Model-based runtime analysis of distributed reactive systems

Andreas Klaus Bauer

Institut für Informatik
der Technischen Universität München

Model-based runtime analysis of distributed reactive systems

Andreas Klaus Bauer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Alois Knoll

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h. c. Manfred Broy
2. Dr. Perdita Stevens, Reader,
   University of Edinburgh, UK

Die Dissertation wurde am 20.12.2006 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.04.2007 angenommen.

Abstract

As interactions and dependencies within distributed reactive systems increase, the problem of detecting failures which depend on the exact situation and environmental conditions they occur in grows. As a result, not only the detection of failures is increasingly difficult, but also the differentiation between the symptoms of a fault, and the actual fault itself, i. e., the cause of a failure.

This thesis proposes an efficient approach for the analysis of distributed reactive systems at runtime. It introduces a framework, referred to as monitoring-based runtime reflection framework, for the detection of failures as well as identification of their causes. As an interface to the framework, this thesis proposes a specification language, SALT, to express desired system properties, which then have to hold at all times while a system executes. Detection of failures in the framework is based upon monitoring systems with respect to their associated properties, defined in the high-level specification language SALT. SALT specifications are translatable into the real-time temporal logic TLTL as well as in the untimed temporal logic LTL, and as such can also be used with other temporal logic verification frameworks, such as model checkers. For both logics, an efficient monitor generation procedure is developed for either monitoring qualitative assertions about a system's behaviour in the untimed case, or quantitative assertions reflecting dense real-time constraints. In order to reflect the semantics of LTL, respectively TLTL, with respect to a finite observational trace in a suitable and unambiguous manner, a 3-valued interpretation is proposed. The corresponding 3-valued monitoring procedure is shown to be optimal with respect to the space-complexity of the generated monitors, and able to detect all minimal bad prefixes of non-conforming system behaviour. Based on the results of the monitors, a dedicated failure diagnosis is performed to identify possible explanations for an observed deviation. As such, diagnosis can either confirm that a monitor detected the root cause for a failure, or indicate that the fault is located elsewhere, and possibly outside the scope of the monitored system. In the runtime reflection framework, diagnosis is based upon the principles of first-order model-based diagnosis, but developed in the propositional domain. The propositional diagnosis problem is then shown to correspond to a deterministic implementation of a solution to the #SAT problem for which a computationally efficient realisation is introduced.

This thesis develops both the theoretical foundations for runtime reflection as well as efficient means for its implementation.

Acknowledgements

First and foremost, I want to express my gratitude to Manfred Broy for taking me on in his highly energetic research group as well as for the supervision of this thesis. Manfred Broy created a unique research and work environment for his group from which I and my ideas have greatly benefited over the last almost four years. Further, I want to thank Perdita Stevens from the University of Edinburgh for co-supervising this thesis. Her acceptance of this task was beyond any call of duty, and not always easy over a considerable spatial distance.

Martin Leucker must be pointed out as my academic role model over the past few years. He had major influence on my development as an academic and researcher.

Selfishly, I made various people read through and comment on drafts of this thesis. Timothy Bourke, Peter Braun, Gerwin Klein, Wendy Proctor, Michael Tautschnig, and Martin Wildmoser even took the time to read and critically comment on almost the entire document. I would like to sincerely thank them for their efforts.

Moreover, I am grateful for many stimulating and interesting discussions on topics of this thesis (or, at first sight, rather unrelated themes, which later turned out to be very influential) with Michael Beetz, Meir Manny Lehman, Reinhold Letz, Hans-Wolfgang Loidl, Roland Martin, Christian Schallhart, Bernhard Schätz, Anika Schumann, and Gernot Stenz. Their broad experience and technical insights have directly or indirectly shaped many different aspects of this thesis.

I also could not have done it without the moral support from various colleagues in our group, and friends I have found within. In particular, I would like to point out the members of the BASE.XT team as well as Johannes Grünbauer who has always been a pleasant office mate and a dear friend. Markus Pizka deserves a special mention without whom I would probably not even have ended up in Manfred Broy's Software & Systems Engineering group at all. He also supported me with counsel and encouragement when I needed it.

Finally, thanks are due to my colleagues and collaborators from industry with whom I had the pleasure of working with foremost in the research projects AutoMoDe and BASE.XT. In particular, I would like to thank Richard Bogenberger and Martin Wechs from the BMW Group for many pleasant and interesting discussions which, after all, centred not always around automotive software.

Andreas Bauer
München, November 2006

Chronology

Much of §3 is based upon material developed by Bauer et al. [2006c], and subsequently published at the *Eigth International Conference on Formal Engineering Methods* (ICFEM, see Bauer et al. [2006d]). The approach to runtime verification described in §4 is based upon results developed first by Arafat et al. [2005], and subsequently published at the *26th Conference on Foundations of Software Technology and Theoretical Computer Science* (FSTTCS, see Bauer et al. [2006b]). The approach to diagnosis as developed in §5 has been published in large parts at the *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems* (CPAIOR, see Bauer [2005]). Various aspects of §6 (as well as earlier chapters) are based upon a contribution to the *IEEE Australian Software Engineering Conference* (ASWEC, see Bauer et al. [2006a]). Further implementation aspects detailed on in §6 are accepted for publication at the 2007 Conference on *Design, Automation and Test in Europe* (DATE, see Bauer et al. [2007]).

Moreover, some minor results from papers for the *USENIX Annual Technical Conference* (see Bauer [2004]), the *2005 ACM Conference on Embedded Software* (EMSOFT, see Romberg and Bauer [2004]), and for the journal *Informatik – Forschung und Entwicklung* (see Bauer et al. [2005]) are used and referred to throughout this thesis.

Contents

Chapter 1

Introduction

> Because all models are wrong, we reject the notion that models can be validated in the dictionary definition sense of "establishing truthfulness", instead focusing on creating models that are useful, on the process of testing, on the ongoing comparison of the model against all data of all types, and on the continual iteration between experiments with the virtual world of the model and experiments in the real world.
>
> *(John D. Sterman,* Reflections on becoming a systems scientist*)*

Software systems are pervasive in all aspects of society. From everyday objects such as mobile phones, or automatic teller machines (ATMs) at banks to complex control systems found in modern cars and aircraft, a significant part of our daily life is mediated by software. For instance, in cars, there are multimedia and navigation systems as well as vehicle dynamics systems (cruise control, stability control, electronic brakes, etc.) to make driving easier, and ultimately safer. Also in the manufacturing industry, the automation, mechanisation and computerisation of processes means that human operators, more than ever, are at the mercy of the computer-controlled environment.

As the world around us has become increasingly complicated and dependent on computer-controlled technology, the nature of the failures that occur has also become more complex, and, in some cases, occurrence also more frequent. For instance, according to the *German Automobile Club* (ADAC [2005]) the majority of all recently registered automotive faults relate to the malfunction of electric and electronic systems. For comparison, only 9% of all faults directly involve the engine, whereas electric and electronic systems account for 36% of all recent faults, however, not differentiating any further between software and hardware faults.

Increased technological dependence has also led to various severe accidents in the past, such as plane crashes (cf. Rushby [1993], Neumann [1995], Sarsfield et al. [2000], Lebow et al. [2000], Knight [2002], or Greenwell and Knight [2003]), involving many people and great damage to property and environment. As a result, public awareness of such accidents has increased. In the aftermath of an accident, the public often demands to know what happened and detailed analysis reports are created. For instance, Ladkin [1997] notes that aircraft accidents are known to be the most carefully researched failures in all of engineering. Much of our knowledge about faults that lead to big accidents stem from such analyses.

However, it must also be pointed out that the increased use of electric and electronic systems in cars has, despite an often observed high failure rate, also led to increased road safety. According to the ADAC [2006], every year, there are roughly 20% fewer fatal road accidents than in the previous years.

In traditional engineering, failure is often correlated with a persistent failure state, i. e., something physically broke, and what is broken will stay broken (unless someone repairs or changes a component). With systems designed for complex behaviour this is no longer so. Such systems may exhibit unwanted behaviour or may fail, even though nothing is physically broken. A trivial but suitable illustration, originally stressed by Gerdsmeier et al. [1997], is putting money into a vending machine, obtaining the desired item, but failing to receive change. Each action is appropriate in and of itself, but giving no change is not appropriate *given* the sequence of actions and states before it.

When the machine is an isolated system, then these deviations can often be automatically detected and eventually traced back, often to their origins in the design of the system, be it software or hardware. But suppose a system is not isolated, and communicates with other such systems, as well as human operators, in a changing environment. It is then much less clear where faults can be detected and traced, not only to individual systems, to the operator, and to the environment, but also to the interactions between these components. Failure analysis becomes technically complex and intellectually difficult.

1.1 Runtime reflection at a glance

The runtime reflection framework as presented in this thesis performs a dynamic system analysis, i. e., reasons about the overall system status while it executes. As such, it is able to tell *when* a failure occurred and also *why*. This is achieved in the framework by providing methods for monitoring a system's behaviour at runtime as well as for diagnosing the result of this process. In the case where such a result is sufficiently detailed and unambiguous, it could be used subsequently to issue control commands back to the system, i. e., to perform a dynamic reconfiguration in order to reestablish a well-defined system state.

1.1.1 Architectural overview

The framework consists of different "layers", each serving a different purpose in the process of runtime reflection. Fig. 1.1 gives an overview of these layers as they are used for analysing a distributed reactive system at runtime.

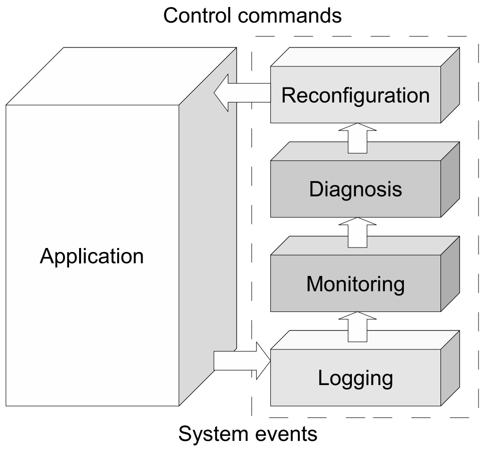


Fig. 1.1: Layered view on the runtime reflection framework.

The sole task of the logging layer is to translate native system events, into so-called *actions* that can be understood and processed by monitors. Thus, the monitoring layer consists of one or many monitors observing the stream of actions provided by the logger. By comparing the received actions with a reference behaviour, formally specified in terms of an assertion (or sometimes called an invariant), a monitor then indicates whether the system is currently in a well-defined state with respect to the assertion, or not. Any behavioural aberration detected by a monitor is then transmitted to the diagnosis layer for inferring the root causes.

Throughout this thesis, various references are made to concrete implementations of a logging layer. However, since logging as well as reconfiguration constitute highly domain-specific tasks, this work focusses on developing the monitoring and diagnosis layers whose formal foundations are specified in a domain-independent manner. In a setup where reconfiguration mechanisms exist, however, the diagnosis information can be used to return a system to a well-defined state, such as a fall-back "limp-home" mode known from other disciplines (cf. Deur et al. [2003]). If reconfiguration is not possible, the framework could be used for storing detailed log-files that not only register symptoms but also causes of failure. (For the technical details, see also §6.)

1.1.2 Practical view on runtime reflection

The runtime reflection framework depends on various different types of inputs. For instance, *at runtime*, the system's observable events pose the minimal prerequisite for the framework to function; that is, via the logging layer, one or many monitors continuously analyse the stream of system actions and determine whether or not the observations satisfy a set of predefined properties usually specified in terms of a *temporal logic formula* (see Fig. 1.2). Hence, in the *systems development stage*, the required inputs for the framework to be used are the *requirements* from which the monitors can be generated from.

Of particular interest are the so-called *functional requirements* which specify more or less exactly how a system should operate or behave. In practice, functional requirements are often supplemented by *nonfunctional requirements* which impose additional, but difficult to explicitly quantify or formalise constraints on a desired behaviour. For instance, "the system needs to execute as fast as possible" would be normally considered a non-functional requirement, whereas "a task should be executed every $5ms$" would impose a functional requirement on a system (cf. Maciaszek [2001]).

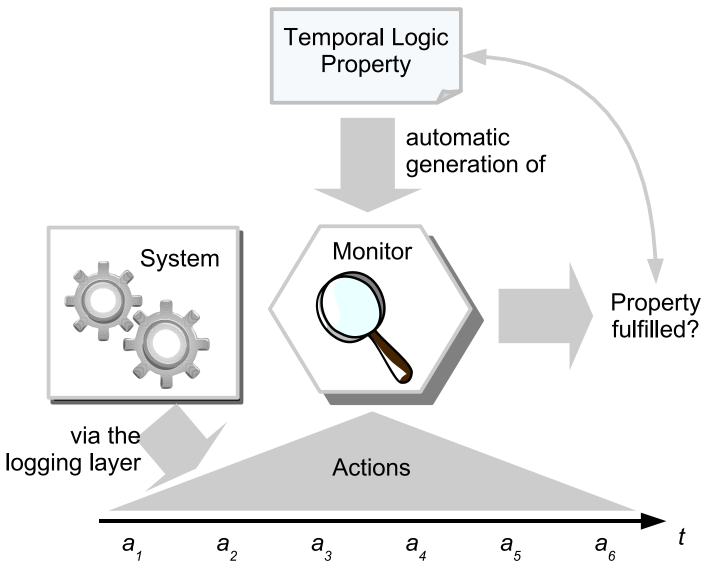


Fig. 1.2: Overview runtime verification.

Runtime reflection as presented in this thesis does *not* make it necessary for the developer to manually encode the monitors. However, the formal specification from which a monitor can be automatically generated from has to be derived from a set of (possibly informal) system requirements which describe a desired behaviour of the system. There exists no constraint on the types of properties that can be monitored as long as they are formally expressible in the temporal logics presented in this thesis (see §3.3 and §4). In other words, every requirement which can be written down precisely in terms of a temporal logic formula is suitable for the automatic monitor generation and subsequent system analysis as it is described in §4. How to actually obtain, in a stepwise process, a formal specification from an initial set of informal requirements, although practically challenging (see, e. g., Broy et al. [1993], Broy [1997], Maciaszek [2001] or Schätz et al. [2005]), is not the subject of this thesis. However, this process is technically supported in terms of a structured assertion language, SALT, which acts as a front-end to the framework and offers, as this thesis argues, convenient means for the formal capturing of functional or temporal requirements (see §3.3).

In the event of one or more monitors detecting violations of a property, the runtime reflection framework performs a system's diagnosis. In a nutshell, the diagnostic task consists of a comparison of the monitors' (negative) results with a reference system model where all monitors report a non-erroneous state. Basically, this reference model captures the overall causality of the system under scrutiny in terms of a dependence graph consisting of signals and communication paths between subcomponents of a system and its accompanying monitors (see §5). Since such dependencies are either implicitly defined by the actual code of a system, or explicitly captured in an abstract system model or a *communication matrix* (cf. Heinecke [2005], Köhl et al. [2005]), the required information for diagnosis can in many cases be extracted automatically with little or no human interaction. Examples for such automatic extractions include models of (embedded) *control systems*, which are often created using dedicated, graphical CASE-tools such as MATLAB/Simulink (Mathworks [2000]), for instance. For this particular tool set, Bauer et al. [2007] describe an approach to automatically extract abstract system models that are useful for diagnosis and formal analysis using the tools described in greater detail in §6. However, depending on the application and the types of systems to be analysed at runtime, the methods of model extraction may practically vary.

1.2 Detailed problem statement

From an academic point of view, there exist many methods that aid the construction of correct systems. For instance, the use of so-called *formal methods* aims primarily at cutting down on system errors (cf. Clarke et al. [1996]). Formal methods used in software engineering commonly include the precise, in a mathematical sense, specification of systems or properties, specification analysis and proof, transformational development, and various verification tasks, many of which are tool-supported (cf. Broy [1999]). Yet, the development of complex distributed reactive systems remains a potentially error-prone activity, since the use of formal methods cannot, in general, guarantee the non-occurrence of failures at runtime.

1.2.1 System failures and their inevitability

Formal verification is used to determine whether a system satisfies a specification. Ideally, this happens statically and automatically, without executing the system under scrutiny, such as it happens when checking program data types, for instance.

However, it is well-known that program and systems verification in general is an undecidable problem (Turing [1936]), and human interaction necessary to guide this process (unless specific abstractions are made, such as focusing merely on the type-correctness issue). Hence, for many real-world systems, we cannot eliminate all errors prior to deployment and real-world employment.

However, failures at runtime, especially in distributed reactive systems can happen for many different reasons and may not be addressed by static verification methods at all; for example, unforeseen side effects of an operation, or assumptions made about the operational environment of a system prove inadequate in the real-world or for real-world employment. System models, whether they are formal, semi-formal, or informal ones, used in the design and development stages of systems only reflect certain aspects that were anticipated and thus, specified, e. g., by a carefully undertaken *requirements engineering* process, forming the borderline between an informal system description and formal system development with specification and subsequent verification (cf. Broy et al. [1993]).

However, all system models reflecting design artifacts about systems and their environment are, up to an extent, incomplete by definition, since modelling always involves *abstraction* from certain aspects and truths. Thus, systematic modelling (and subsequent verification) helps to cut down on errors, but not at establishing system correctness as truthfulness in a dictionary-sense of the word. Sterman [2002] mused about this in his by-now frequently cited *Jay Wright Forrester Prize Lecture*. Therein, he emphasised that all models are *wrong* in a sense that they are incomplete with respect to the real-world and real-world influences. A similar observation was also made in an earlier study based on empirical investigation, and restated only recently by Lehman [2005], who identified underlying assumptions in system models as the driving force behind the evolution of systems, in that models have to be continuously revised and refined to better match reality (see also Lehman and Parr [1976]).

