg\, f \tag{C.216}$$

BsrEqvBtr [*DRS*]

$$\vdash \ (\boxed{\mathsf{s}}\, f)^r \equiv \boxed{\mathsf{t}}\, f^r \tag{C.217}$$

DsrEqvDtr [*DRS*]

$$\vdash \ (\Diamond\!\!\!\text{s}\, f)^r \equiv \Diamond\!\!\!\text{t}\, f^r \tag{C.218}$$

BtrEqvBsr [*DRS*]

$$\vdash \ (\boxed{\text{t}}\, f)^r \equiv \boxed{\text{s}}\, f^r \tag{C.219}$$

DtrEqvDsr [*DRS*]

$$\vdash \ (\Diamond\!\!\!\text{t}\, f)^r \equiv \Diamond\!\!\!\text{s}\, f^r \tag{C.220}$$

AlwaysImpBt [*DRS*]

$$\vdash \ \Box f \supset \boxed{\text{t}}\, f \tag{C.221}$$

C.1.9 First occurrence operator

FstDef [*DRS*]

$$\rhd f \ \ \widehat{=} \ \ f \wedge \boxed{\text{s}} \neg f \tag{C.222}$$

C.1.9.1 First with conjunction and disjunction

FstWithAndImp [*DRS*]

$$\vdash \ \rhd f \wedge g \supset \rhd(f \wedge g) \tag{C.223}$$

FstWithOrEqv [*DRS*]

$$\vdash \quad \triangleright(f \vee g) \equiv (\triangleright f \wedge \boxed{s} \neg g) \vee (\triangleright g \wedge \boxed{s} \neg f) \tag{C.224}$$

FstFstAndEqvFstAnd [*DRS*]

$$\vdash \quad \triangleright(\triangleright f \wedge g) \equiv \triangleright f \wedge g \tag{C.225}$$

C.1.9.2 First with true, false, empty, more

FstTrue [*DRS*]

$$\vdash \quad \triangleright \mathsf{true} \equiv \mathsf{empty} \tag{C.226}$$

FstFalse [*DRS*]

$$\vdash \quad \triangleright \mathsf{false} \equiv \mathsf{false} \tag{C.227}$$

FstChopFalseEqvFalse [*DRS*]

$$\vdash \quad \triangleright f \;;\; \mathsf{false} \equiv \mathsf{false} \tag{C.228}$$

FstEmpty [*DRS*]

$$\vdash \quad \triangleright \mathsf{empty} \equiv \mathsf{empty} \tag{C.229}$$

FstAndEmptyEqvAndEmpty [*DRS*]

$$\vdash \quad \triangleright f \wedge \mathsf{empty} \equiv f \wedge \mathsf{empty} \tag{C.230}$$

FstEmptyOrEqvEmpty [*DRS*]

$$\vdash \quad \triangleright(\mathsf{empty} \vee f) \equiv \mathsf{empty} \tag{C.231}$$

FstChopEmptyEqvFstChopFstEmpty [*DRS*]

$$\vdash \quad \triangleright f \ ; \ g \wedge \mathsf{empty} \equiv \triangleright f \ ; \ \triangleright g \wedge \mathsf{empty} \tag{C.232}$$

FstMoreEqvSkip [*DRS*]

$$\vdash \quad \triangleright \mathsf{more} \equiv \mathsf{skip} \tag{C.233}$$

C.1.9.3 First with initial intervals

FstOrDiEqvDi [*DRS*]

$$\vdash \quad \triangleright f \vee \diamondsuit f \equiv \diamondsuit f \tag{C.234}$$

FstAndDiEqvFst [*DRS*]

$$\vdash \quad \triangleright f \wedge \diamondsuit f \equiv \triangleright f \tag{C.235}$$

DiEqvDiFst [*DRS*]

$$\vdash \quad \diamondsuit f \equiv \diamondsuit \triangleright f \tag{C.236}$$

FstDiEqvFst [*DRS*]

$$\vdash \quad \triangleright \diamondsuit f \equiv \triangleright f \tag{C.237}$$