In *embedded systems*, which engage the physical environment, interacting directly and continuously with sensors and actuators, failures can be particularly subtle and highly dependent on the exact environmental conditions they occur in. Hence, they are difficult to anticipate and almost irreproducible under controlled conditions. As system interactions and dependencies even in small embedded systems increase, it becomes a costly and non-trivial challenge for manufacturers trying to analyse failure situations in detail, let alone doing so at runtime when the failure situation occurs.

Consistency and adequacy of (verification) models. Moreover, specifications can also be inconsistent in and of themselves, or may not reflect the *intention* of the person who created a specification or a model. However, system failures caused by that are not a direct result of the complex influences, say, the environment has on the later system, but rather of the inability to reflect such influences in an appropriate or unambiguous manner. It could also be argued that widely used modelling and specification languages such as the *Unified Modelling Language* (UML, Booch et al. [1998]) contribute to this problem by offering users on the one hand side a very high level of abstraction, but on the other no detailed and rigorous semantics (cf. France et al. [1998] and Amálio and Polack [2003]).[1]

1.2.2 To know when a failure occurred

In order to be able to react to unforeseen events or even failures that occur during a system's operation, it is essential that such events and failures are *detectable* in the first place.

The monitoring layer of the runtime reflection framework arranges for this detection, in that it continuously reasons about a system's behaviour at runtime based on observing the validity of a formal specification of *desired system properties over time*. This process is also known from the emerging scientific area of *runtime verification* (see §4).

For reasons outlined above, the formal specifications used in runtime verification are meant to complement existing behavioural specifications which are primarily used to build systems, but not necessarily to monitor them. As also outlined in the previous section, a main reason why runtime verification came into being is because the state-space of the systems under scrutiny is potentially infinitely large, and can often not be captured prior in a comprehensive system model for verification. For instance, the physical environment of systems can often only be approximated by complex, continuous differential equations (cf. Edwards et al. [1997]). In other situations, explicit system models are not even available, such as is the case when reasoning about *black* or *grey-box systems* (cf. Büchi and Weck [1999]).

What makes failure detection additionally difficult in practice is, aside from the inherent complexity of interconnected systems interacting continuously with and at the pace of their environment, that failures may not always be determined by just *one* specific deviation or event. As the simple example of the vending machine demonstrates, observed behavioural sequences are often considered correct under normal circumstances, but may lead to undesired effects in another, and probably less frequently encountered scenario.

Naturally, other, but not less frequently experienced reasons for failures in software systems include the actual breakage of underlying computer hardware, or other physical components the software depends upon, which typically results in unpredictable behavioural system aberrations and thus, unwanted overall system behaviour.

From a practical point of view, failure detection by means of runtime verification can be considered as a supplementation to other dynamic validation and verification

[1] Although it could also be argued that this is an important reason for UML's evident success, since its users are able to create models without having to think through all of its details, or having to worry about a rigorous semantics. However, for the sake of argument, this line of thought is not developed further at this point.

methods such as model-based testing (cf. Broy et al. [2005]), in that dedicated *monitor* components are executed in parallel with the system under scrutiny. Their task is to notice any aberrations of a predefined behaviour or invariant, such that an alarm or notification can be issued. As such, monitors detect *symptoms* of faults.

Reasoning with truncated paths

Most runtime verification approaches known from the literature are based on ideas derived from or used similarly in the area of *model checking* (cf. Clarke et al. [1999]) but adapted for use at runtime and for application over finite sequences of observed system behaviours. The monitors used in runtime verification base their verdict upon a behavioural sequence of actions, which is at most finite, i. e., from the initialisation phase of a system to the current instant of time. As such, the known runtime verification approaches (for a detailed overview see §4) also have various limitations, namely that the semantics of temporal logic which is used for specifying system properties and invariants is defined over infinite sequences of actions, whereas at runtime only finite prefixes of such infinite behaviours are available.

Detection of minimal bad prefixes

Additionally, the verdict of most monitoring procedures (as is the case with the commonly used "watchdogs" used in traditional engineering disciplines) is usually focused on the current instant of time, in that no failures are reported unless an explicitly marked erroneous state in the monitor is reached. However, when observing a *sequence* of system events, there may be situations where the verdict could be *predictive*; that is, the monitors could raise an alarm based on the observations so far, and even before an inevitable failure has actually happened. Current runtime verification approaches address this issue, only unsatisfactorily, if at all, in that additional means for trace evaluation are created, or in that they work only for a limited class of properties and systems, such as *quasi-synchronous* systems (see §4.5 for a more precise definition of this term).

Addressing real-time systems and requirements

However, execution of many critical systems does not adhere to an ideally synchronous model, but is spontaneous and event-triggered. For instance, embedded real-time systems impose strict deadlines on execution and communication time. Reasoning about such systems at runtime involves dealing with a different paradigm and, ultimately, different algorithms, models, and verification techniques when compared to common runtime verification techniques based on, e. g., *linear time temporal logic* (Pnueli [1977]), where real-time is not directly reflected.

1.2.3 To know why a failure occurred

When a failure or general behavioural aberration is detected in a distributed reactive system, a central question is whether or not a monitor noticed the *symptom* of a fault, or points to the actual part or component which is directly responsible for the observed deviation. Current approaches to runtime verification do not answer this question, focusing on the detection of "bugs", rather than the steps which could and should follow afterwards, such as checking the plausibility of a verdict.

The runtime reflection framework caters for this differentiation in terms of its diagnosis layer. The diagnosis layer provides an efficient realisation of *model-based diagnosis* (see §5), originally referred to as consistency-based diagnosis due to its formal foundations in consistency checking of first-order logic theories. It provides deductive methods to differentiate between the symptoms of a fault in a distributed system and the root causes.

Basically, model-based diagnosis provides methods to reason about a distributed, or rather a component-based system using a structural and also behavioural system model as well as a finite set of observations for the system. The method aims at isolating (all) possible causes for an observed failure, i. e., so-called *diagnoses*, that explain the root causes of failure. A diagnosis is then defined with respect to a set of faulty system components.

Since the term model-based diagnosis was first coined by Reiter [1987], much diagnosis-related research has been undertaken in many different disciplines. Consequently the term does not necessarily relate back to first-order consistency-based diagnosis anymore, but nowadays also to the use of neural networks, and fuzzy or probabilistic models, such as realised by Bayesian networks, to name just a few common examples[2].

Computational complexity of model-based diagnosis

Diagnostic problem solving is formalised as a method for finding the source of inconsistency in the logical (first-order) description of the normal functioning of a system when supplied with observed findings, where some of the findings are a direct consequence of a system defect or failure in reality. For a system consisting of n diagnosable components, there exist (at least) 2^n possible diagnoses, describing the current system state with respect to the number and distribution of faults in the system. Some approaches to diagnosis additionally cater for the fact that, in reality, not everything relevant about a system may always be observable; that is, so-called *unknowns* are taken into account. Basically, an unknown corresponds in the logical underpinning of model-based diagnosis to a variable whose value cannot be truthfully determined.

[2] From this point onward, the terms model-based and consistency-based diagnosis are used interchangeably, referring to logic-based methods for solving the diagnosis problem.

Moreover, the introduction of unknowns results in an additional "blowup" of the conflict search space; thus, making the proposed methods often unwieldy in practice, and inconclusive as a result, e. g., given m unknowns and n components, $2^{(n+m)}$ different possibilities have to be considered.

Application to dynamic systems

Although model-based diagnosis has been successfully applied even in mission-critical and industrial environments, its prime application remains the analysis of static systems, such as hardware systems (cf. Mikaelian et al. [2005]). There are mainly two reasons for this: 1. reasoning based on the underlying diagnosis models in first-order logic must either be guided by humans (i. e., interactive theorem proving), or, if automated, can be a computationally complex and possibly non-terminating task; 2. the logical models used in model-based diagnosis are not suited for adequately expressing real-time requirements and entire behavioural sequences which are needed for reasoning about many present-day reactive systems[3]. Moreover, continuous diagnostic reasoning about a system that may in fact never show a failure is also a computationally expensive undertaking, in particular, in strictly resource-bounded environments.

1.3 Contributions of this thesis

This section highlights specific contributions of this thesis divided into contributions to the area of runtime verification, model-based diagnosis, and property specification using temporal logic. Moreover, the feasibility as well as limitations of the runtime reflection approach to systems analysis are briefly discussed.

Contributions to runtime verification

The approach to runtime verification developed in this thesis covers both untimed and real-time systems. It is based upon a *3-valued interpretation* of finite behavioural traces resembling a system's observable actions over time. The 3-valued interpretation remedies the problems that currently exist in the dynamic interpretation of temporal specifications over finite traces: at the end of a trace, i. e., on the last observation, it is not possible to determine future obligations in an unambiguous way to (1) satisfy the standard semantics of temporal logic as well as take into account (2) the unpredictable future behaviour of the system. This problem is solved by introducing, besides *true* and *false*, a third truth value, namely *inconclusive*, to denote that a specification

[3] At this stage, the possibility of expressing certain temporal logic formulae using first-order logic over words as suggested first by Kamp [1968] is deliberately disregarded. It is considered as a possibility of merely theoretical value.

could not (yet) be verified or respectively falsified with respect to a prefix of a possibly infinite behavioural extension.

Furthermore, an efficient method is developed that, given a specified property in LTL, constructs a monitor component which observes a system, and whose output corresponds to the three truth values with respect to the finite behaviour observed so far.

This construction allows detection of all *minimal* bad prefixes that predict an eventual but unavoidable violation of a system property as early as possible. Note that predictive analyses also play an increasingly important rôle in industry, e. g., in the automotive sector, where the aim is to predict critical failures before their occurrence. In the automotive sector, this is commonly referred to as "preventive diagnosis" (cf. Varchmin [2005]).

The overall approach to monitoring is first developed in the untimed setting using standard LTL as a formalism, and then "lifted" to the timed setting, where a linear real-time logic—TLTL, short for timed LTL—is used. In the untimed setting, a translation into executable, extended Moore machines is given, whereas in the timed setting, this thesis develops a monitor generation algorithm based on the notion of *event-clock automata* which have been introduced formally by Alur et al. [1999]. The real-time case constitutes a change of paradigm, in that the monitors are no longer observing merely quasi-synchronous system actions in a stepwise manner, but rather patterns of real-time events; that is, each observed action is associated with a timestamp $t \in \mathbb{R}^{\geq 0}$ denoting when exactly the event was observed, or has occurred. As a consequence, the established verification methods from LTL, e. g., generation of *Büchi automata* (Büchi [1962]) for LTL properties, do not transfer to this setting in a straightforward manner, and need to be changed substantially when considering the real-time case in a dense-time domain.

Contributions to model-based diagnosis

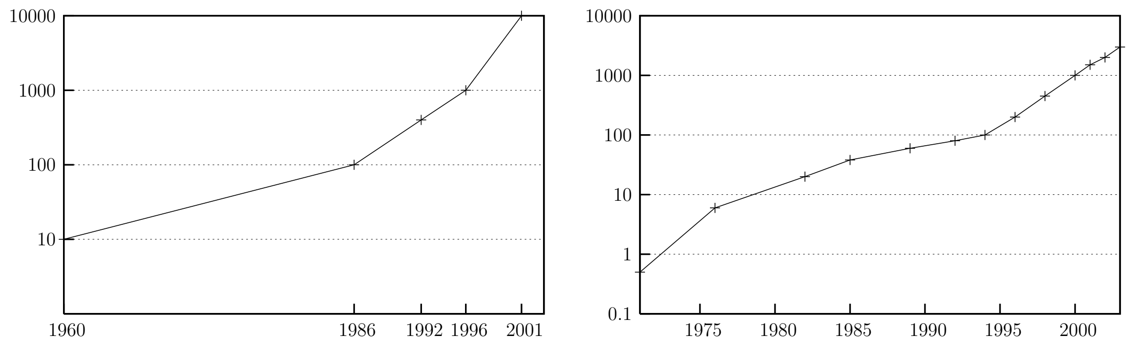
As noted by Mikaelian et al. [2005], model-based diagnosis has been applied, foremost, to the analysis of hardware or mechanical systems. Software systems exposing complex behavioural patterns, exchanging and accessing information concurrently, and operating in resource-bounded environments, such as embedded systems, have not been the focus of this method. Complexity and undecidability issues in first-order reasoning impose strict constraints on what can be achieved by model-based diagnosis, whose system models are general first-order sentences.

This thesis circumvents some of these limitations by reducing the original diagnosis problem to propositional logic, based on the verdicts of the monitors as observations. As such, runtime reflection does pay respect to dynamic behavioural patterns, and allows for the localisation of faults when aberrations are detected. The determination of diagnoses is then mapped to a problem whose technical solution has experienced dramatic improvements over the past few years, namely k-satisfiability, in short k-

SAT, using state-of-the-art SAT-solvers. Due to Cook [1971], the general or k-SAT-problem was the first known $\mathcal{NP}$-complete problem, for which very efficient solving algorithms now exist (cf. Davis and Putnam [1960], Davis et al. [1962] as well as Zhang [1997], Moskewicz et al. [2001]).

Modern SAT-solvers, such as the one developed in this thesis, often solve even difficult instances of a SAT-problem, involving many thousands of variables, within seconds. The mapping of the diagnostic problem to Boolean satisfiability is thus shown to be an efficient means for the implementation of an engine for model-based diagnosis.

Example (Moore vs. SAT). The feasibility of this particular approach can be illustrated using Fig. 1.3, which shows (a) the advances achieved in SAT-solving in recent years, compared to (b) the advances in processor speed. *Moore's Law* (Moore [1965]) adds roughly a factor 100 to the clock speed since 1990, while SAT-solving experienced factor of roughly 10^{30}. The left curve, which depicts the problem size in terms of the number of Boolean variables, is similar to the right one depicting Moore's Law with respect to clock speed in MHz. However, the increase on the left is, effectively, far greater than on the right, since not only the number of variables grows exponentially, but also the overall search space of the satisfiability problem spanned by the total number of variables.



(a) Advances in SAT-solvers. (b) Advances in processors.

Fig. 1.3: Technology advances. (Sources: `http://www.satcompetition.org/` and `http://www.intel.com/`)

The variables in the propositional diagnosis model correspond to (1) the respective components of a system that may fail (whether these are hardware or software components), (2) their causal relationships, and (3) to the verdict of the attached monitors or, if not available, unknowns. From a practical point of view, this resembles an *abstract* failure model, which constitutes the foundation for inferring the *concrete* diagnoses. Hence, in the context of the runtime reflection framework a custom, optimised Boolean solver is developed, LSAT, which is tailored for model-based diagnosis.

In diagnosis, however, one is not only interested as to whether a given problem has *a* solution, i. e., the SAT-problem, but also in the overall number and types of assignments, i. e., all the diagnoses, and, more importantly, the minimal sets of these.

Instead of determining the minimal sets of *all* possible (and possibly irrelevant) diagnoses, only those diagnoses are computed in the chosen approach, which contain at most n faulty components, where $n \in \mathbb{N}$ is a value that can be arbitrarily fixed by the user of the system. This is referred to as the n-fault assumption, which constitutes an effective pruning criterion of the unfolded problem search space representing all the supersets of possible diagnoses. Setting $n = \infty$, the problem corresponds to the at least equally complex #SAT problem which is in the complexity class $\#\mathcal{P}$ (cf. Papadimitriou [1994]). LSAT addresses this problem with only linear space requirements in its algorithm, and a custom data structure to facilitate a deterministic computation of the #SAT and the diagnosis problem using the n-fault assumption.

Facilitating property specification with temporal logic

SALT, the structured assertion language for temporal logic, acts as a high-level and intuitive-to-use interface to the runtime reflection framework (see §3.3). It allows the specification of custom, temporal system properties in a programming-language-like manner while still being fully translatable into timed temporal logic (TLTL), and untimed temporal logic (LTL). Which of the targets is used, is determined solely by the operators occurring in a SALT specification.

SALT offers many constructs not present in either temporal logic formalism, such as scope operators, exceptions, macro definitions, or a subset of regular expressions. While the dynamic analysis techniques described above aim at cutting down runtime failures, SALT intends to reduce specification errors by providing intuitive, human-readable, and abstract means for systems specification.

This thesis describes the language features, its expressiveness, the translation into temporal logics with rigorous semantics, as well as results from an optimising SALT compiler that accepts a SALT specification and produces TLTL or LTL specifications. The latter can also be used for formal verification frameworks other than runtime reflection, such as model checkers, for instance. The experimental results from the compiler presented in this thesis, suggest that the translation of a SALT specification to temporal logic adds little or no redundancy to the specification, when compared to a manually constructed formula expressing the same property. In many cases, the SALT specification even results in more compact formulae (see §6).

Feasibility and limitations

A schematic functional setup of the overall runtime reflection framework is provided in Fig. 1.4. The four system components monitored, $C_1, \ldots, C_4$, are represented in squares, whereas the monitors, $M_1, \ldots, M_4$, depicted as hexagons, each give feedback to the centralised model-based diagnosis engine, when a failure occurs. Each monitor verifies specific properties regarding the input and output channels, denoted i and o, of adjacent components. The diagnosis engine then infers whether an alarm raised by a monitor M_i corresponds to a faulty component C_i, or whether the fault is located in a remote component $C_{j \neq i}$ (whose accompanying monitor, if present, may have missed an aberration).