DiAndFstOrEqvFstOrDiAnd [*DRS*]

$$\vdash \ {\Diamond\mkern-12mu{\scriptstyle i}\mkern6mu} f \wedge (\rhd f \vee g) \equiv \rhd f \vee ({\Diamond\mkern-12mu{\scriptstyle i}\mkern6mu} f \wedge g) \tag{C.238}$$

DiOrFstAndEqvDi [*DRS*]

$$\vdash \ {\Diamond\mkern-12mu{\scriptstyle i}\mkern6mu} f \vee (\rhd f \wedge g) \equiv {\Diamond\mkern-12mu{\scriptstyle i}\mkern6mu} f \tag{C.239}$$

FstDiAndDiEqv [*DRS*]

$$\vdash \ \rhd({\Diamond\mkern-12mu{\scriptstyle i}\mkern6mu} f \wedge {\Diamond\mkern-12mu{\scriptstyle i}\mkern6mu} g) \equiv (\rhd f \wedge {\Diamond\mkern-12mu{\scriptstyle i}\mkern6mu} g) \vee (\rhd g \wedge {\Diamond\mkern-12mu{\scriptstyle i}\mkern6mu} f) \tag{C.240}$$

BiNotFstEqvBiNot [*DRS*]

$$\vdash \ \boxed{\scriptstyle i} \neg \rhd f \equiv \boxed{\scriptstyle i} \neg f \tag{C.241}$$

BsNotFstEqvBsNot [*DRS*]

$$\vdash \ \boxed{\scriptstyle s} \neg \rhd f \equiv \boxed{\scriptstyle s} \neg f \tag{C.242}$$

C.1.9.4 First with state formulae

FstState [*DRS*]

$$\vdash \ \rhd w \equiv \mathsf{empty} \wedge w \tag{C.243}$$

FstStateAndBsNotEmpty [*DRS*]

$$\vdash \ \rhd w \wedge \boxed{\scriptstyle s} \neg \ \mathsf{empty} \equiv \rhd w \tag{C.244}$$

FstStateImpFstStateOr [*DRS*]

$$\vdash \ \rhd w \supset \rhd(w \vee f) \tag{C.245}$$

HaltStateEqvFstFinState [*DRS*]

$$\vdash \ \mathsf{halt}\, w \equiv \rhd(\mathsf{fin}\, w) \tag{C.246}$$

HaltStateEqvFstHaltState [*DRS*]

$$\vdash \ \mathsf{halt}\, w \equiv \rhd(\mathsf{halt}\, w) \tag{C.247}$$

FstDiamondStateEqvHalt [*DRS*]

$$\vdash \ \rhd(\Diamond w) \equiv \mathsf{halt}\, w \tag{C.248}$$

FstBoxStateEqvStateAndEmpty [*DRS*]

$$\vdash \ \rhd(\Box w) \equiv w \wedge \mathsf{empty} \tag{C.249}$$

C.1.9.5 First and unique length

FstLenSame [*DRS*]

$$\vdash \ \diamondsuit (\rhd f \wedge \mathsf{len}(i)) \wedge \diamondsuit (\rhd f \wedge \mathsf{len}(j)) \supset i = j \tag{C.250}$$

DiImpExistsOneDiLenAndFst [*DRS*]

$$\vdash \ \diamondsuit f \supset \exists_1\, k \bullet \diamondsuit (\mathsf{len}(k) \wedge \rhd f) \tag{C.251}$$

C.1.9.6 First with chop distribution through conjunction

LFstAndDistr [DRS]

$$\vdash \ (\rhd f \wedge g_1) \ ; \ h_1 \wedge (\rhd f \wedge g_2) \ ; \ h_2 \equiv (\rhd f \wedge g_1 \wedge g_2) \ ; (h_1 \wedge h_2) \tag{C.252}$$

LFstAndDistrA [DRS]

$$\vdash \ (\rhd f \wedge g_1) \ ; \ h \wedge (\rhd f \wedge g_2) \ ; \ h \equiv (\rhd f \wedge g_1 \wedge g_2) \ ; \ h \tag{C.253}$$

LFstAndDistrB [DRS]

$$\vdash \ (\rhd f \wedge g) \ ; \ h_1 \wedge (\rhd f \wedge g) \ ; \ h_2 \equiv (\rhd f \wedge g) \ ; (h_1 \wedge h_2) \tag{C.254}$$