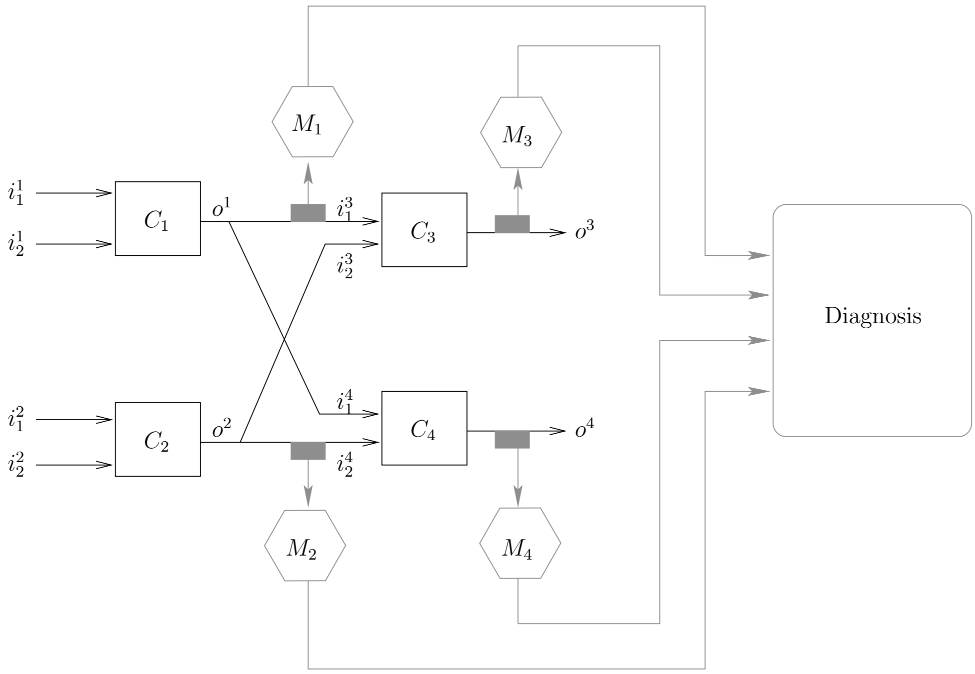


Fig. 1.4: Functional view on the runtime reflection framework.

Failures *detectable* by the runtime reflection framework, are generally those which can be noticed by the monitors; that is, local behavioural aberrations of components that result in a violation of a specified property or system invariant. For example, if aub and afb are two behaviours, where f represents a faulty and u the desired behaviour of a component, a directly adjacent monitor cannot differentiate the two, if u (respectively, f) is an internal event not communicated to another component, or directly to the monitor. However, given the detection of an aberration or a symptom, more complex types of failure are *diagnosable*, e. g., distributed or causal failures which may be due to single aberrations that have gone unnoticed by directly adjacent monitors. For example, if the corresponding monitor in the present example misses the faulty behaviour f, subsequent failures which are caused by f may be noticed by other, remote monitors in (physically) different parts of the system, and will thus lead to a diagnosis that locates the actual fault in the system.

As such, the proposed approach covers a wide range of failures that may occur in distributed reactive systems, but it does not cater for the analysis of the underlying program logic that may have lead to a failure within components, e. g., division by zero, stack overflow, null-pointer dereference, etc. Moreover, it cannot differentiate whether a failure is due to environmental influences or whether it relates to, say, early design errors. However, a more detailed discussion regarding the nature and types of failures in distributed reactive systems is given in §2.

Also, with application domains for distributed reactive systems on the rise, such as avionics or automotive, the issue of analysing such systems at runtime will become or already is an important factor not only for safety reasons, but also as a distinguishing and marketable feature between different manufacturers of systems. In the aforementioned domains, however, already a large number of systems are supplied by third-party companies who do not necessarily provide documentation on a system's internals or the source code; that is, manufacturers already integrate a big proportion of black and grey-box systems into their products. Hence, dynamic systems analysis not only needs to provide methods for detection and diagnosis of failures, but also a means for reflecting the needs of analysing black and grey-box systems, without actively influencing their actual behaviour. Thus, the proposed runtime reflection approach caters for transparent integration of monitors that do not necessarily need insight into the systems under scrutiny, as long as system events are technically observable, e. g., by means of a (standardised) middleware (cf. Heinecke et al. [2004]), or other low level logging facilities (cf. Gunter et al. [2002]).

1.4 Results of this thesis

Due to the results of this thesis, many of the problems sketched in §1.2 can be circumvented. In summary, those particular results as developed in this thesis are, (almost) in chronological order:

- A formal foundation for the analysis of distributed reactive (and real-time) systems at runtime.
- A methodological differentiation between the detection and diagnosis of failures.
- A high-level, intuitive-to-use specification language for temporal logic that facilitates the expression of untimed and timed system properties in a domain-independent manner.
- A 3-valued semantics for untimed and timed temporal logic (called LTL_3 and TLTL_3, respectively) for dynamically reasoning about systems using finite behavioural traces.
- Two constructions of *optimal-size* monitor components from a given formula φ, which can either be in LTL_3 or TLTL_3, and which allows the detection of *minimal* bad prefixes that violate φ.
- An efficient diagnosis engine based on propositional logic for the localisation of faults from the observation of symptoms (that are given by the monitors).
- An efficient solution to the #SAT problem whose deterministic realisation is

shown to correspond to the model-based diagnosis problem in the propositional domain.

- Comprehensive (and freely available) tool-support of the runtime reflection framework, in terms of an optimising compiler for Salt, a code generator for monitors, an efficient diagnosis engine, and several use-cases demonstrating the capabilities and use of the tool-chain.

1.5 A brief guided tour through this thesis

This section provides a brief summary of the subsequent chapters of this thesis.

Chapter 2—Failures and faults in distributed reactive systems. This chapter outlines the background of this thesis in more detail. It introduces the notion of distributed reactive and real-time systems, and sketches problems in their design and development as well as operation. Also, some basic terminology regarding common terms such as failure, fault, error, and symptom is given, and described how these concepts relate to one another, and are used throughout this thesis.

Chapter 3—Formal systems specification and verification with temporal logic. In this chapter, first, some of the theoretical foundations for runtime verification are discussed; that is, it details on the syntax and semantics of LTL, the classification of properties into safety and liveness, and different types of automata as means of verifying system models with respect to temporal properties. Then, some of these concepts are extended, in that the custom specification and assertion language for temporal logic, Salt, is presented whose core semantics corresponds exactly to LTL, and in the timed setting, to TLTL.

Chapter 4—Failure detection through runtime verification. In this chapter, an approach for runtime verification of reactive and real-time systems is developed. After examining some existing approaches, LTL_3 is introduced, a 3-valued semantics for LTL, and a dynamic automata-based decision procedure is given. Then the approach is discussed in the timed setting, in that $TLTL_3$ is introduced, a 3-valued variant of TLTL. Afterwards, a decision procedure for $TLTL_3$ is presented, which is based on symbolic runs over event-clock automata.

Chapter 5—Fault detection using model-based diagnosis. Chapter 5 describes how, given an observed symptom for failure, to differentiate the symptom from an actual cause of failure, i. e., from actual system faults. First, the formal foundations of the diagnosis problem are introduced and then an efficient solution for consistency-based diagnosis developed. The approach is tailored for runtime analysis of distributed component-based systems, and is based upon a mapping of diagnosis to a deterministic implementation of a solution to the #SAT problem.

Chapter 6—Implementation, tool-support, and comparative results. This chapter presents tool-support, and summarises some experimental results of the techniques used in the runtime reflection framework. In particular the technical realisations, employed data structures, and algorithms are reviewed, and some objective benchmarks for comparison given where feasible. Further, some example use-cases are sketched and a comprehensive runtime verification case-study for C++ applications discussed.

Chapter 7—Conclusions. Conclusions of this work as well as potential for further research are provided in the final chapter of this thesis.

Appendix A—SALT translation schema. In Appendix A, a detailed translation scheme for SALT is given. It thereby reflects the formal semantics of the language in terms of a translation to LTL as well as TLTL.

Appendix B—Runtime reflection on the web: obtaining the files. In Appendix B, some pointers are given where the runtime reflection framework as presented in this thesis can be obtained from. The tools realising the runtime reflection framework as well as the SALT compiler are open source software.

Some important keywords of this thesis are collected in the Index starting on p. 207.

Chapter 2

Failures and faults in distributed reactive systems

> Much software culture today is based on the notion of trying to achieve perfect software, which of course is an in-your-face manifestation of designer centered design.
>
> *(P. Koopman and R. Hoffman, IEEE Intelligent Systems, 2003)*

THE AIM OF THIS CHAPTER is to give a brief outline of the background and some of the terminology used in this thesis. It gives detail on the systems this thesis is focused on, and an overview of the methods used to correctly specify and develop them, and explains why they are necessary but often insufficient to avoid system failures, and in some rare events even disasters.

Section §2.1 introduces the notion of a *distributed reactive and real-time system*, and explains the necessity for the use of *formal methods* in their development process as well as subsequent operation (see §2.2). In §2.3.1, the different problems which can occur during the execution of distributed and reactive systems are classified, and a practical common denominator identified, namely the notion of a *symptom*, which underlies the used concept of system correctness at runtime.

Unfortunately, over more than two decades, the different research communities (reliability, safety, security, etc.) have given different definitions of common terms such as failure, fault, or error, somewhat complicating proper classification. However, in this thesis, the commonly accepted definitions of the community of so-called *dependable systems* are used. A large part of this terminology was introduced by very early works of Avižienis [1967], Carter [1979] or Laprie [1992] to name just a few of the most influential authors, and acts as a reference for many disciplines even today (cf. Lee and Anderson [1990], Avižienis et al. [2001], and Breitling [2001]).

The author's intention is *not* to give a comprehensive list of all the terms which are commonly associated with the concept of dependability, fault tolerance, or safety. This is done more thoroughly and in various "handbooks", or practical guides for dependability and reliability engineering, such as those by Musa et al. [1987], Ebeling [1997], and Pham [2003], for instance. This chapter focuses on those concepts mandatory for an understanding of the remainder of this thesis.

2.1 Distributed reactive and real-time systems

A *real-time system* is one in which the temporal aspects of its behaviour are part of its specification. Therefore, the correctness of a real-time system not only depends on its generated output, but also on the time at which it becomes available. Many contemporary computer systems qualify as being real-time sensitive, ranging from complex business information systems to telecommunication routers, and even highly specialised controllers to be found in modern vehicles such as cars, passenger aircraft, or space shuttles. The degree of sensitivity or responsiveness determines whether the system operates under *hard or soft real-time constraints*. The constraint for hard real-time systems is that no deadlines must be missed during execution, whereas for soft real-time it is acceptable to miss some deadlines, occasionally. Although there exists no universally precise definition of "occasionally", it should be obvious that the timing requirements of, say, a business information system are inherently different to those of an airbag control system.

A common characteristic of real-time systems, however, is that they may, and increasingly do, consist of many components operating in parallel; they are then known as *concurrent (real-time) systems*. As many real-time systems must react to every stimulus from the environment, they are then often also referred to as *reactive systems* (Pnueli [1986]); a characterisation which incidentally accounts for real-time sensitive business information systems, and not only controller devices.

In the past, reactive systems were mostly *centralised systems* (or *single-processor systems*) consisting of a single CPU, its memory, peripherals, terminals or other physical inputs such as sensor devices. However, the development of increasingly compact and powerful microprocessors, as well as advances in network technology, have led to a more decentralised architecture of many such systems, now consisting of a collection of independent computers or CPUs, each connected by a shared network. Such systems are called *distributed systems* and are conceived by the user as being one single system. For example, an anti-lock brake system (ABS) is a physically distributed real-time system that the driver perceives as a single functionality in the car. In reality, several CPUs, sensors, and actuators are involved in order to activate the system in a timely precise manner. The overall correctness of the ABS therefore depends on the correctness of every subsystem. However, the ability to show correctness of a distributed real-time system is usually indirectly proportional to the number of subsystems, interactions, and general dependencies, physical or logical.

As noted by Harel and Pnueli [1985], a reactive system "resembles a cactus rather than a box", containing multiple inputs, outputs and various relations between them

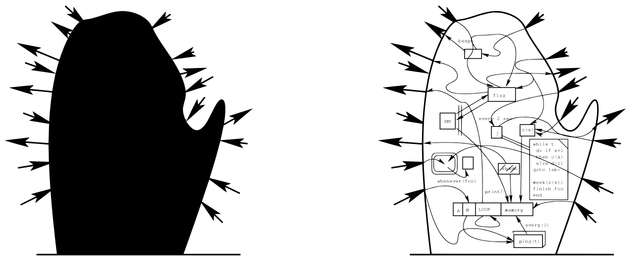


Fig. 2.1: Reactive systems resemble a cactus rather than a box. Various dependencies exist between various inputs and outputs. (Source: Möller [2002]).

(see Fig. 2.1). Capturing the entire state space of such systems is, therefore, generally infeasible. That is why they must be specified and verified in terms of their behaviours. A fact which has to be taken into account not only during specification and development, but as this thesis argues, also and especially at runtime.

A recent and rapidly evolving development complicates reasoning over distributed reactive systems even further: distributed reactive and real-time systems are increasingly *embedded*; that is, they are integrated into physical devices other than computers, such as cars, aircraft, mobile phones, or even washing machines, which impose more or less strict deadlines on the controlling software, and are often part of safety-critical applications.

2.2 The quest for correctness

It is a primary goal of software and systems engineering to develop systems which are correct with respect to a precise specification describing the systems to be constructed in an unambiguous manner. Therefore, formal methods such as *static verification* by means of model checking (cf. Clarke et al. [1999]), or deductive reasoning by means of theorem proving (cf. Gabbay et al. [1994]) get employed in the design and development process, if at all, in order to prove that a system satisfies a set of predetermined and desired properties. In contrast to a *dynamic verification* method, static verification subsumes all methods which can be applied without actually executing the system for analysis.[1]

The term formal method refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. Gen-

[1]The term "static verification" must not be confused with a reference to the area of *static analysis* (cf. Ball and Rajamani [2002]). Here and in the remainder, the term "static" is used in combination with those tasks of analysis that can be performed without having to actually execute the system under scrutiny.

erally, specifications used in formal methods are defined by well-formed statements in a mathematical logic and the formal verifications are rigorous deductions in that formalism, i. e., each step follows from an inference rule and can thus be checked by a mechanical process (cf. Broy [1999]). Naturally, many such techniques are nowadays tool-supported and even fully automated, e. g., model checking finite state spaces.

Common examples of formal methods often target the early design stages of the system development process, such as checking of design documents and models, or facilitating their creation in the first place. Other techniques, like *model-based testing* (cf. Broy et al. [2005]) aim at later stages in the process, where prototypes exist and systems have been fully developed. In essence, model-based testing not only covers the actual test process of a system, but also caters for the generation of a test suite from abstract system models and use-cases.

System testing can mitigate some well-known limitations of the aforementioned techniques: for example, automated model checking works only over finite state systems which are, if necessary, obtained from adequate abstractions, but may nonetheless lead to exponentially large state spaces which even modern computers are incapable of processing (cf. Clarke et al. [1999]). Further, a common problem in using theorem provers, for instance, aside from finding proofs that actually *do* verify a system property, is the even more time consuming task of analysing proofs that have failed (cf. Pike et al. [2004]).

Usually, system testing is the last step before finalising a system, or shipping a product, although newer methodologies, such as *agile methods*, emphasise testing also for the early stages (cf. Beck and Fowler [2000], Beck [2002]). It has long been recognised in software systems development that testing costs range from 50% to 70% of the cost of producing the first working version of a system (cf. Boehm [1981], Jones [1998], Dustin et al. [1999]). As illustrated in Fig. 2.2, in the widespread *V-model* each development step is associated with a test or validation effort at the same level of abstraction; thus, putting strong emphasis on this activity (cf. Dröschel and Wiemers [2000], or for recent updates Broy and Rausch [2005]).

However, model-based testing is by far not the only test method used in practice. For instance, Pretschner [2003] assigns all those activities, which aid in showing accordance or aberration of a system's implementation with its actual and intended behaviour to the wide area of testing.[2]

Obviously, testing is a useful and mandatory step in the development of complex distributed reactive systems, but there are various issues to keep in mind. Foremost, faults in the implementation are usually the most costly ones for removal: as a rule of thumb, one can say the later a fault is discovered in the process, the more expensive it is to resolve, e. g., prototypes may need to be changed, test cases may need to be redeveloped, re-executed, and reassessed, and so on (cf. Boehm and Papaccio [1988],

[2]Depending on the research community one looks at, the term testing sometimes refers to a dynamic, and sometimes to a static method, e. g., when applied to early requirements or design documents in terms of so-called "walkthroughs", or reviews.

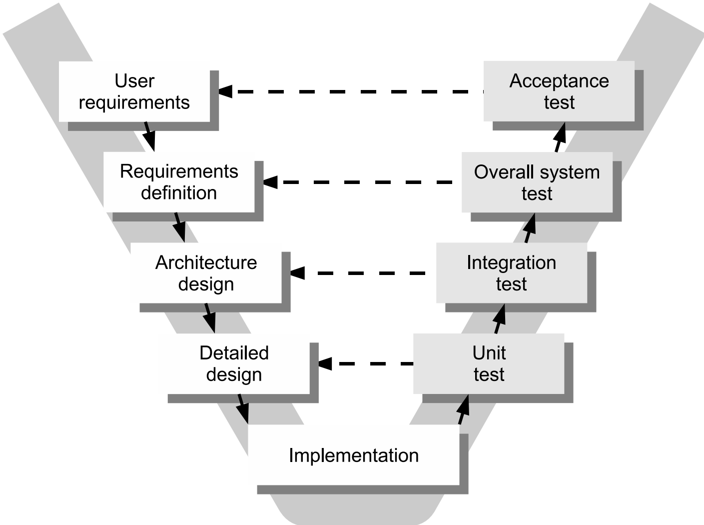


Fig. 2.2: The V-model for software and systems development.

Jones [1998], Dustin et al. [1999]). However, an even more fundamental problem with testing as a validation activity, has already been acknowledged in an early contribution by Dijkstra (see Dahl et al. [1972]) who pointed out that testing merely allows us to detect the presence of a fault, but not its absence.

The feasibility as well as the limitations of the aforementioned (formal) methods can be illustrated by a simplified, but nonetheless realistic example. Suppose a custom communication protocol has to be designed and implemented. At some stage in the design and development process, the designer may then want to use a model checking tool in order to show the protocol's correctness with respect to a formal specification; this is standard today (cf. Holzmann [1991]). However, a satisfactory result does not necessarily lead to a correct implementation (let alone using the model checker to verify the suitability of the protocol's properties for real-world use, in which inherent design assumptions may be violated and thus lead to unintended behaviour). However, the confidence in the correctness of the system can be increased further, by performing tests on the actual system which implements the protocol. But here Dijkstra reminds: testing efforts alone, regardless of how elaborate, cannot guarantee that the system will actually operate as is intended under all possible conditions. These efforts merely show conformance to preselected test cases, but not anything more.

In many scenarios it is therefore desirable, and in the case of so-called *safety-critical systems* often even mandatory, to arrange for additional, dynamic verification and analysis methods, performed transparently at runtime, which help detection and response to *unforeseen* and therefore highly situation-dependent faults occurring in relation only to very specific environmental conditions.

Note that a safety-critical system is referred to as such, if a failure of the system can have catastrophic impact on the environment, or cause injury and even death to human beings. Examples of safety-critical systems are airbag or brake controllers in modern cars as well as control systems of nuclear power plants. Due to the pervasiveness and compactness of computer systems, an increasing number of modern distributed real-time systems are commonly regarded as being safety-critical, although there are various, and also variously precise terms associated with criticality. The working group 10.4 on *Dependable Computing and Fault Tolerance* of the *International Federation of Information Processing* (IFIP) defines safety-criticality in terms of an attribute of the more general class of *dependable systems* (cf. Laprie [1992], Avižienis et al. [2001], IFIP/WG-10.4).

2.3 Terminology and classification

An incorrect, but nonetheless popular belief—that also led to what later became known as the "Ariane 5 disaster"—is that software does not break. For the launch of the Ariane 5 rocket, a software component was re-used that had been employed in the predecessor Ariane 4. In their minds, the responsible engineers had no reason to doubt that the software for Ariane 4 would have deteriorated since, and that it could impose a serious risk to the successor mission that it was later used in (cf. NASA [1996], Leveson [2004]).

At first glance, this reasoning may even seem plausible, because unlike mechanical parts such as bolts, levers, or electronic devices, software does stay as is—unless there are problems in the hardware that change the storage content or its data path. Indeed, software does not wear out, rust, deform or crack, and there is no environmental constraint for software to operate as long as the CPU it runs on is functioning. However, as Ariane 5 has demonstrated vividly, software systems do undergo various types of faults, deteriorating their operation, reducing their availability, and often even safety.

Essentially, software and systems engineers often refer to software errors merely in terms of *bugs* (which can be tackled, e. g., using so-called debugging tools; cf. Zeller [2005]). And once "these last few bugs have been removed, the system will be perfect and ready to go." In reality, such engineers are often in an ambivalent position as can be seen from various negative examples in computer history (cf. Lee and Anderson [1990], Zeller [2005]). This is because the notion of a "bug" grossly oversimplifies the matter and thus, gives a false sense of security. Various subtle differences between the cause of a problem and its visible system deviations exist, which usually deserve different methods of detection and treatment. Consequently, a classification and more precise terminology is needed for an understanding of the remainder of this thesis.

2.3.1 Failures, faults, and errors

Intuitively, a “bug” or a runtime error is defined by *something* going wrong in a system that may lead to undesired consequences and effects. In this section, the underlying concepts of “bugs”, such as system failures, faults, and errors, are examined in greater detail, and a classification of how these relate to one another is provided.

Failure. Laprie [1992] describes a *system failure* as an event that occurs when the delivered service of a system deviates from correct service. A system may fail either because it does not comply with the specification, or because the specification did not adequately describe its function. A failure is thus a transition from correct service to incorrect service, i. e., to not implementing the system function.

For the sake of simplicity, a *service* that a system delivers is defined merely in terms of the behaviour as it is perceived by its user. A user, in turn, may be another system, human or computer, that interacts at a dedicated service interface. An example is the fitting service model introduced by Powell [1995]: the service delivered by a system with a single user can be defined in terms of a sequence of service items, s_i, with $i = 1, 2, 3, \ldots$, each characterised by a tuple (vs_i, ts_i), where vs_i is the value, or content of service item s_i, and ts_i is the time, or instant of observation of service item s_i. A service item $s_i = (vs_i, ts_i)$ is *correct*, if and only if $vs_i \in SV_i \wedge ts_i \in ST_i$, where SV_i and ST_i are respectively the specified sets of values and times for service item s_i. Both SV_i and ST_i are generally defined as functions of the (history of) inputs, but may also be independent of the inputs. For many systems, the value and time sets are reduced to $SV_i = \{sv_i\}$, and $ST_i = [st_{\min}(i), st_{\max}(i)]$.

For example, a timer service could be defined in terms of the minimum scheduled time of the ith service item, $st_{\min}(i)$, the maximum scheduled time of the ith service item, $st_{\max}(i)$, and a maximum clock reading jitter of Δ as follows:

$$\begin{aligned} ST_i &= [st_{\min}(i), st_{\max}(i)] \\ SV_i &= [st_{\min}(i) - \Delta, st_{\max}(i) + \Delta] \end{aligned}$$

Various types of failure are distinguished in the literature. Let $s_i = (vs_i, ts_i)$ be a service item. Laprie [1992], amongst others (cf. Powell [1995], Burns and Wellings [2001]), points out that the impact of a failure can then be either in the *value domain*, *time domain*, or both:

- Value failure: the service delivers a wrong value, i. e., $s_i : vs_i \notin SV_i$.
- Time failure: the service does not deliver in a timely precise manner, i. e., $s_i : ts_i \notin ST_i$.
  - Too early: the service is delivered earlier than required, i. e., $s_i : ts_i < \min(ST_i)$
  - Too late: the service is delivered later than required (often referred to as performance error), i. e., $s_i : ts_i > \max(ST_i)$

- Infinitely late: the service is never delivered (often referred to as omission failure), i. e., $s_i : ts_i = \infty$

- Impromptu failure: the service delivers unexpectedly such that $s_i : vs_i \notin SV_i \wedge ts_i \notin ST_i$.

Mean time to failure. The reliability of a system is often expressed in terms of the *mean time to failure* (MTTF). Therefore it is necessary to establish the formal correlation between the reliability of a system and the occurrence of a failure. Suppose that the time to failure T has the probability density function $f(t)$. The *failure distribution function* is then the integral of the failure density function within interval $0 \leq \tau \leq t$:

$$F(t) = P(T \leq t) = \int_0^t f(\tau)d\tau, \text{ where } t \geq 0.$$

In contrast, the *reliability function* is the probability of a system *not* to fail between $0 \leq \tau \leq t$:

$$R(t) = 1 - F(t) = P(T > t).$$

The MTTF can now be defined as the integral of either the failure or the reliability distribution:

$$MTTF = E(T) = \int_0^\infty tf(t)dt = \int_0^\infty R(t)dt.$$

Related to MTTF is the concept of *mean time between failure* (MTBF) which is often better suited for systems in which failures may be reacted upon, and thus the rate of failure occurrence is of importance. For example, when the time required to repair a system is much shorter than MTTF it is safe to assume, without loss of generality, that:

$$MTTF \approx MTBF.$$

Error. When reasoning about distributed and reactive systems, as is also pointed out by Breitling [2001], the concept of an error only makes sense in reference to *state-based systems*. An error is then defined as part of the overall system state with respect to an aberration to a specified valid state. In other words, an error is that part of the system state which is liable to lead to subsequent failure. An error is latent, when it has not been recognised as such or detected by an algorithm, or mechanism (cf. Laprie [1992]).

Errors do not always lead to failure, e. g., an undefined value stored in some internal system variable may be overwritten before creating damage. Wrong values, for example, may be due to physical faults in memory hardware. The time that elapses before an error becomes noticeable is called "error latency". The time that elapses between a fault, and the manifestation as an error is called "fault latency".

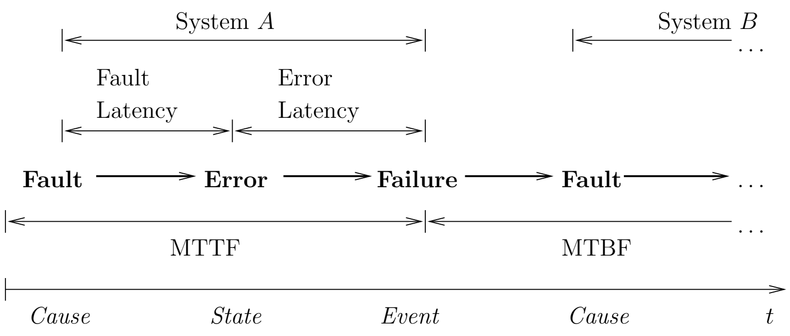


Fig. 2.3: Concepts and causality in a possible "fault chain".

Fault. In general, a fault is defined as the cause of an error (cf. Laprie [1992], Breitling [2001]). More specifically, a fault is called active if it produces an error, and dormant otherwise. For example, faults may be due to physical breakage, or caused by wrong assumptions regarding a system's execution environment.

In summary, failures and faults are probably best understood from the diagram depicted in Fig. 2.3. It illustrates in a systematic manner the different relations to each other and causality of events in a possible "fault chain" of a hypothetical distributed system consisting of part A and B. The original fault is in A, which causes an error in A leading to a failure. Since B depends on the correctness of A, the failure results in a fault in B, and so on.

If a fault originates from the physical environment a system operates in, i. e., is not caused by the system itself, it is often termed an *external fault*. A formal definition of an external fault in the context of this thesis is given in §5. Further, differentiation between faults which are internal, external, temporary, permanent, and so forth can also be made (cf. Laprie [1992]). Again, different research communities prefer different classification schemes.

2.3.2 Alternative correctness criteria

As could be seen in the previous section, the term correctness is always a relative one; that is, it is often possible to state that a system is correct with respect to a specification, e. g., by means of automatic verification techniques. However, when reasoning about systems during normal operation, this notion of correctness not necessarily helps in dealing with occurring runtime errors and faults. More so, this static notion of correctness can give users of a system a false sense of safety, where, in reality, there is no safety. Amongst other infamous "software disasters" (cf. Gilb [1988], NASA [1999], McCurdy [2001], Mann [2002], Zeller [2005]), Ariane 5 has shown that correctness with respect to a formal design document is not always sufficient to establish system safety at runtime. Clearly, when focusing on runtime errors and safety issues of (possibly safety-critical) distributed reactive and real-time systems, additional or alternative *correctness criteria* need to hold that can then be addressed dynamically, i. e., when the system operates.

Correctness at runtime

Benveniste [2002] defines *correctness at runtime* intuitively in terms of systems that behave the way they are intended to and are supposed to. This informal notion of correctness trivially asserts that under no circumstances, whatsoever, must a safety-critical system endanger the safety of its users and environment. If it does, even if strictly following its specified (and possibly also verified) behaviour, it is *not* correct in the above, intuitive sense. However, the definition by Benveniste is far too imprecise and based on it alone, no firm criteria can be established how to actually know when a running system deviates from its intended behaviour. What is required is a formal notion of correctness which addresses all sorts of behavioural aberrations in systems.

Symptom-freeness

Two important concepts, which are going to be used later-on in this thesis (see §4) include the notions of a *symptom* and also *symptom-freeness* as part of a correctness criterion for a system's behaviour at runtime.

Originally, the term symptom (Greek *símbtomma*, consisting of the words *syn* meaning con or plus, and *pipto* meaning fall; together to coexist) had two similar meanings in the context of physical and mental health (cf. Beys and Jansen [1999]):

- In the strict sense, a symptom is a sensation or change in health function experienced by a patient. Thus, symptoms may be loosely classified as being strong, mild or weak. In this medically correct sense of the word, a symptom is a *subjective* report, as opposed to a sign, which is objective evidence of the presence of a disease or disorder.
- A symptom may loosely be said to be a physical condition which shows that one has a particular illness or disorder (cf. Procter [1995]). An example of a symptom in this sense of the word would be a skin rash.

In the context of this thesis, a symptom represents a subjective report, whereas a failure is an objective evidence (of a fault). In other words, a symptom merely indicates that a failure might have occurred, but does not necessarily indicate where, if, or why. More so, in distributed systems, symptoms are often observed in or at components that turn out not to be faulty at all. Consequently, symptoms indicating faults in non-faulty systems are referred to as *false positives*, and symmetrically *false negatives*.

As in medical science, it can be difficult, time consuming, and take an unpredictable amount of time to find the root cause of a symptom, i. e., the fault that caused the failure, so that a system engineer or service technician knows what to replace or modify. Distributed reactive systems may show symptoms of failures, faults, and errors for a variety of different reasons; in fact, as is argued above, usually for an unlimited number of reasons when environmental influences are taken into account. For instance, physical failure is not only due to actually broken parts, but may also be due to temporary influence of other environmental conditions affecting, say, the communication medium used by a distributed system. Very high or low temperatures, strong electromagnetic interferences caused by nearby power lines or mobile phones, are just a few examples of what may degrade the efficiency of a communication bus with unpredictable consequences for the software systems sending and receiving signals over it.

At runtime, one is interested in detecting either problem whether it was anticipated by the designers of a system, pure coincidence, or physical breakage. As such, symptoms resemble a common denominator between the concepts of failure, fault, and error; that is, if either one is present in a system it will typically be noticed at some stage by means of showing noticeable behavioural aberrations. Using the notion of a *symptom-free* system, i. e., a system not revealing a failure, from this point forward, a system's execution is considered correct, if it does not reveal symptoms indicating the presence of a fault.

As such, the concept of a program *invariant* (cf. Gries [1982]) as originally introduced in *Hoare-logic* (Hoare [1969]) comes to mind, which basically resembles an assertion about a system's operation that has to hold before and after the operations are executed. If the operations, for whatever reason, lead to the invariant being invalidated, a natural conclusion would be that somewhere in the system a fault is present.

How to formally specify which system executions actually resemble aberrations, how to detect them technically at runtime, and then how to locate their causes, will be subject of the remaining chapters, where §5 provides a formal definition of symptom-freeness.

2.4 Summary

This chapter briefly outlines the general background of this thesis, and introduces the notions of reactive, real-time, concurrent, and distributed systems. It discusses means for developing and establishing correctness of these systems statically with respect to their specification and design documents, and identifies typical shortcomings of the employed methods, thus motivating the runtime reflection approach. Moreover, this section gives detailed definitions of common terms such as failure, fault, and error, and identifies symptoms as a general concept applicable for reasoning about systems correctness at runtime.

Chapter 3

Formal systems specification and verification with temporal logic

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> *(Donald E. Knuth,* Notes on the van Emde Boas construction of priority deques: An instructive use of recursion*)*

Detecting a system failure by means of runtime verification means having to continuously check user-defined system properties, e. g., invariants, specified over sequences of system executions. For the specification of desired system properties, *temporal logic* (see §3.1.2) has become an established and in some domains even standardised (see, e. g., §3.3.2) formalism not only to model, but also to verify system properties, statically and dynamically. Temporal logic allows one to reason about the behaviour of one or many cooperating systems over time. In this chapter, therefore, temporal logic is explored as a monitoring requirements specification language and details given on its formal syntax, semantics, and interesting properties with regard to systems verification.

Since the verification of temporal logic formulae is usually based upon a translation of the specifications into state-transition systems which constitute possible computations of a system, the link between state-transition systems, executable (finite and infinite) automata, and temporal logic itself is established first (see §3.1). In §3.2, important properties of temporal logic specifications are examined, such that they can be efficiently verified statically or even checked dynamically at runtime. This is done because not all specified properties can be checked by equal means, and, this section also explains why. Without going into detail at this point, one should keep in mind that runtime verification (i. e., observing a system while it executes) means having an only finite view on the system's behaviour. Therefore, properties which would require knowledge about the potentially infinite future of a system, inherently pose a problem in that setting. Practically, such properties are often "request-acknowledgement patterns" asserting that the system eventually acknowledges a request, but does not specify *when* this has to happen.

Despite temporal logic having a well-understood formal background, and despite its success in various practical domains, such as hardware verification (cf. Janssen [1989], Dill and Rushby [1996], Delgado Kloos and Damm [1997]), it is still widely considered as a tool for experts only. Therefore, most importantly, in §3.3, a structured and more high-level assertion and specification language is introduced, named SALT, which aims to address problems often experienced by users of plain temporal logic. However, SALT is fully embedded into formally defined temporal logics, and in that sense acts as a front end to the runtime reflection approach as presented in this thesis. Chapter §3.3 examines SALT's main language constructs, its expressiveness and semantics in detail, and concludes with various specification examples highlighting individual features of the language.

Note that using SALT, it is also possible to specify quantitative properties of event traces over time, which is often a prerequisite to deal with real-time systems that adhere to a continuous event-based execution scheme rather than a synchronous one. The underlying formal logic used in this setting is TLTL and introduced more formally in §4.5.1 of this thesis.