LFstAndDistrC [DRS]

$$\vdash \ \rhd f \ ; \ h_1 \wedge \rhd f \ ; \ h_2 \equiv \rhd f \ ; (h_1 \wedge h_2) \tag{C.255}$$

LFstAndDistrD [DRS]

$$\vdash \ \Diamond_i (\rhd f \wedge g_1) \wedge \Diamond_i (\rhd f \wedge g_2) \equiv \Diamond_i (\rhd f \wedge g_1 \wedge g_2) \tag{C.256}$$

C.1.9.7 Further useful theorems

FstEqvBsNotAndDi [DRS]

$$\vdash \ \rhd f \equiv \boxed{s} \neg f \wedge \Diamond_i f \tag{C.257}$$

NotFstChop [*DRS*]

$$\vdash \ \neg\,(\rhd f \ ; \ g) \equiv \neg\, \Diamond_{i}\, \rhd f \vee \rhd f \ ; \ \neg\, g \tag{C.258}$$

BsNotFstChop [*DRS*]

$$\vdash \ \boxed{s}\, \neg\,(\rhd f \ ; \ g) \equiv \mathsf{empty} \vee \neg\, \Diamond_{i}\, \rhd f \vee \rhd f \ ; \ \boxed{s}\, \neg\, g \tag{C.259}$$

FstFstChopEqvFstChopFst [*DRS*]

$$\vdash \ \rhd(\rhd f \ ; \ g) \equiv \rhd f \ ; \ \rhd g \tag{C.260}$$

FstFixFst [*DRS*]

$$\vdash \ \rhd \rhd f \equiv \rhd f \tag{C.261}$$

DsImpNotFst [*DRS*]

$$\vdash \ \Diamond_{s}\, f \supset (\neg\, \rhd f) \tag{C.262}$$

FstLenAndEqvLenAnd [*DRS*]

$$\vdash \ \rhd(\mathsf{len}(k) \wedge f) \equiv \mathsf{len}(k) \wedge f \tag{C.263}$$

FstAndElimL [*DRS*]

$$\vdash \ \rhd f \supset f \tag{C.264}$$

FstImpNotDiChopSkip [*DRS*]

$$\vdash \quad \triangleright f \supset \neg\, (\Diamond\mkern-11mu{\scriptstyle i}\;\, f \;;\; \mathsf{skip}) \tag{C.265}$$

FstImpDiEqv [*DRS*]

$$\vdash \quad \triangleright f \supset (\Diamond\mkern-11mu{\scriptstyle i}\;\, f \equiv f) \tag{C.266}$$

FstAndDiFstAndEqvFstAnd [*DRS*]

$$\vdash \quad \triangleright f \wedge \Diamond\mkern-11mu{\scriptstyle i}\;\, (\triangleright f \wedge g) \equiv \triangleright f \wedge g \tag{C.267}$$

FstAndDiImpBsNotAndDi [*DRS*]

$$\vdash \quad (\triangleright f \wedge \Diamond\mkern-11mu{\scriptstyle i}\;\, g) \supset (\boxed{\scriptstyle s}\, \neg\, (\Diamond\mkern-11mu{\scriptstyle i}\;\, f \wedge g)) \tag{C.268}$$

FstFstOrEqvFstOrL [*DRS*]

$$\vdash \quad \triangleright(\triangleright f \vee g) \equiv \triangleright(f \vee g) \tag{C.269}$$

FstFstOrEqvFstOrR [*DRS*]

$$\vdash \quad \triangleright(f \vee \triangleright g) \equiv \triangleright(f \vee g) \tag{C.270}$$

FstFstOrEqvFstOr [*DRS*]

$$\vdash \quad \triangleright(\triangleright f \vee \triangleright g) \equiv \triangleright(f \vee g) \tag{C.271}$$