3.1 Preliminaries

Formal verification methods, such as *model checking*, often represent reactive systems as *Kripke structures*. A Kripke structure is a transition system or digraph, where the nodes correspond to system states. As such, they can be used in order to reason about a system's behaviour over time in terms of paths through the structure.

Formally, a Kripke structure can be defined as follows.

Definition 3.1.1: *Let AP be a finite set of propositions. A* Kripke structure *is a tuple $M = (S, s_0, \delta, l)$, where*

- *S is a finite set of states,*
- *$s_0 \in S$ some distinguished initial state,*
- *$\delta \subseteq S \times S$ a total transition relation, and*
- *$l : S \rightarrow 2^{AP}$ a labelling function assigning propositions to states.*

A state $s \in S$ has a transition to state $t \in S$, if and only if $(s, t) \in \delta$. A transition is *total*, if and only if $\forall s \in S : \exists t \in S : (s, t) \in \delta$.

From the verification perspective, a reactive system's *execution* then corresponds to an infinite sequence of states $w = a_0 a_1 \ldots$ in a Kripke structure, such that s_0 is the initial state and $(s_i, s_{i+1}) \in \delta$ for all $i \geq 0$. It can be seen that for every infinite sequence $w = a_0 a_1 \ldots$, there exists a sequence $\zeta = l(a_0) l(a_1) \ldots$, called the *interpretation* of w. For convenience, $l(w) = \zeta$ will be used as well.

From Kripke structures to model checking. Kripke structures, being syntactically close to automata, form the foundation for static verification by means of model checking (see §3.1.1). The model checking problem is relatively easy to describe as follows. A finite-state model of the system, M, and a property to be checked, φ, formulated in some temporal logic (see §3.1.2), are given. The verification tool then constructs an automaton, $\mathcal{A}_M$, which accepts all behaviours of M. Additionally, an automaton $\mathcal{A}_\varphi$ is constructed which accepts all models for $\neg\varphi$. The verification process then consists of checking for an automaton, $\mathcal{A} = \mathcal{A}_M \cap \mathcal{A}_\varphi$, the intersection of $\mathcal{A}_M$ and $\mathcal{A}_\varphi$ for emptiness, i. e., whether the accepted language of the automaton, denoted as $\mathcal{L}(\mathcal{A})$, is not empty. $\mathcal{L}(\mathcal{A})$, if defined, corresponds to all counter examples of M violating the property to be checked (see Fig. 3.1).

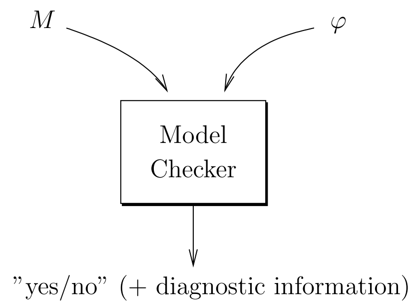


Fig. 3.1: The structure of a model checker.

That is, model checking exploits the idea that Kripke structures define a language of infinite words over 2^{AP}. The language accepted by a Kripke structure is denoted by $\mathcal{L}(M) = \{l(w) \mid w \text{ is an execution of } M\}$, and extend $l(s)$ to state sequences in a natural way. An execution, w, projected with the labelling function, $l(w)$, is then called a *computation* (of a system).

The basic approach was pioneered by works of Queille and Sifakis [1982], Clarke and Emerson [1982], and finally based on the notion of *ω-automata* by Vardi and Wolper [1986]. It was later greatly optimised and first put into wider practice by works of Holzmann [1991] (i. e., the SPIN tool), and McMillan [1992] (i. e., the symbolic model verifier tool SMV). The latter introduced an efficient symbolic representation of a system's state space in terms of *ordered binary decision diagrams* (OBDD). Employing OBDDs, it was possible to symbolically check state spaces far beyond 10^6 states, which had roughly been state-of-the-art before McMillan's findings (cf. Clarke et al. [1999]).

3.1.1 Automata over strings

Automata are generally defined over *alphabets*. Therefore, first the notion of an alphabet is introduced.

Definition 3.1.2: *An* alphabet Σ *is a non-empty finite set. The elements of* Σ *are called* actions.

Generally $a, b, c, \ldots$ are used to denote (a system's) actions. Further, let Σ^* be the set of finite *strings* over Σ, and Σ^ω the set of (countably) infinite strings generated by Σ with $\omega = \{0, 1, 2, \ldots\}$. Set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$, and let ϵ denote the empty word.

Finite automata

Before continuing with a formal introduction of infinite automata as they are used for model checking, the notion of a *finite automaton* is recalled as it also plays an important part in the particular runtime verification approach developed in this thesis (see §4.4 and §4.5.4, respectively).

Definition 3.1.3: *A (nondeterministic)* finite automaton *over* Σ *is a tuple* $A = (\Sigma, Q, Q_0, \delta, F)$, *where*

- Σ *is an alphabet,*
- Q *a finite set of states,*
- $Q_0 \subseteq Q$ *a distinguished set of initial states,*
- $\delta : Q \times \Sigma \to 2^Q$ *a transition function, and*
- $F \subseteq Q$ *a distinguished set of final states.*

Definition 3.1.4: *A (nondeterministic) finite automaton* A *is called* complete, *if and only if for all* $q \in Q$ *and* $a \in \Sigma$ *there exists a state* $q' \in Q$ *with* $\{q'\} \subseteq \delta(q, a)$.

Theorem 3.1.1 (Folklore): *For every non-complete (nondeterministic) finite automaton* $A = (\Sigma, Q, Q_0, \delta, F)$, *there exists a complete automaton* $A' = (\Sigma, Q', Q_0', \delta', F')$, *such that* $\mathcal{L}(A) = \mathcal{L}(A')$.

Proof:
Let A' be the same automaton as A except for one more state $q_e \in Q'$, which is not in Q. Then, δ' extends δ to include the additional state. If a run reaches q_e it means that the word leading to it is not in the language of A'. Let $q \in Q$, then

$$\delta' = \delta \cup \{(q_e, a, q_e) \mid a \in \Sigma\} \cup \{(q, a, q_e) \mid \text{there is no } q' \in Q \text{ with } (q, u, q') \in \delta\}.$$

Obviously, A' is complete by the third union above, and then it holds that $\mathcal{L}(A') = \mathcal{L}(A)$, since the transitions added only go to the new state if the read word is not accepted by A and leaves A' in the "error state" (second union). $\square$

The reference "(nondeterministic)" is omitted from this point forward, whenever the type of automaton is either irrelevant or clear from the context.
A finite automaton $A = (\Sigma, Q, Q_0, \delta, F)$ is assumed for the rest of this section.

Definition 3.1.5: *A* finite run *of a finite automaton A over a string $u = a_0a_1 \dots a_n \in \Sigma^*$ is a mapping $\rho : \mathbb{N} \to Q$ such that*

- $\rho(0) \in Q_0$, *and*
- $\rho(i+1) \in \delta(\rho(i), a_i)$ *for all $i < n$, where $i, n \in \mathbb{N}$.*

Definition 3.1.6: *A run $\rho = q_0q_1 \dots q_n$, where q_0 is an initial state in A, is called* accepting *if and only if $q_n \in F$.*

To tie everything together, $\mathcal{L}(A)$, the language of finite strings over Σ^* accepted by A, is $\mathcal{L}(A) = \{u \in \Sigma^* \mid \text{ there exists an accepting run of } A \text{ over } u\}$.

It is well-known (cf. Hopcroft and Ullman [1979]) that a language $L \subseteq \Sigma^*$ is accepted by some nondeterministic automaton if and only if it is accepted by some *deterministic finite automaton*, i. e., where Q_0 is a singleton set, and for all $q \in Q$ and $a \in \Sigma$ it holds that $|\delta(q, a)| = 1$.

The class of languages of finite strings over Σ accepted by finite automata are the *regular languages*. These languages are also referred to as being the *recognisable languages*. Another formalism used to describe regular languages of finite strings is the concept of *regular expressions* (Kleene [1956]).

Definition 3.1.7: *The* syntax of regular expressions *is inductively defined by the following grammar:*

$$\pi ::= \emptyset \mid \epsilon \mid a \mid \pi + \pi \mid \pi;\pi \mid \pi^* \quad (a \in \Sigma),$$

with $\pi \in RE(\Sigma)$, $\emptyset$ denoting the empty set, and ϵ denoting an empty word.

The semantics of a regular expression can be laid down by a mapping associating the set of finite strings described by it.

Definition 3.1.8: *With each regular expression, a set of finite strings is associated via the* map $[\![\cdot]\!] : RE(\Sigma) \mapsto 2^{\Sigma^*}$. *This map is defined in an inductive manner as follows:*

$$\begin{array}{lcll}
[\![\emptyset]\!] & = & \emptyset & \\
[\![\epsilon]\!] & = & \{\epsilon\} & \\
[\![a]\!] & = & \{a\} & \textit{(for each } a \in \Sigma\textit{)} \\
[\![\pi_1 + \pi_2]\!] & = & [\![\pi_1]\!] \cup [\![\pi_2]\!] & \textit{(union)} \\
[\![\pi_1 ; \pi_2]\!] & = & [\![\pi_1]\!][\![\pi_2]\!] & \textit{(composition)} \\
[\![\pi^*]\!] & = & \bigcup_{i \in \mathbb{N}} [\![\pi]\!]^i & \textit{(iteration)},
\end{array}$$

where π^* denotes the Kleene-star *or* iterator *which is recursively interpreted as follows:*

$$\begin{array}{lcl}
[\![\pi]\!]^0 & = & \{\epsilon\}, \textit{ and} \\
[\![\pi]\!]^{i+1} & = & \{uv \mid u \in [\![\pi]\!] \textit{ and } v \in [\![\pi]\!]^i\} \quad \textit{(for every } i \in \mathbb{N}\textit{)}.
\end{array}$$

Let the language $L = [\![\pi]\!]$ be the set of finite strings defined by π, then L^* defines the set of all those strings which can be made by concatenating zero or more strings from L. For example, in the most simple form $\Sigma = \{a, b\}$; then application of the iterator yields $\Sigma^* = \{\epsilon, a, b, aa, bb, ab, aaa, \ldots\}$.

Notice the above operations and constants of regular expressions together form an algebra better known as *Kleene algebra* (cf. Kozen [1990]).

On the other hand, automata are typically represented as edge-labelled *digraphs*, where nodes correspond to states and edges to transitions labelled with actions. As such an edge labelled with $a \in \Sigma$ from node $q \in Q$ to a state $q' \in Q$ exists if and only if $q' \in \delta(q, a)$. Initial states in Q_0 are marked with an incoming arrow, final (or, accepting) states in F with double circles.

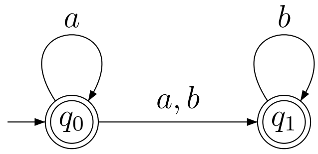


Fig. 3.2: Graphical representation of a nondeterministic finite automaton.

Example. Fig. 3.2 depicts a representation of a nondeterministic finite automaton, $A = (\{a, b\}, \{q_0, q_1\}, \{q_0\}, \delta_A, \{q_0, q_1\})$, where

$$\delta_A = \{(q_0, a, q_0), (q_0, a, q_1), (q_0, b, q_1), (q_1, b, q_1)\},$$

and each $(q, s, q') \in \delta_A$ means that there exists a transition from state q to q' with symbol a. This automaton accepts arbitrarily many a's followed by arbitrarily many b's (including zero occurrences of either action); that is, the language defined by the regular expression $a^*; b^*$. In this example, it formally holds that $\mathcal{L}(A) = [\![a^*; b^*]\!]$.

This correspondence has been recognised and generalised in Kleene's famous theorem:

Theorem 3.1.2 (Kleene [1956]): *A language $L \subseteq \Sigma^*$ is regular if and only if there exists a regular expression $\pi \in RE(\Sigma)$ such that $L = [\![\pi]\!]$.*

From that, it follows that both regular expressions and finite automata characterise the class of regular languages of finite strings over Σ.

Omega automata

Omega automata (in short, *ω-automata*) are structurally equivalent to finite automata as introduced above, i. e., they are represented in terms of a 5-tuple, denoted as $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$, but yield a suitably modified acceptance condition. More specifically,

ω-automata provide acceptance over *infinite* strings. This is different to the previous setting of ordinary finite automata (in contrast to ω-automata, abbreviated from this point forward as $*$-*automata*), where the last state visited by a run ultimately determines its acceptance. Infinite strings of actions resemble the nature of continuous interactions of reactive systems with their environment in a more natural way than finite strings. Moreover, certain system properties cannot be verified at all on finite prefixes of strings of actions (see §3.2.2).

Probably the most widely used class of ω-automata are *Büchi automata* (Büchi [1962]). A Büchi automaton is an ω-automaton equipped with a *Büchi acceptance condition* over infinite runs. The according definition is as follows.

Definition 3.1.9: *An* infinite run *of an ω-automaton $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$ over a string $w = a_0 a_1 \ldots \in \Sigma^\omega$ is a mapping $\rho : \mathbb{N} \mapsto Q$ such that*

- *$\rho(0) \in Q_0$, and*
- *$\rho(i+1) \in \delta(\rho(i), a_i)$ for all $i \in \mathbb{N}$.*

For an infinite run ρ and an ω-automaton $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$, the set of states visited infinitely often is denoted by

$$\mathrm{Inf}(\rho) = \{q \in Q \mid \text{ for infinitely many } k \in \mathbb{N}, \text{ it holds that } \rho(k) = q\}.$$

Definition 3.1.10: *An infinite run $\rho \in Q^\omega$ of a (nondeterministic) ω-automaton $A = (\Sigma, Q, Q_0, \delta, F)$ is called* Büchi-accepting, *if and only if Inf(ρ) $\cap F \neq \emptyset$.*

In other words, a Büchi-accepting run passes infinitely often through at least one final state in F.

The language accepted by a Büchi automaton $\mathcal{A}$ is then defined as $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\omega \mid \text{ there exists an accepting run of } \mathcal{A} \text{ over } w\}$. A language $L \subseteq \Sigma^\omega$ is definable by a Büchi automaton, i. e., is *Büchi recognisable*, if and only if there exists a Büchi automaton $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = L$. If $\mathcal{L}(\mathcal{A}) = \emptyset$, the automaton is called *empty*.

For the remainder of this section, unless otherwise noted, a Büchi automaton $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$ is assumed. $\mathcal{A}$ is *deterministic*, if and only if for all $q \in Q$, and $a \in \Sigma$ it holds that $|\delta(q, a)| \leq 1$, and Q_0 is a singleton set.

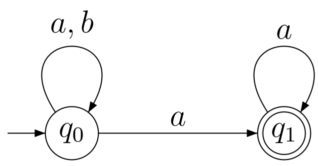


Fig. 3.3: Graphical representation of a Büchi automaton.

Example. Consider the graphical representation of a Büchi automaton as depicted in Fig. 3.3. It accepts the language $L = [\![(a+b)^* a^\omega]\!]$, if the Kleene-star is complemented with an infinite iteration or ω-operator as in π^ω. In other words, the language spans all words which consist of finitely many a's or b's, followed by infinitely many a's.

The previous is also an example for a class of languages referred to as the *ω-regular languages*.

Definition 3.1.11: *A subset $L \subset \Sigma^\omega$ is called an* ω-regular language, *if and only if L is a finite union of $U; V^\omega$, where $U, V \subseteq \Sigma^*$ are regular sets of finite words, and $V^\omega = \{v_1 v_2 \ldots \mid v_i \in V$ for all $i \in \mathbb{N}\}$.*

Büchi showed (see Theorem 3.1.3) that Kleene's Theorem carries over to the setting of ω-regular languages in a straightforward manner. Like in the previous example, ω-regular languages can be captured by regular expressions complemented with the ω-operator, then also called *ω-regular expressions*, to denote infinite recurrence. Thus it seems natural to take the ω-regular languages to be the Büchi recognisable languages.

Theorem 3.1.3 (Büchi [1962]): *A language $L \subseteq \Sigma^\omega$ is definable by a nondeterministic Büchi automaton, if and only if L is ω-regular.*

The theorem hints to the fact that nondeterministic Büchi automata are strictly more expressive than deterministic ones. In other words, it is possible to define an ω-regular language using a nondeterministic Büchi automaton, which cannot be recognised by a deterministic Büchi automaton. For example, there exists no deterministic Büchi automaton which accepts the language used in the previous example, i. e., $L = [\![(a+b)^* a^\omega]\!]$.
To be able to use deterministic ω-automata, various other acceptance conditions have been introduced in the literature, which all capture the full class of ω-regular languages. An important one is, e. g., *Muller acceptance*, which is defined over a set of accepting state sets $\mathcal{F} = \{F_i\}_{i=1}^n$, where each $F_i \subseteq Q$:

Definition 3.1.12: *An infinite run ρ of a (nondeterministic) ω-automaton $\mathcal{A} = (\Sigma, Q, Q_0, \delta, \mathcal{F})$ is called* Muller-accepting, *if and only if Inf(ρ) $\in \mathcal{F}$.*

In other words, an infinite run is Muller accepting, if and only if the infinitely recurring states exclusively match those in a set $F_i \in \mathcal{F}$. ω-automata equipped with Muller acceptance are referred to as *Muller automata*, denoted by $\mathcal{M}$. Muller automata can be turned into deterministic ones, while still accepting the same language as their nondeterministic counterparts. This is summarised in the theorem of McNaughton [1966] as follows.

Theorem 3.1.4 (McNaughton's Theorem—Part 1): *If a language L is deterministically Muller recognisable, then L is (nondeterministically) Büchi recognisable.*

Most importantly, the opposite direction also holds.

Theorem 3.1.5 (McNaughton's Theorem—Part 2): *If a language L is (nondeterministically) Büchi recognisable, then L is deterministically Muller recognisable.*

Safra [1988] and Michel [1988] then demonstrated—the former by algorithmic construction—that for every nondeterministic Büchi automaton, $\mathcal{A}$, comprising n states and accepting the language $\mathcal{L}(\mathcal{A})$, there exists a deterministic Muller automaton, $\mathcal{M}$, comprising of at most $2^{O(n \log n)}$ states, such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{A})$.

Unlike in the case of determinisation, Büchi automata are closed under *complementation*. That is, given a Büchi automaton, $\mathcal{A}$, accepting the language $\mathcal{L}(\mathcal{A})$, *Safra's construction* can be used to obtain a Büchi automaton, $\overline{\mathcal{A}}$, which accepts $\mathcal{L}(\overline{\mathcal{A}})$ such that $\mathcal{L}(\overline{\mathcal{A}}) = \overline{\mathcal{L}(\mathcal{A})}$, where $\overline{\mathcal{L}(\mathcal{A})} = \Sigma^\omega \backslash \mathcal{L}(\mathcal{A})$. This, again, involves an exponential "blowup" as is summarised in the following theorem.

Theorem 3.1.6 (Safra [1988]): *Let $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$ be a Büchi automaton with $|Q| = n$. Then there exists a Büchi automaton $\overline{\mathcal{A}} = (\Sigma, Q', Q'_0, \delta', F')$, where $|Q'| = 2^{O(n \log n)}$.*

Besides Muller there exist alternative acceptance conditions for infinite runs, such as realised in *Rabin* or *Streett automata*, all of which span the full class of ω-regular languages as well. For completeness, these are outlined only very briefly here as they compare to Muller automata and do not play an important role in the remainder of this thesis. However, given their expressiveness, they could also be considered as possible alternatives for an implementation of the runtime reflection framework.

Rabin acceptance is defined over pairs of state sets as in $\mathcal{F} = \{(F_i, F'_i)\}_{i=1}^n$. In this scheme an infinite run $\rho \in \Sigma^\omega$ is accepting if there exists some $1 \leq i \leq n$ such that $\mathrm{Inf}(\rho) \cap F_i = \emptyset$ and $\mathrm{Inf}(\rho) \cap F'_i \neq \emptyset$. In other words, the run visits finitely often states from F_i, and infinitely often states from F'_i. *Streett acceptance* is somewhat dual to Rabin in that it is also defined over pairs of sets, but for every i it has to hold that $\mathrm{Inf}(\rho) \cap F_i \neq \emptyset$ implies also $\mathrm{Inf}(\rho) \cap F'_i \neq \emptyset$. Since Muller, Streett, and Rabin automata are all closed under complement as well as determinisation, it shall suffice at this point to refer to Farwer [2001] for a good overview on the exact correlations and properties of these automata.

Another important property of Büchi automata, which has been implicitly used already, but which needs to be discussed in more detail in the context of this thesis is the *emptiness check*. A Büchi automaton is non-empty, if and only if there exists a cyclic path, reachable from an initial state that contains one or more accepting states. The automaton depicted in Fig. 3.3 is non-empty, since it accepts the language defined by $(a + b)^* a^\omega$. Since the number of states in a Büchi automaton is finite, there must exist a cycle on which a final state occurs for the automaton to be non-empty. Checking that such a cycle is reachable from some initial state can be achieved by a depth-first search algorithm, and in linear time as summarised in the following theorem.

Theorem 3.1.7 (Emerson and Lei [1985]): *The non-emptiness problem for Büchi automata is decidable in linear time.*

Now, coming back to the original problem of model checking models of reactive systems, observe that Kripke structures, the computational model introduced in the beginning of this section, correspond directly to ω-automata, where all the states are accepting. Let $M = (S, s^0, \delta, l)$ be a Kripke structure over a set of propositions AP. Then, a Büchi automaton $\mathcal{A}_M = (\Sigma, S_M, S_M^0, \delta_M, F_M)$ can be defined, where

- $\Sigma = 2^{AP}$,
- $S_M = S \cup \{s^i\}$,
- $S_M^0 = \{s^i\}$,
- $F_M = S_M$, and
- for all $s, s' \in S_M$, $a \in \Sigma$ it holds that $(s, a, s') \in \delta_M$ if and only if $l(s) = a$ and $((s, s') \in \delta)$ or $(s = s^i$ and $s' = s^0)$.

The Büchi automaton $\mathcal{A}_M$ accepts exactly those infinite sequences of labellings which correspond to infinite paths of the Kripke structure starting from some initial state. The next section discusses how, given a formula φ, formulated in some linear temporal logic, a Büchi automaton $\mathcal{A}_\varphi$ can be created, which accepts exactly all the infinite sequences of valuations satisfying φ.

3.1.2 Linear time temporal logic

As seen in the previous section, ω-regular expressions can be used to specify properties over infinite strings; although, the use of ω-regular expressions may not always turn out to be the most convenient means of specification. Thinking of infinite strings in terms of infinite computations, a more intuitive means of specification is the use of logical operations transforming simple expressions into more complex expressions (see §3.3). *Linear time temporal logic* (LTL) as proposed by Pnueli [1977] for the verification of reactive systems, can be used to specify computations of Kripke structures and ω-automata. However, the languages definable using LTL actually form a strict subset of the languages definable using ω-regular expressions (cf. Thomas [1990] and §3.3.2).

In this section, the formal syntax and semantics of LTL is recalled, and the satisfiability problem of arbitrary LTL formulae discussed based on Büchi automata.

Syntax

A finite set AP of atomic propositions is assumed, and a finite alphabet $\Sigma = 2^{AP}$ defined. a_i denotes a single element of Σ, i. e., a_i is a (possibly empty) set of propositions taken from AP. Finite strings over Σ are elements of Σ^*, and are usually denoted by $u, u', u_1, u_2, \ldots$, whereas infinite strings are elements of Σ^ω, and usually denoted by $w, w', w_1, w_2, \ldots$. For some trace $w = a_0 a_1 \ldots$, w^i denotes the suffix $a_i a_{i+1} \ldots$.

Definition 3.1.13: *The set of well-formed propositional linear time temporal logic formulae over an alphabet Σ is denoted by LTL(Σ), and given by the following* abstract syntax*:*

$$\varphi ::= true \mid a \mid \neg\varphi \mid \varphi\ op\ \varphi \mid \varphi\mathbf{U}\varphi \mid \mathbf{X}\varphi \quad (a \in \Sigma),$$

with $\varphi \in LTL(\Sigma)$, and where op represents a binary Boolean operator defined by the set $op \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$.

A formula $\varphi \in \mathrm{LTL}(\Sigma)$ that does not contain any temporal operators, i. e., $\mathbf{X}$ or $\mathbf{U}$, is called a *propositional formula* and *temporal formula*, otherwise.

If the parameterisation of the set of formulae is clear from the context, the set of propositions or the concrete alphabet in its name can be omitted, and simply LTL rather than LTL(Σ) be used, when referring to the set of well-formed LTL formulae.

Semantics

LTL formulae are generally interpreted over infinite strings of sets of atomic propositions chosen from AP, i. e., elements of the alphabet $\Sigma = 2^{AP}$. The classic semantics of LTL is then defined using a binary relation, $\models$, between infinite strings $w = a_0a_1 \ldots \in \Sigma^\omega$ of subsets of AP and, inductively, formulae of that logic. In the following, w, i is used to denote the ith position in an (infinite) string w.

Definition 3.1.14 (Basic operators): *Let $\varphi \in LTL(\Sigma)$ and $i \in \mathbb{N}$ denote a position. The* semantics *of LTL formulae is then defined inductively over infinite strings $w = a_0a_1 \ldots \in \Sigma^\omega$ as follows:*

$$\begin{array}{lll}
w, i \models true & & \\
w, i \models \neg\varphi & \Leftrightarrow & w, i \not\models \varphi \\
w, i \models p \in AP & \Leftrightarrow & p \in a_i \\
w, i \models \varphi_1\ op\ \varphi_2 & \Leftrightarrow & (w, i \models \varphi_1\ op\ w, i \models \varphi_2) \\
w, i \models \varphi_1\mathbf{U}\varphi_2 & \Leftrightarrow & \exists k \geq i : ((w, k \models \varphi_2) \wedge \forall l : (i \leq l < k \Rightarrow w, l \models \varphi_1)) \\
w, i \models \mathbf{X}\varphi & \Leftrightarrow & w, i + 1 \models \varphi
\end{array}$$

Further, let $w \models \varphi$, if and only if $w, 0 \models \varphi$.

$w \in \Sigma^\omega$ satisfies (alternatively, is a model of) the formula $\varphi \in \mathrm{LTL}(\Sigma)$, if and only if $w \models \varphi$ holds. The set given by $\mathcal{L}(\varphi) = \{w \in \Sigma^\omega \mid w \models \varphi\}$ of all models of φ is called the *language* of φ. The formula φ is satisfiable if $\mathcal{L}(\varphi) \neq \emptyset$ and unsatisfiable, otherwise. A formula φ is valid, if and only if $\neg\varphi$ is unsatisfiable.

LTL is now looked at from a more intuitive or from a user's perspective. Recall, the infinite strings in Definition 3.1.14 are the computations of a system observed over an (infinite) period of time. Intuitively, a formula $\varphi \equiv p \in AP$ then asserts that in the current instance an event (or, an observable system action), encoded by the propositional symbol p, has occurred. The truth value *true*, and the Boolean operators are interpreted as expected. The temporal operators, in turn, are interpreted as follows. An "until" formula, e.g., $\varphi \equiv \psi\mathbf{U}\eta$, states that η holds at a present or some future instant, and that ψ holds until then. A "next" formula, e.g., $\varphi \equiv \mathbf{X}\psi$, states that ψ holds in the next time instant. As such, the next operator mandates the notion of a *discrete time-step*. There also exist so-called *next-free* variants of LTL, where this discrete notion of time does not apply. Such variants, however, cannot distinguish between, e.g., the next and the second next state of a string and are, therefore, a strict subset of standard LTL (cf. Pnueli [1977], Lamport [1983], or Lichtenstein and Pnueli [1985]).

Besides the basic set of LTL-operators, there exist a number of practically useful derivations, which help make specifications more concise, and thus, readable, while still being relatively straightforward to translate back into standard LTL.

Let w and φ be defined as in Definition 3.1.14, respectively. The following can be defined on top of that.

Definition 3.1.15 (Derived operators):

$$\begin{array}{llll}
w \models \mathbf{F}\varphi & \Leftrightarrow & w \models \mathit{true}\mathbf{U}\varphi & \text{(eventually)} \\
w \models \mathbf{G}\varphi & \Leftrightarrow & w \models \neg\mathbf{F}\neg\varphi & \text{(globally)} \\
w \models \varphi_1\mathbf{U}_w\varphi_2 & \Leftrightarrow & w \models \mathbf{G}\varphi_1 \vee (\varphi_1\mathbf{U}\varphi_2) & \text{(weak until)} \\
w \models \varphi_1\mathbf{R}\varphi_2 & \Leftrightarrow & w \models \varphi_2\mathbf{U}(\varphi_1 \wedge \varphi_2) & \text{(release)} \\
w \models \varphi_1\mathbf{R}_w\varphi_2 & \Leftrightarrow & w \models \varphi_2\mathbf{U}_w(\varphi_1 \wedge \varphi_2) & \text{(weak release)}
\end{array}$$

For specifications, the most important derivations are the *eventually* and the *globally operators* (see also §3.2.1). The eventually or future operator, denoted by $\mathbf{F}$, asserts that a certain property has to hold in some present or future instant, while the globally operator, denoted by $\mathbf{G}$, expresses that a given property has to hold as long as the system's computations are observed and analysed, e.g., in the case of most reactive systems, infinitely. The remaining operators, typically, play no important role for the forward specification of system properties as they are mainly used to transform arbitrary LTL formulae into very specific *temporal normal forms*. Their meaning is not necessarily intuitive at all.

For instance, in *negative normal form* (NNF), LTL formulae are written such that all occurring negations directly precede atomic propositions. Additionally, only the connectives $\wedge$, $\vee$, $\mathbf{X}$, $\mathbf{U}$, and $\mathbf{R}$ are allowed. As such, the release operator acts foremost as *duality* for the more intuitive-to-handle until operator:

$$\begin{array}{rcl}
\neg(\varphi_1\mathbf{R}_w\varphi_2) & \equiv & (\neg\varphi_1\mathbf{U}\neg\varphi_2) \\
\neg(\varphi_1\mathbf{R}\varphi_2) & \equiv & (\neg\varphi_1\mathbf{U}_w\neg\varphi_2)
\end{array}$$

Vice versa, the following equivalences also hold:

$$\begin{aligned}\neg(\varphi_1 \mathbf{U} \varphi_2) &\equiv (\neg\varphi_1 \mathbf{R}_w \neg\varphi_2)\\ \neg(\varphi_1 \mathbf{U}_w \varphi_2) &\equiv (\neg\varphi_1 \mathbf{R} \neg\varphi_2)\end{aligned}$$

Formulae which are in negative normal form are often referred to as *normalised* or as *positive formulae*, since they make a formula "positive" with respect to its constituent literals. Thus, normalisation clearly identifies positive and negative occurrences of the constituent atoms in a formula. If the objective were simply to push down negations then also the symbols $\Leftrightarrow$, $\Rightarrow$ could be allowed, and the potentially exponential blowup avoided, using the tautology $\neg(\varphi \Leftrightarrow \psi) \Leftrightarrow (\neg\varphi \Leftrightarrow \psi)$. However, in the resulting formula it is no longer clear which variables occur positively and which negatively.

So far, only LTL operators have been considered that allowed the formalisation of requirements dealing with present and future obligations; that is, the so-called *future fragment of LTL* . The dual to pure future LTL is *past LTL* , and defined as follows.

Definition 3.1.16 (Past operators): *Let $\varphi \in LTL(\Sigma)$ and $i \in \mathbb{N}$ denote a position. The* past operators *of LTL are then defined inductively over infinite strings $w = a_0a_1 \ldots \in \Sigma^\omega$ as follows:*

$$\begin{array}{lcl} w,i \models \mathbf{Y}\varphi & \Leftrightarrow & i > 0 \wedge w, i-1 \models \varphi \\ w,i \models \mathbf{Z}\varphi & \Leftrightarrow & i = 0 \vee w, i-1 \models \varphi \\ w,i \models \mathbf{O}\varphi & \Leftrightarrow & \exists k \leq i : k \geq 0 \wedge w, k \models \varphi \\ w,i \models \mathbf{H}\varphi & \Leftrightarrow & \forall k \leq i : k \geq 0 \Rightarrow w, k \models \varphi \\ w,i \models \varphi_1 \mathbf{S} \varphi_2 & \Leftrightarrow & \exists k \leq i : ((k \geq 0 \wedge w, k \models \varphi_2) \wedge \forall l \leq i : (l > k \Rightarrow w, l \models \varphi_1)) \\ w,i \models \varphi_1 \mathbf{T} \varphi_2 & \Leftrightarrow & \forall k \leq i : ((k \geq 0 \Rightarrow w, k \models \varphi_2) \vee \exists l \leq i : (l > k \wedge w, l \models \varphi_1)) \end{array}$$

The past operators are dual to the future operators, in that $\mathbf{Y}$ is the opposite of $\mathbf{X}$ with special handling of the initial state: a formula $\mathbf{X}\varphi$ states that in the next instant φ must hold, whereas $\mathbf{Y}\varphi$ states that previously φ did hold. $\mathbf{Z}$ is similar to $\mathbf{Y}$, but deals with the initial instant differently. $\mathbf{O}$ is the once-operator, and therefore dual to $\mathbf{F}$, the eventually-operator. $\mathbf{H}$ is the past time version of the globally-operator and asserts in a formula $\mathbf{H}\varphi$ that φ did hold at all instances in the past. The dual for the until-operator is the since-operator, i. e., $\varphi_1 \mathbf{S} \varphi_2$ is true if and only if φ_2 holds somewhere in the past and φ_1 is true from then up to the present instant. Finally, $\mathbf{T}$ is a useful dual for normal form representations as it can be used to express the since-operator differently: $\varphi_1 \mathbf{T} \varphi_2 \equiv \neg(\neg\varphi_1 \mathbf{S} \neg\varphi_2)$.

Various authors such as Cimatti et al. [2004] have proposed and use the moniker PLTL (short for "past LTL") when referring only to the past fragment of LTL.

Gabbay [1989] provided a proof that every temporal logic formula using only the past fragment of operators can be transformed into a temporal logic formula using only future operators. The results spans all kinds of temporal logics, besides LTL, such as CTL, μ-calculus, etc. Hence, adding the past fragment to LTL does not add expressiveness, but may be more natural for users when reasoning particularly about past modalities.

For instance, the property "a grant follows and is issued only upon an according request" can be represented as

$$\mathbf{G}(grant \Rightarrow \mathbf{Y}(\neg grant \mathbf{S} request))$$

compared to the corresponding future fragment translation

$$(request \mathbf{R} \neg grant) \wedge \mathbf{G}(grant \Rightarrow (request \vee (\mathbf{X}(request \mathbf{R} \neg grant)))).$$

Obviously, the former variant is more concise and thus, easier for users to read. The intuition that adding past operators to LTL, indeed, does lead to more succinct formulae, has been formally justified by Markey [2003]. Since Markey's result, it is known that the succinctness gap between past and future LTL is, in fact, exponential with respect to the temporal operators. However, restating the formal proof is not carried out at this point as past LTL will play only a subsidiary role for the results of this thesis.