C.1.9.8 First with len and skip

FstLenEqvLen [*DRS*]

$$\vdash \ \triangleright \mathsf{len}(k) \equiv \mathsf{len}(k) \tag{C.272}$$

FstSkip [*DRS*]

$$\vdash \ \triangleright \mathsf{skip} \equiv \mathsf{skip} \tag{C.273}$$

FstLenEqvLenFst [*DRS*]

$$FstLenEqvLenFst \tag{C.274}$$

FstNextEqvNextFst [*DRS*]

$$FstNextEqvNextFst \tag{C.275}$$

C.1.9.9 First occurrence with iteration

FstCSEqvEmpty [*DRS*]

$$\vdash \ \triangleright(f^*) \equiv \mathsf{empty} \tag{C.276}$$

FstIterFixFst [*DRS*]

$$\vdash \ (\triangleright f)^n \equiv \triangleright((\triangleright f)^n), \quad [n \geq 0] \tag{C.277}$$

C.1.9.10 Dual of first

NFstDef [DRS]

$$\trianglerighteq f \quad \widehat{=} \quad \neg \triangleright \neg f \tag{C.278}$$

NFstEqvOrDsNot [DRS]

$$\vdash \quad \trianglerighteq f \equiv f \vee \diamondsuit_{s} \neg f \tag{C.279}$$

NotFstEqvNFstNot [DRS]

$$\vdash \quad \neg \triangleright f \equiv \trianglerighteq \neg f \tag{C.280}$$

NotNFstEqvFstNot [DRS]

$$\vdash \quad \neg \trianglerighteq f \equiv \triangleright \neg f \tag{C.281}$$

C.1.9.11 Reflection of the first occurrence operator

LstDef [DRS]

$$\triangleleft f \quad \widehat{=} \quad f \wedge \boxed{t} \neg f \tag{C.282}$$

NLstDef [DRS]

$$\trianglelefteq f \quad \widehat{=} \quad \neg \triangleleft \neg f \tag{C.283}$$

FstrEqvLstr [*DRS*]

$$\vdash \;(\rhd f)^r \equiv \lhd f^r \tag{C.284}$$

LstrEqvFstr [*DRS*]

$$\vdash \;(\lhd f)^r \equiv \rhd f^r \tag{C.285}$$

FstChopFstREqvLstrChopLstr [*DRS*]

$$\vdash \;(\rhd f \;;\; \rhd g)^r \equiv \lhd g^r \;;\; \lhd f^r \tag{C.286}$$

FstFstChoprEqvLstrChopLstr [*DRS*]

$$\vdash \;(\rhd(\rhd f \;;\; g))^r \equiv \lhd(g^r \;;\; \lhd f^r) \tag{C.287}$$

LstChopLstEqvLstChopLst [*DRS*]

$$\vdash \;\lhd(g \;;\; \lhd f) \equiv \lhd g \;;\; \lhd f \tag{C.288}$$

C.2 ITL Monitor definitions, combinators and laws

C.2.1 ITL Monitor definitions

MFirstDef [*DRS*]

$$\mathcal{M}(\textsf{\textbf{FIRST}}\,(f)) \quad \widehat{=} \quad \rhd f \tag{C.289}$$

MUptoDef [*DRS*]

$$\mathcal{M}(a\ \textsf{\textbf{UPTO}}\ b) \ \widehat{=}\ \rhd(\mathcal{M}(a) \vee \mathcal{M}(b)) \tag{C.290}$$

MThruDef [*CAU*]

$$\mathcal{M}(a\ \textsf{\textbf{THRU}}\ b) \ \widehat{=}\ \rhd(\diamondsuit\mathcal{M}(a) \wedge \diamondsuit\mathcal{M}(b)) \tag{C.291}$$

MThenDef [*DRS*]

$$\mathcal{M}(a\ \textsf{\textbf{THEN}}\ b) \ \widehat{=}\ \mathcal{M}(a)\ ;\ \mathcal{M}(b) \tag{C.292}$$

MWithDef [*DRS*]

$$\mathcal{M}(a\ \textsf{\textbf{WITH}}\ f) \ \widehat{=}\ \mathcal{M}(a) \wedge f \tag{C.293}$$

C.2.2 ITL Monitor derived definitions

MHaltDef [*DRS*]

$$\textsf{\textbf{HALT}}\,(w) \ \widehat{=}\ \textsf{\textbf{FIRST}}\,(\mathsf{fin}\ w) \tag{C.294}$$

MLenDef [*DRS*]

$$\textsf{\textbf{LEN}}\,(k) \ \widehat{=}\ \textsf{\textbf{FIRST}}\,(\mathsf{len}(k)) \tag{C.295}$$

MEmptyDef [*DRS*]

$$\textsf{\textbf{EMPTY}} \ \widehat{=}\ \textsf{\textbf{FIRST}}\,(\mathsf{empty}) \tag{C.296}$$

MSkipDef [*DRS*]

$$\textsf{\textbf{SKIP}} \quad \widehat{=} \quad \textsf{\textbf{FIRST}}\,(\textsf{skip}) \tag{C.297}$$

MGuardDef [*DRS*]

$$\textsf{\textbf{GUARD}}\,(w) \quad \widehat{=} \quad \textsf{\textbf{EMPTY WITH}}\ w \tag{C.298}$$

MTimesDef [*DRS*]

$$\begin{array}{lcl} a\ \textsf{\textbf{TIMES}}\ 0 & \widehat{=} & \textsf{\textbf{EMPTY}} \\ a\ \textsf{\textbf{TIMES}}\ (k+1) & \widehat{=} & a\ \textsf{\textbf{THEN}}\ (a\ \textsf{\textbf{TIMES}}\ k), \quad k \geq 0 \end{array} \tag{C.299}$$

MFailDef [*DRS*]

$$\textsf{\textbf{FAIL}} \quad \widehat{=} \quad \textsf{\textbf{FIRST}}\,(\textsf{false}) \tag{C.300}$$

MAlwaysDef [*DRS*]

$$a\ \textsf{\textbf{ALWAYS}}\ w \quad \widehat{=} \quad a\ \textsf{\textbf{WITH}}\ (\boxed{\scriptstyle i}\ \textsf{fin}\ w) \tag{C.301}$$

MSometimeDef [*DRS*]

$$a\ \textsf{\textbf{SOMETIME}}\ w \quad \widehat{=} \quad a\ \textsf{\textbf{WITH}}\ (\Diamond_{i}\ \textsf{fin}\ w) \tag{C.302}$$

MUntilDef [*DRS*]

$$w_1\ \textsf{\textbf{UNTIL}}\ w_2 \quad \widehat{=} \quad (\textsf{\textbf{HALT}}\ w_2)\ \textsf{\textbf{WITH}}\ (\boxed{\scriptstyle m}\ \ w_1) \tag{C.303}$$

MWithinDef [*DRS*]

$$a \textsf{ WITHIN } f \;\widehat{=}\; a \textsf{ WITH } (\boxed{s} \neg f) \tag{C.304}$$

MAndDef [*DRS*]

$$a \textsf{ AND } b \;\widehat{=}\; a \textsf{ WITH } \mathcal{M}(b) \tag{C.305}$$

MIterateDef [*DRS*]

$$a \textsf{ ITERATE } b \;\widehat{=}\; a \textsf{ WITH } (\mathcal{M}(b))^* \tag{C.306}$$

MStarDef [*DRS*]

$$a \textsf{ STAR } f \;\widehat{=}\; \textsf{FIRST } (\Diamond f) \textsf{ ITERATE } a \tag{C.307}$$

MRepeatDef [*DRS*]

$$a \textsf{ REPEATUNTIL } w \;\widehat{=}\; (\textsf{HALT } w) \textsf{ ITERATE } (a \textsf{ WITH } (\boxed{m} \neg w)) \tag{C.308}$$

C.2.3 ITL Monitor laws

MFixFst [*DRS*]

$$\vdash \mathcal{M}(a) \equiv \triangleright \mathcal{M}(a) \tag{C.309}$$

MGuardFalseEqvFalse [*DRS*]

$$\vdash \mathcal{M}(\textsf{GUARD } (\mathsf{false})) \equiv \mathsf{false} \tag{C.310}$$

MFstFalseEqvFalse [*DRS*]

$$\vdash \ \mathcal{M}(\textbf{FIRST}\,(\mathsf{false})) \equiv \mathsf{false} \tag{C.311}$$