Some example properties

After having introduced the formal syntax and semantics of LTL, some typical examples of LTL-specifications are examined in greater detail. The following list contains some abstract LTL formulae, φ, and their intuitive semantics with respect to an infinite string of actions $w = a_0 a_1 \ldots \in \Sigma^\omega$, and propositions $p, q \in AP$. Formally, a *property* can be considered as a subset of Σ^ω, i. e., a possibly infinite set of strings of actions.

- $p \Rightarrow \mathbf{F}q$: if p holds at a_0, then q holds at a_i for some $i \geq 0$.
- $\mathbf{G}(p \Rightarrow \mathbf{F}q)$: every p is eventually followed by a q.
- $\mathbf{GF}q$: the sequence w contains infinitely many q's.
- $\mathbf{FG}q$: all but finitely many states in w satisfy q; q eventually stabilises.
- $\neg p \mathbf{U} q$: the sequence w must not contain p, before eventually q holds.
- $\mathbf{G}(p \vee \mathbf{G}\neg p)$: either p holds entirely over w, or if there is no $a_i \in w$ where p holds, then p does not hold in all future states from the current instant forward.

These example specifications are universal to use, but rather abstract in their present form since the respective propositions are not defined by any means. In practice, propositions like these are typically "instantiated" with more meaningful identifiers, such as names of programming language variables. For instance, *mutual exclusion* for two processes of a program may be specified by formula

$$\varphi \equiv \mathbf{G}\neg(at\_l_3 \wedge at\_m_3),$$

stating that no computation of a program includes a state in which some process P_1 is at l_3, while some other process P_2 is at m_3 at the same time.

A property asserting that, while a car is moving, its key cannot be removed may be specified in terms of

$$\mathbf{G}(\neg(\mathit{speed} = 0) \Rightarrow \neg(\mathit{ignition} = \mathit{keyout})),$$

where $(\mathit{speed} = 0), (\mathit{ignition} = \mathit{keyout}) \in AP$, i. e., are propositions. How to actually make sure that the vehicle speed compares to zero is not part of the specification. The formula merely captures the fact, $\mathit{speed} = 0$, in terms of a proposition.

Considering the call sequence of a program handling a complex data structure, one may want to specify that between a call to a function $\mathit{enqueue}(\mathit{data})$ and $\mathit{empty}(\mathit{true})$ there must be a call to $\mathit{dequeue}(\mathit{data})$:

$$\begin{aligned}\mathbf{G}((&\mathit{call\_enqueue}(\mathit{data}) \wedge \mathbf{F}\mathit{return\_empty}(\mathit{true})) \Rightarrow \\ &\qquad\qquad (\neg \mathit{return\_empty}(\mathit{true})\mathbf{U}\mathit{return\_dequeue}(\mathit{data}))).\end{aligned}$$

Finally, a more complex example taken from Dwyer et al. [1999] is studied below, which asserts that between the time an elevator is called at a floor and the time it opens its doors at that floor, the elevator can arrive at that floor at most twice:

$$\begin{aligned}&\mathbf{G}((\mathit{call} \wedge \mathbf{F}\mathit{open}) \Rightarrow \\ &\quad ((\neg \mathit{atfloor} \wedge \neg \mathit{open})\mathbf{U} \\ &\qquad (\mathit{open} \vee ((\mathit{atfloor} \wedge \neg \mathit{open})\mathbf{U} \\ &\qquad\quad (\mathit{open} \vee ((\neg \mathit{atfloor} \wedge \neg \mathit{open})\mathbf{U} \\ &\qquad\qquad (\mathit{open} \vee ((\mathit{atfloor} \wedge \neg \mathit{open})\mathbf{U} \\ &\qquad\qquad\qquad (\mathit{open} \vee (\neg \mathit{atfloor}\mathbf{U}\mathit{open}))))))))))).\end{aligned}$$

However, this last example is also showing some practical limitations of using LTL: the deep nesting of opening and closing brackets as well as of temporal operators may lead to subtle specification errors even for simple requirements, such as the elevator, if not engineered carefully.

Further use-cases for LTL are not examined until later-on in this thesis (see §4). The next two sections instead look at a classification scheme for properties and more or less convenient ways to express these.

3.2 Safety and liveness properties

The notion of safety and liveness properties has been first introduced by Lamport [1977], who also noted that all "interesting" properties of systems can be expressed using these classes of properties.

3.2.1 Safety properties

Intuitively, a *safety property* expresses that "something bad never happens" (during a system's execution). Formally, a safety property is defined with respect to a *bad prefix* as follows:

Definition 3.2.1: *Let $L \subseteq \Sigma^\omega$ be a language on infinite strings over alphabet Σ. A finite prefix $u \in \Sigma^*$ is a* bad prefix *for L, if and only if for every $w' \in \Sigma^\omega$ the following holds: $uw' \notin L$. If $\forall w \in \Sigma^\omega \backslash L : \exists u \in \Sigma^*$ such that $w = uw'$ for some $w' \in \Sigma^\omega$, and $\forall w'' \in \Sigma^\omega : uw'' \notin L$, L is a* safety language *(also called a* safety property*).*

In other words, a language, $L \subseteq \Sigma^\omega$, is a safety language, if and only if every infinite string not in L has a finite bad prefix. Thus, it follows that *all* counterexamples of safety properties are finite—an important characteristic which is exploited in runtime verification approaches.

The differentiation between safety and liveness properties has turned out a useful concept since it allows for specific and efficient verification methods (cf. Kupferman and Vardi [2001]), but there are many cases where this common classification is still too coarse. As a matter of fact, sometimes it is not easy to establish, based on the above definition, as to whether a given formula specified in LTL actually defines a safety language, or not.

Therefore, Sistla [1994] gave a syntactic, and more intuitive characterisation of LTL safety formulae, which is referred to in the following as *syntactic safety formulae*.

Theorem 3.2.1 (Sistla [1994]): *Every propositional formula is a safety formula. If φ, ϕ are safety formulae, then so are $\varphi \wedge \phi$, $\varphi \vee \phi$, $\mathbf{X}\varphi$, $\varphi\mathbf{U}_w\phi$, and $\mathbf{G}\varphi$.*

Proof:
A detailed proof of this theorem can be instructional for the understanding of the overall concept of safety properties. The proof here is slightly different to Sistla and proceeds by induction on the term structure of syntactic safety formulae.

Base case: Given that φ and ϕ are propositional formulae, it is straightforward to show that both express safety properties as follows. Let $w \in \Sigma^\omega$ be a string such that $w \not\models \varphi$ (respectively ϕ). From the semantics of LTL, it follows that $w, 0 \not\models \varphi$, which is a finite bad prefix for φ (respectively ϕ).

Induction: Let φ and ϕ be safety formulae and $w \in \Sigma^\omega$ a model such that $w \not\models \psi$, where ψ is defined as

1. $\psi \equiv \varphi \wedge \phi$. By definition of the $\wedge$-relation it is sufficient to show that either $w \not\models \varphi$, or $w \not\models \phi$ holds, based on a finite bad prefix. It follows directly from the base case that there exists $w \not\models \varphi$ such that $w, 0 \not\models \varphi$.
2. $\psi \equiv \varphi \vee \phi$. By definition of the $\vee$-relation it must hold that $w \not\models \varphi$ as well as $w \not\models \phi$, based on a finite bad prefix $u \in \Sigma^*$, where $w = uw'$. Since both φ and ϕ are safety properties, there exist finite bad prefixes, u_φ and u_ϕ. Setting $u_\phi \subseteq u_\varphi$ (or, vice versa), then u_φ (respectively u_ϕ) is a finite bad prefix for ψ.
3. $\psi \equiv \mathbf{X}\varphi$. By definition of the next operator, the existence of a string $w^1 \not\models \varphi$ which has a finite bad prefix must be shown. It follows directly from the base case that there exists an $i \geq 0$, such that for an element w_i^1 it holds that $w_i^1 \not\models \varphi$. Hence, any uw_i^1, where $u \in \Sigma^\omega$, is a finite bad prefix for ψ.
4. $\psi \equiv \mathbf{G}\varphi$. It follows from the base case that there is a string $w \not\models \varphi$, where $w = u_\varphi w'$ and $u_\varphi \in \Sigma^*$. By definition of the globally operator every u_φ which is a finite bad prefix for φ is also a finite bad prefix for ψ.
5. $\psi \equiv \varphi \mathbf{U}_w \phi$. Since $\varphi \mathbf{U}_w \phi \equiv \mathbf{G}\varphi \vee (\varphi \mathbf{U} \phi)$ and by 2., one basically has to show that every finite bad prefix $u \in \Sigma^*$, where $w = uw'$, $w' \in \Sigma^\omega$, and $w \not\models \psi$, is a finite bad prefix for both $\mathbf{G}\varphi$ *and* $\varphi \mathbf{U} \phi$.

   From the base case and 4. it follows directly that if there exists an $l^{\geq 0}$, such that $w, l \not\models \varphi$, then $w, l \not\models \mathbf{G}\varphi$. From the safety of φ and ϕ, and from 2. it then follows that there exists a position l in a “bad string” w, such that $w, l \not\models \varphi$ and $w, l \not\models \phi$. By the definition of the until operator, $w, i \models \varphi \mathbf{U} \phi$, if and only if $\exists k \geq i : ((w, k \models \phi) \wedge \forall l : (i \leq l < k \wedge w, l \models \varphi))$, every such w, l is then a finite bad prefix for ψ. □

In principle, the proof follows another one presented by Latvala [2002], but in that paper a different (although equivalently expressive) subset of LTL is used. As such, the present proof is more in the vein of Sistla's, using exactly the original set of operators, but more detailed to illustrate the underlying concepts.

It is easy to see, that the mutual exclusion and the car-key examples are not only safety properties, but also qualify as syntactic safety properties (see §3.1.2). However, the elevator example is not a syntactic safety property, since it contains the operators, $\mathbf{U}$ and $\mathbf{F}$. As a matter of fact, it will soon become clear that it is not even a safety property at all.

A concept which, in combination with monitoring safety properties, turns out to be useful is the one of a *stuttering* run. A property defined by $\varphi \in$ LTL is said to be closed under stuttering if and only if $w = w_0 w_1 \dots w_i w_{i+1} \dots \models \varphi$ as well as $w = w_0 w_1 \dots w_i w_i w_{i+1} \dots \models \varphi$. That is, the finite repetition of isolated states does not affect the validity of the property. Stuttering-invariant properties could be used, e. g., to tolerate minor measurement aberrations when actually monitoring real systems.

The next-free variant of LTL is known to express properties invariant to stuttering (cf. Clarke and Schlingloff [2001]). Sistla continues to show:

Proposition 3.2.1 (Sistla [1994]): *Every positive formula using only* $\mathbf{X}$ *and* $\mathbf{U}_w$ *as temporal operators is a safety formula. Every positive formula using only* $\mathbf{U}_w$ *as temporal operator then specifies a safety property closed under stuttering.*

Proof:
Merely the idea for this straightforward proof is sketched. It can be shown by induction that a formula using only the until operator expresses a property closed under stuttering. The second part of the theorem follows directly from this observation and Theorem 3.2.1.

Finally, the least expressive class of LTL safety properties presented in the context of this thesis is the one of *strong safety properties.*

Definition 3.2.2: *Every positive formula* φ *using only* $\mathbf{G}$ *as temporal operator, is called a* strong safety formula.

In other words, if $p \in AP$ represents something "bad" it follows that $\mathbf{G}\neg p$ is a strong safety formula.

Another important argument for the syntactic classification of safety formulae appears in light of the complexity involved to decide, in the general case, whether a given formula $\varphi \in$ LTL specifies a safety property, or not. The problem of deciding whether φ expresses a safety property is known to be PSPACE-complete in the size of the formula (Kupferman and Vardi [2001]). It basically involves constructing a corresponding (Büchi) automaton, $\mathcal{A}$, which is known to be worst-case exponential in the size of the formula, and then performing an adapted test for emptiness; that is, one is interested to know whether there exists an accepting run in $\mathcal{A}$ for all finite prefixes such that the property is eventually satisfiable by an extension of the prefixes. Clearly, a less complex and for the user also more intuitive categorisation is the previous one based on the term syntax alone.

3.2.2 Liveness properties

Intuitively, a liveness property asserts that *eventually* something "good" happens (during a system's execution). There exist, just as in the case of safety languages, (see §3.2.1), various shades of liveness properties (cf. Sistla [1994], Gärtner [2001]). In the context of this thesis, a *liveness language*, shall be defined according to Alpern and Schneider [1984] as follows.

Definition 3.2.3: *Let* $L \subseteq \Sigma^\omega$ *be a language on infinite strings over alphabet* Σ*.* L *is called a* liveness language, *if and only if for all finite prefixes* $u \in \Sigma^*$ *the following holds:* $\exists w \in \Sigma^\omega : uw \in L$ *(then also called a* liveness property*).*

From the definition it follows that to detect violations of liveness properties infinite strings (of actions) are required. For this reason, liveness properties are generally considered unsuited for runtime verification, where only finite strings of actions are at hand, i. e., the observations from the start of a system until the last action observed. However, the definition also implies that any superset of a liveness property is, again, a liveness property.

Fairness

This last observation leads to the concept of *fairness*, which is important for many verification tasks, such as model checking communication protocols (cf. Holzmann [1991]). A typical example of fairness is a recurring request-acknowledgement pattern. Generally, there are two types of fairness, strong and weak. Let $p_i, q_i \in AP$, a *weak fairness* property is then expressed by the following LTL formula

$$\bigwedge_{1\leq i}(\mathbf{FG}p_i \Rightarrow \mathbf{GF}q_i) \quad \text{which is, in fact, equivalent to} \quad \bigwedge_{1\leq i}(\mathbf{GF}(\neg p_i \vee q_i)).$$

It states that certain conditions are true infinitely often. In contrast, *strong fairness* asserts that if certain conditions are true infinitely often, then certain other conditions must also be true infinitely often. Strong fairness is captured by the following LTL-pattern:

$$\bigwedge_{1\leq i}(\mathbf{GF}p_i \Rightarrow \mathbf{GF}q_i).$$

The aforementioned request-acknowledgement sequence of a communication protocol belongs to the latter category of fairness. Notice, all fairness properties are also liveness properties, since counterexamples are generally infinite.

Bounded liveness

In practice, there exist properties with infinite counterexamples that are being checked in bounded time and space. These properties are the so-called *bounded (liveness) properties*. For example, consider the requirement that a process eventually terminates, expressed in terms of LTL as $\varphi \equiv \mathbf{F}\mathit{terminates}$, where $\mathit{terminates} \in AP$. Reactive systems, as they are the subject of this thesis, typically, do not terminate; they operate infinitely long on stimuli provided by their environment. Hence, monitoring φ over the standard infinite-trace semantics of LTL, using only finite behavioural strings would be impossible in such a setting. On the other hand, in a different setting, it may be possible to be more specific, and one can search for possible violations of this requirement within a certain bound, i. e., the process must terminate within 30 steps of recurring observations.

So-called *bounded model checking* is based on this idea and uses a *bounded semantics of LTL* which safely under-approximates the standard semantics. It allows the use of a prefix $u = a_0a_1 \ldots a_k$ of an infinite string $w = uw' \in \Sigma^\omega$ to check the formula. Biere et al. [2003] have shown that if a formula φ holds in the bounded semantics, denoted

$w \models_k \varphi$, where k is the bound, it also implies that $w \models \varphi$. For a good overview on bounded model checking, see also Biere et al. [2003], and Clarke et al. [2004].

Property decomposition

Based on previous findings by Schneider [1987], it is shown in a subsequent work by Alpern and Schneider [1987] that any property on infinite strings can be decomposed into a safety property and a liveness property whose conjunction is the original.

Theorem 3.2.2 (Decomposition theorem): *For any property P defined over infinite strings $w \in \Sigma^\omega$ there exists a safety property P_{safe} and a liveness property P_{live} (both over Σ^ω), such that $P = P_{safe} \cap P_{live}$.*

Proof:
The proof for the *decomposition theorem* is based on the observation that the safety properties resemble the closed sets on Σ^ω, whereas the liveness properties are the dense sets. The attributes "closed" and "dense" refer in this context to the *Cantor topology* over infinite strings in Σ^ω (cf. Finkel [2003], Finkel and Simonnet [2003]).

To relate this back to LTL and ω-automata, it is concluded that it is possible to decompose any Büchi automaton, $\mathcal{A}$, into the automata, $\mathcal{A}_{safe}$ and $\mathcal{A}_{live}$, such that the set of strings accepted by $\mathcal{A}_{safe}$ is a safety property and the set of strings accepted by $\mathcal{A}_{live}$ is a liveness property.

3.3 SALT—Structured assertion language for temporal logic

3.3.1 Motivation

Temporal logic as introduced in the previous sections is a specification formalism suited to express desired properties of a set of traces and comes with rigorous semantics. Importantly, automatic verification techniques, such as model checking, are successfully used to verify such specifications over finite state system models.

However, despite obvious advantages over semi-formal and informal notations, temporal logic is often disregarded in industrial practice; the exception being the hardware verification domain (see §3). Instead, a considerable amount of verification related questions are answered only partially by means of testing and simulation with well-known drawbacks (see §2.2). Temporal logic, on the other hand, is still widely considered to be a vehicle for specially skilled verification engineers, if not even an "academic toy" to some.