C.2.4 ITL Monitor alternative definitions

MFailAlt [*DRS*]

$$\vdash \ \mathcal{M}(\textbf{FAIL}) \equiv \mathsf{false} \tag{C.312}$$

MEmptyAlt [*DRS*]

$$\vdash \ \mathcal{M}(\textbf{EMPTY}) \equiv \mathsf{empty} \tag{C.313}$$

MSkipAlt [*DRS*]

$$\vdash \ \mathcal{M}(\textbf{SKIP}) \equiv \mathsf{skip} \tag{C.314}$$

MGuardAlt [*DRS*]

$$\vdash \ \mathcal{M}(\textbf{GUARD}\,(w)) \equiv \mathsf{empty} \wedge w \tag{C.315}$$

MLengthAlt [*DRS*]

$$\vdash \ \mathcal{M}(\textbf{LEN}\,k) \equiv \mathsf{len}(k) \tag{C.316}$$

MAlwaysAlt [*DRS*]

$$\vdash \ \mathcal{M}(a \ \textbf{ALWAYS} \ w) \equiv \mathcal{M}(a) \wedge \Box w \tag{C.317}$$

MSometimeAlt [*DRS*]

$$\vdash\ \mathcal{M}(a\ \textsf{\textbf{SOMETIME}}\ w) \equiv \mathcal{M}(a) \wedge \Diamond w \tag{C.318}$$

MWithinAlt [*DRS*]

$$\vdash\ \mathcal{M}(a\ \textsf{\textbf{WITHIN}}\ f) \equiv \mathcal{M}(a) \wedge \boxed{\mathsf{s}} \neg f \tag{C.319}$$

MTimesAlt [*DRS*]

$$\vdash\ \mathcal{M}(a\ \textsf{\textbf{TIMES}}\ k) \equiv (\mathcal{M}(a))^k \tag{C.320}$$

MUptoAlt [*DRS*]

$$\vdash\ \mathcal{M}(a\ \textsf{\textbf{UPTO}}\ b) \equiv (\mathcal{M}(a) \wedge \boxed{\mathsf{i}} \neg \mathcal{M}(b)) \vee (\mathcal{M}(b) \wedge \boxed{\mathsf{i}} \neg \mathcal{M}(a)) \vee (\mathcal{M}(a) \wedge \mathcal{M}(b)) \tag{C.321}$$

MThruAlt [*DRS*]

$$\vdash\ \mathcal{M}(a\ \textsf{\textbf{THRU}}\ b) \equiv (\mathcal{M}(a) \wedge \Diamond\hspace{-0.62em}\mathsf{i}\ \ \mathcal{M}(b)) \vee (\mathcal{M}(b) \wedge \Diamond\hspace{-0.62em}\mathsf{i}\ \ \mathcal{M}(a)) \tag{C.322}$$

MHaltAlt [*DRS*]

$$\vdash\ \mathcal{M}(\textsf{\textbf{HALT}}\ w) \equiv \textsf{halt}\ w \tag{C.323}$$

C.2.5 ITL Monitor equivalence

EqDef [*CAU*]

$$(a \simeq b) \equiv (\vdash\ \mathcal{M}(a) = \mathcal{M}(b)) \tag{C.324}$$

MonEqRefl $_{[CAU]}$

$$a \simeq a \tag{C.325}$$

MonEqSym $_{[CAU]}$

$$a \simeq b \vdash b \simeq a \tag{C.326}$$

MonEqTrans $_{[CAU]}$

$$a \simeq b,\ b \simeq c \vdash a \simeq c \tag{C.327}$$

MonEq $_{[CAU]}$

$$(a \simeq b) = (\vdash\ \mathcal{M}(a) = \mathcal{M}(b)) \tag{C.328}$$

C.2.6 Efficient implementation of FAIL

MFailEqvFstFalseWithinEmpty $_{[DRS]}$

$$\vdash\ \textbf{FAIL} \simeq \textbf{FIRST}\ (\text{false})\ \textbf{WITHIN}\ \text{empty} \tag{C.329}$$

C.2.7 ITL Monitor annihilator and identity laws

MFailUpto $_{[DRS]}$

$$\textbf{FAIL UPTO}\ a \simeq a \tag{C.330}$$

MFailThru $_{[DRS]}$

$$\textsf{\textbf{FAIL THRU}}\ a \simeq \textsf{\textbf{FAIL}} \tag{C.331}$$

MFailAnd $_{[DRS]}$

$$\textsf{\textbf{FAIL AND}}\ a \simeq \textsf{\textbf{FAIL}} \tag{C.332}$$

MThenFail $_{[DRS]}$

$$a\ \textsf{\textbf{THEN FAIL}} \simeq \textsf{\textbf{FAIL}} \tag{C.333}$$

MFailThen $_{[DRS]}$

$$\textsf{\textbf{FAIL THEN}}\ a \simeq \textsf{\textbf{FAIL}} \tag{C.334}$$

MFailWith $_{[DRS]}$

$$\textsf{\textbf{FAIL WITH}}\ f \simeq \textsf{\textbf{FAIL}} \tag{C.335}$$

MWithFalse $_{[DRS]}$

$$a\ \textsf{\textbf{WITH}}\ \textsf{false} \simeq \textsf{\textbf{FAIL}} \tag{C.336}$$

MWithTrue $_{[DRS]}$

$$a\ \textsf{\textbf{WITH}}\ \textsf{true} \simeq a \tag{C.337}$$

MEmptyUpto [*DRS*]

$$\textsf{\textbf{EMPTY UPTO}}\ a \simeq \textsf{\textbf{EMPTY}} \tag{C.338}$$

MEmptyThru [*DRS*]

$$\textsf{\textbf{EMPTY THRU}}\ a \simeq a \tag{C.339}$$

MThenEmpty [*DRS*]

$$a\ \textsf{\textbf{THEN EMPTY}} \simeq a \tag{C.340}$$

MEmptyThen [*DRS*]

$$\textsf{\textbf{EMPTY THEN}}\ a \simeq a \tag{C.341}$$

MEmptyIterate [*DRS*]

$$\textsf{\textbf{EMPTY ITERATE}}\ b \simeq \textsf{\textbf{EMPTY}} \tag{C.342}$$

C.2.8 ITL Monitor idempotence laws

MIterateIdemp [*DRS*]

$$a\ \textsf{\textbf{ITERATE}}\ a \simeq a \tag{C.343}$$

MUptoIdemp [*DRS*]

$$a\ \textsf{\textbf{UPTO}}\ a \simeq a \tag{C.344}$$

MThruIdemp [*DRS*]

$$a \textsf{ THRU } a \simeq a \tag{C.345}$$

MAndIdemp [*DRS*]

$$a \textsf{ AND } a \simeq a \tag{C.346}$$

MWithIdemp [*DRS*]

$$(\textsf{WITH } f) \circ (\textsf{WITH } f) \simeq (\textsf{WITH } f) \tag{C.347}$$

C.2.9 ITL Monitor commutativity laws

MUptoCommut [*DRS*]

$$a \textsf{ UPTO } b \simeq b \textsf{ UPTO } a \tag{C.348}$$

MThruCommut [*DRS*]

$$a \textsf{ THRU } b \simeq b \textsf{ THRU } a \tag{C.349}$$

MAndCommut [*DRS*]

$$a \textsf{ AND } b \simeq b \textsf{ AND } a \tag{C.350}$$

MWithCommut [*DRS*]

$$(\textsf{WITH } f) \circ (\textsf{WITH } g) \simeq (\textsf{WITH } g) \circ (\textsf{WITH } f) \tag{C.351}$$

C.2.10 ITL Monitor associativity laws

MUptoAssoc [DRS]

$$(a \ \textsf{\textbf{UPTO}} \ b) \ \textsf{\textbf{UPTO}} \ c \simeq a \ \textsf{\textbf{UPTO}} \ (b \ \textsf{\textbf{UPTO}} \ c) \tag{C.352}$$

MThruAssoc [DRS]

$$(a \ \textsf{\textbf{THRU}} \ b) \ \textsf{\textbf{THRU}} \ c \simeq a \ \textsf{\textbf{THRU}} \ (b \ \textsf{\textbf{THRU}} \ c) \tag{C.353}$$

MAndAssoc [DRS]

$$(a \ \textsf{\textbf{AND}} \ b) \ \textsf{\textbf{AND}} \ c \simeq a \ \textsf{\textbf{AND}} \ (b \ \textsf{\textbf{AND}} \ c) \tag{C.354}$$

MThenAssoc [DRS]

$$(a \ \textsf{\textbf{THEN}} \ b) \ \textsf{\textbf{THEN}} \ c \simeq a \ \textsf{\textbf{THEN}} \ (b \ \textsf{\textbf{THEN}} \ c) \tag{C.355}$$

C.2.11 ITL Monitor absorption laws

MWithAbsorp [DRS]

$$(\textsf{\textbf{WITH}} \ f) \circ (\textsf{\textbf{WITH}} \ g) \simeq (\textsf{\textbf{WITH}} \ (f \wedge g)) \tag{C.356}$$

MUptoThruAbsorp [DRS]

$$a \ \textsf{\textbf{UPTO}} \ (a \ \textsf{\textbf{THRU}} \ b) \simeq a \tag{C.357}$$

MThruUptoAbsorp [*DRS*]

$$a \textsf{ THRU } (a \textsf{ UPTO } b) \simeq a \tag{C.358}$$

C.2.12 ITL Monitor distributivity laws

MUptoThruDistrib [*DRS*]

$$a \textsf{ UPTO } (b \textsf{ THRU } c) \simeq (a \textsf{ UPTO } b) \textsf{ THRU } (a \textsf{ UPTO } c) \tag{C.359}$$

MUptoThruRDistrib [*DRS*]

$$(a \textsf{ UPTO } b) \textsf{ THRU } c \simeq (a \textsf{ THRU } c) \textsf{ UPTO } (b \textsf{ THRU } c) \tag{C.360}$$

MThruUptoDistrib [*DRS*]

$$a \textsf{ THRU } (b \textsf{ UPTO } c) \simeq (a \textsf{ THRU } b) \textsf{ UPTO } (a \textsf{ THRU } c) \tag{C.361}$$

MThruUptoRDistrib [*DRS*]

$$(a \textsf{ THRU } b) \textsf{ UPTO } c \simeq (a \textsf{ UPTO } c) \textsf{ THRU } (b \textsf{ UPTO } c) \tag{C.362}$$

MWithAndDistrib [*DRS*]

$$(a \textsf{ AND } b) \textsf{ WITH } f \simeq (a \textsf{ WITH } f) \textsf{ AND } (b \textsf{ WITH } f) \tag{C.363}$$

MThenAndDistrib [*DRS*]

$$a \textsf{ THEN } (b \textsf{ AND } c) \simeq (a \textsf{ THEN } b) \textsf{ AND } (a \textsf{ THEN } c) \tag{C.364}$$

MAndThenDistrib [*DRS*]

$$(a \textsf{ AND } b) \textsf{ THEN } c \simeq (a \textsf{ THEN } c) \textsf{ AND } (b \textsf{ THEN } c) \tag{C.365}$$

MThenUptoDistrib [*DRS*]

$$a \textsf{ THEN } (b \textsf{ UPTO } c) \simeq (a \textsf{ THEN } b) \textsf{ UPTO } (a \textsf{ THEN } c) \tag{C.366}$$

MThenThruDistrib [*DRS*]

$$a \textsf{ THEN } (b \textsf{ THRU } c) \simeq (a \textsf{ THEN } b) \textsf{ THRU } (a \textsf{ THEN } c) \tag{C.367}$$

MHaltWithAndDistrib [*DRS*]

$$((\textsf{HALT } w) \textsf{ WITH } f) \textsf{ AND } ((\textsf{HALT } w) \textsf{ WITH } g) \simeq (\textsf{HALT } w) \textsf{ WITH } (f \wedge g) \tag{C.368}$$

MHaltWithUptoHaltWithEqvHaltWithOr [*DRS*]

$$((\textsf{HALT } w) \textsf{ WITH } f) \textsf{ UPTO } ((\textsf{HALT } w) \textsf{ WITH } g) \simeq (\textsf{HALT } w) \textsf{ WITH } (f \vee g) \tag{C.369}$$

MHaltWithThruHaltWithEqvHaltWithAndHaltWith [*DRS*]

$$((\textsf{HALT } w) \textsf{ WITH } f) \textsf{ THRU } ((\textsf{HALT } w) \textsf{ WITH } g) \simeq ((\textsf{HALT } w) \textsf{ WITH } f) \textsf{ AND } ((\textsf{HALT } w) \textsf{ WITH } g) \tag{C.370}$$