In this section it is argued that the latter point of view is misleading. It is admitted, however, that, for example, LTL's syntax—together with the typical reduction to a minimal set of operators which is done in most research papers—makes it additionally hard for formulating concise and correct specifications, even for specialists.

For example, consider the simple requirement "s precedes p after q", which is formulated in LTL by Dwyer et al. [1999] as $(\mathbf{G}\neg q) \vee \mathbf{F}(q \wedge (\neg p \mathbf{U}_w s))$. At first sight, this looks correct: "either q never holds or, when q becomes true, there is no p before an s". Nevertheless, the formula contains a subtle error: it states that *eventually* $q \wedge (\neg p \mathbf{U}_w s)$ holds, but does not require it to be the first occurrence of q. The sequence $qpqs$ satisfies the formula, although it is clear that it should not. Consequently, the correct formula would be $(\mathbf{G}\neg q) \vee \neg q \mathbf{U}(q \wedge (\neg p \mathbf{U}_w s))$. Avoiding this kind of mistake in specifications altogether is practically impossible. LTL's minimalistic set of operators, however, forces its users to build complex, error-prone formulae for even very simple requirements as can be seen in this example.

It is very unlikely, however, that a completely different temporal specification formalism—of whatever kind—would stand a chance to compete with LTL (and its derivatives, for that matter) for at least two different reasons:

1. LTL has well-accepted and precise semantics,
2. powerful model checking tools and runtime verification approaches based on LTL exist already.

To address these problems, SALT is introduced in this section, whose name is an acronym for *Structured Assertion Language for Temporal Logic*. SALT was first described in greater detail by Bauer et al. [2006c], and a realisation in terms of an optimising compiler was given by Streit [2006]. The latter contains a detailed discussion of the translation process of SALT specifications as well as its realisation in terms of an optimising compiler, now available from `http://salt.in.tum.de/` under an open source license. Note that the author of this thesis co-supervised the work of Streit.

To programmers, SALT looks similar to a general-purpose imperative programming language, while still being translatable to LTL, or, in case real-time operators are used, to TLTL (see §4.5.1). As such, SALT is not only suitable as a front end to the runtime reflection framework, but also to already existing model checking and runtime verification tools which are based on standard LTL (e. g., SPIN or SMV). More importantly, being syntactically close to a general-purpose language, SALT is, as the examples throughout this section will show, more intuitive to use and understand than pure LTL. For example, besides LTL's temporal operators, SALT provides scoping rules, support for (limited) regular expressions, exceptions, iterators, counting quantifiers, and user-defined macros. In other words, using SALT, one is able to specify properties on a higher level of abstraction than with many other formalisms, such as standard LTL and derivatives.

While compiling a high-level programming language to a more low level representation often has a negative impact in terms of efficiency (see, e. g., Bauer [2004] for a discussion of this problem in light of efficiently compiling high-level functional code), it can be demonstrated that LTL (respectively TLTL) formulae resulting from SALT specifications tend to be considerably compact when compared to their manually written counterparts in LTL; one reason lies in that humans tend to choose the most readable formula among equivalent ones, while the SALT compiler can optimise solely for the size of a formula.

Although, SALT's syntax is exemplified by various practical examples throughout the following sections, its formal syntax is not reprinted in full length as it is also available in terms of a context-free grammar from Streit [2006]. A detailed discussion concerning the formal semantics, however, follows in §3.3.4 and is supplemented by Appendix A, which contains details regarding the translation of the language.

3.3.2 Classification

In the following, the broader context of SALT is detailed upon in terms of a discussion regarding other well-known approaches with similar goals in mind (that is, providing a more high-level abstraction for temporal logic and, in particular, LTL), and secondly, in terms of sketching the overall expressiveness of SALT.

Related approaches

Plain LTL's limited flexibility in real-world scenarios has also been noted by other authors. For various domains, such as hardware design and verification, the use of more high-level and abstract specification languages has become an established means for reasoning about system properties. The success of some of these methods manifests itself in various standardisation efforts undertaken by the IEEE. For brevity, only some of the established approaches are outlined below.

Sugar/PSL. SUGAR/PSL (Property Specification Language) (Beer et al. [2001]) is a high-level specification language tailored for hardware design, originally aimed as a "syntactic sugaring" of the branching time logic CTL (Emerson and Halpern [1982]). However, from version 2.0 on, SUGAR is based on a linear view of time while keeping branching time as an optional extension. SUGAR is currently undergoing standardisation by the IEEE under the name PSL, short for Property Specification Language (cf. Foster et al. [2005]).

The PSL language is structured into Boolean, temporal, verification, and modelling *layers*. The Boolean layer provides operators for propositional logic, while the operators of the temporal layer are used to combine propositional formulae to temporal ones. The verification layer allows to define what the verification tool is expected to do with the specified properties (e. g., check that a property holds, assume that a property holds, etc.). The modelling layer, in turn, is used to model the input to the design or external hardware.

PSL provides a rich set of operators for reasoning over Boolean conditions (e. g., bit-vector operations) and for regular expressions. A so-called *clocking operator* allows to state that an expression is evaluated only in cycles where its clocking condition holds. PSL comes with an abort operator that can be used to model resets: it evaluates a pending expression to false on the occurrence of an exceptional (abort) condition. Furthermore, PSL allows the use of macro directives similar to those of the C preprocessor. Parameterised properties can be instantiated for a set of concrete values. However, PSL does not contain temporal past operators which can be rather intuitive to use as well as make specifications more succinct (see Definition 3.1.16), and no real-time constraints used frequently for modelling and verifying properties of reactive systems imposing strict execution times and deadlines, such as embedded systems.

PSL is often directly used as input to a verification tool, both for formal verification and for generating checks that are executed by a simulation tool. The latter corresponds to a runtime analysis of a simulated hardware design. However, PSL is specific to the hardware domain and a translation into LTL is possible only for a subset of PSL (cf. Tuerk and Schneider [2005]). Therefore it cannot be easily used with existing LTL-based verification tools.

PSL's goals are orthogonal to SALT: On the one hand, SALT provides a more convenient-to-use language; on the other, SALT is not dedicated to either model checking, runtime verification, or to the hardware domain. As such, SALT does not impose its own verification and modelling layer on the user. Although, SALT serves as an interface to the runtime reflection approach as outlined in this thesis.

SpecPatterns. SALT is also influenced by work of Dwyer et al. [1999], in which various real-world specifications have been analysed. Frequently used patterns have been identified and a *pattern system* for property specifications, similar to the design patterns in software engineering (Gamma et al. [1994]) has been elaborated. Basically, a pattern provides a solution to a reoccurring problem, often including notes about its advantages, drawbacks, and alternatives. As such it enables inexperienced users to reuse expert knowledge.

The patterns of Dwyer et al. consist of *requirements*, such as "absence" (i. e., a condition is false) or "response" (i. e., an event triggers another one), that can be expressed under different *scopes*, such as "globally", "before an event r", "after an event q", or "between two events r and q". The specification pattern approach has been adopted by the BANDERA SPECIFICATION LANGUAGE and a compiler that translates such specifications into LTL is part of the BANDERA system (Corbett et al. [2001]).

Dwyer et al. convincingly argue that scopes are needed in many real-world specifications. However, specification patterns as defined by Dwyer et al. suffer from the fact that they cannot be nested: only propositional formulae may be used as their parameters. In other words, adding a new requirement to the pattern system means having to manually write an LTL formula for each scope.

Further approaches. The previous two approaches are not the only specification languages tailored for domain-specific tasks. For instance, the ForSpec Temporal Logic (FTL) (Armoni et al. [2002]) is a specification language developed at Intel, and is based on a linear view of time, aimed at the formal verification of hardware circuits. Much like Sugar/PSL, ForSpec provides regular and clocked expressions as well as accept and reject operators for modelling resets. However, ForSpec does not contain real-time operators, only limited support for references to the past, and cannot be completely translated to LTL.

EAGLE (Barringer et al. [2004]) is a temporal logic with a small but flexible set of primitives. The logic is based on recursive parameterised equations with fix-point semantics and merely three temporal operators: next-time, previous-time, and concatenation. Using these primitives, one can construct the operators known from various other formalisms, such as LTL or regular expressions. While EAGLE allows the specification of real-time constraints, it lacks many high-level constructs such as the nested scopes, exceptions, and counting quantifiers present in Salt.

Notably, Bradfield and Stevens [1998] describe a symmetric approach by providing a more low-level and formal framework in which the various different aspects of different temporal logics can be expressed. They introduce the observational mu-calculus as an "assembly language" for various extensions of temporal logic. In a follow-up paper, Bradfield et al. [2002] describe first results from an integration of the observational mu-calculus into the *Object Constraint Language* (OCL), which also forms part of the UML (cf. Warmer and Kleppe [1998]). However, the goal of this work was not to provide a more rich and natural syntax, but rather a sufficient set of temporal operators.

Expressiveness

Existing approaches as outlined above have shaped various practical considerations in the overall design rationale of the input language Salt. However, from a purely theoretical point of view, Salt's features are more oriented towards the varying expressiveness of the supported logics.

Salt supports translation into LTL, as well as TLTL, which forms the natural extension of LTL for the formulation of real-time constraints: D'Souza [2003] has shown that TLTL corresponds exactly to the first-order fragment (FO) of *monadic second order logic* (MSO) interpreted over timed words (see §4.5.1). This resembles the correspondence of LTL and first-order logic over words as shown by Kamp [1968]. However, LTL is strictly less expressive than second-order logic over words, which is expressively equivalent to ω-regular expressions. This implies that full support of regular expressions is not possible when only LTL properties are in question.

However, for practitioners, regular expressions have always been an accepted and established formalism, often used to specify custom search patterns or input tokens for programming language parsers (cf. Thompson [1968], Aho et al. [1988]). Therefore,

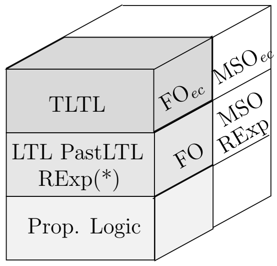


Fig. 3.4: Relationships between propositional, first-order, and temporal logics.

SALT does provide support for regular expressions that do not go beyond star-free regular languages, where "star" refers to the Kleene operator (see Definition 3.1.7), and which can, therefore, be efficiently translated into LTL.

The design of the language SALT also follows a strictly layered approach, in that the language supports specifications that can be translated into either formalism depicted in Fig. 3.4. Moreover, by reflecting and differentiating between the different levels of expressiveness in the language, SALT is theoretically extensible to support other logics in the future as well.

3.3.3 Design rationale and language features

A SALT specification contains one or more assertions that, together, formulate the requirements associated with a system under scrutiny. Each assertion is translated into a separate LTL/TLTL formula, which can then be used in a model checker or the runtime reflection framework. SALT uses mainly textual operators, so that the frequently used LTL formula $\mathbf{G}(p \Rightarrow \mathbf{F}q)$, encoding request-acknowledgement, would be written as:

```
assert always (p implies eventually q).
```

Basically, the SALT language consists of the following three layers, each covering different aspects of the specification:

The propositional layer provides the atomic, Boolean propositions as well as the well-known Boolean operators.

The temporal layer encapsulates the main features of the SALT language for specifying temporal system properties. The layer is divided into a future fragment and a symmetrical past fragment.

The timed layer adds real-time constraints to the language. It is equally divided into a future and a past fragment, similar to the temporal layer.

Within each layer, macros and parameterised expressions can be defined and instantiated by iteration operators, enlarging the expressiveness of each layer into the orthogonal dimension of functions. Depending on which layers are used for specification, the SALT compiler generates either LTL or TLTL formulae (each, with or without past operators). For instance, if only operators from the propositional layer are used, the resulting formulae are purely propositional formulae. If only operators from the temporal and the propositional layer are used, the resulting formulae are LTL formulae, whereas if the timed layer is used, the resulting formulae are TLTL formulae.

Propositional layer

Atomic propositions. Boolean propositions are the atomic elements from which SALT expressions are built. They usually resemble variables, signals, or complete expressions of the system under scrutiny. SALT is parameterised with respect to the propositional layer: any term that evaluates to either *true* or *false* can be used as atomic proposition. For example, this allows propositions to be Java expressions when used for runtime reflection of Java programs, or simple bit-vectors when SALT is used as a front end to verification tools like SPIN or SMV.

Every identifier that is used in the specification and that was not defined as a macro or a formal parameter is treated as an atomic proposition, which means that it appears in the output as it has been written in the specification. Additionally, arbitrary strings can be used as atomic propositions. For example,

```
assert always "state!=ERROR"
```

is a valid SALT specification and, using the SALT compiler, results in the output

`LTLSPEC G state!=ERROR`,

here, in SMV-syntax.

Unlike other approaches such as SUGAR/PSL, SALT does not make any assumptions regarding the validity or consistency of used propositions. However, the realisation (see §6) allows the use of external parsers that are able to perform additional checks on the specification terms used.

Boolean operators. The well-known set of Boolean operators op $= \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$ can be used in SALT both as symbols (`|`, `&`, `->`, `<->`), or as textual operators (`or`, `and`, `implies`, `equals`). Negation is denoted in SALT as `!`.

Additionally, the conditional operators `if-then` and `if-then-else` can be used, which appear similarly in the FORSPEC language. Conditional operators tend to make specifications easier to read, because `if-then-else` constructs are familiar to programmers of almost every language. Using this operator, the introductory request-acknowledgement example could be reformulated as

```
assert always (if p then eventually q).
```

More so, any such formula can be arbitrarily combined using the Boolean connectives given by the set op.

Temporal layer

To support all aspects of modelling with temporal logic, the temporal layer of SALT consists of a future and a past fragment. However, in the following description of the temporal layer, only the future fragment of SALT is presented, for reasons explained earlier in Definition 3.1.16. The past fragment, however, is completely symmetrical to the future fragment. SALT future operators are translated using only LTL future operators, and past operators are translated using only LTL past operators. This leaves users the complete freedom as to whether they do or do not want to have past operators in the result.

Standard operators. SALT provides the common LTL operators $\mathbf{U}$, $\mathbf{U}_w$, $\mathbf{R}$, $\mathbf{G}$, $\mathbf{F}$ and $\mathbf{X}$, written as `until`, `until weak`, `releases`, `always`, `eventually`, and `next`. Thus, untimed SALT (i. e., SALT without real-time operators) has exactly the same expressiveness as standard LTL.

Derived and extended operators. Similarly to SUGAR/PSL, SALT also provides a number of extended operators that help express frequently used requirements.

- `never`: The `never` operator is dual to `always` and requires that a formula never holds. While this could of course be easily expressed with the standard LTL operators, using `never`, can help make specifications easier to understand.
- Extended `until`: SALT provides an extended version of the LTL $\mathbf{U}$ operator. The user can specify whether she or he wants it to be *exclusive* (i. e., in $\varphi\mathbf{U}\psi$, φ has to hold until the moment ψ occurs) or *inclusive* (i. e., φ has to hold until and during the moment ψ occurs).[1]They can also choose whether the end condition is *required* (i. e., must eventually occur), *weak* (i. e., may or may not occur), or *optional* (i. e., the expression is only considered if the end condition eventually occurs). The `until` operator family of SUGAR/PSL provides a similar choice between inclusive/exclusive and weak/strong end conditions.
- Extended `next`. Instead of writing long chains of `next` operators, users of SALT can specify directly that they want a formula to hold at a certain step in the future. It is also possible to use the extended `next` operator with an interval, e. g., specifying that a formula has to hold at some time between three and six steps in the future. Note that this operator refers only to states at certain positions in the sequence, not to real-time constraints.

[1]This has nothing to do with strict or non-strict $\mathbf{U}$: strictness refers to whether the present state (i. e., the left end of the interval where φ is required to hold) is included or not in the evaluation (Fisher [1991]), while inclusive/exclusive defines whether φ has to hold in the state where ψ occurs (i. e., the right end of the interval). Strict SALT operators can be created by adding a preceding `next`-operator.

Counting quantifiers. SALT provides two operators, `occurring` and `holding`, that allow events that have to occur a certain number of times to be specified. `occurring` deals with events that may last more than one step and are separated by one or more steps in which the condition does not hold. `holding` considers single steps in which a condition holds. Both operators can also be used with an interval, e. g., expressing the fact that some condition has to be true *at most* during two steps in the future. To express this requirement manually in LTL, one would have to write

$$\neg p \mathbf{U}_w (p \mathbf{U}_w (\neg p \mathbf{U}_w (p \mathbf{U}_w \mathbf{G} \neg p))).$$

The corresponding SALT specification is written as

```
assert occurring[<=2] p.
```

Exceptions. SALT includes the exception operators `rejecton` and `accepton` that interrupt the evaluation of a formula upon occurrence of an abort condition. `rejecton` evaluates a formula to false if the abort condition occurs and the formula has not been accepted before. For example, monitoring a formula $\mathbf{F}\varphi$ when there has been no occurrence of φ yet would evaluate to false. The dual operator, `accepton`, evaluates a formula to true if it has not been rejected before.

Exceptions can be useful, e. g., when specifying a communication protocol that requires certain messages to be sent, but allows the communication to be aborted at any time by sending a reset message. This would be expressed in SALT as

```
assert (con_open and next (data until con_close)) accepton reset.
```

Similar `rejecton` and `accepton` operators can be found in FORSPEC and in PSL. The formal semantics of LTL enriched with those two operators (called Reset-LTL) was explored in detail by Armoni et al. [2003].

Scope operators. Many temporal specifications contain requirements restricted to a certain *scope*, i. e., they state that the requirement has to hold only before, after, or between some events, and not on the whole sequence. This can be expressed in SALT using the operators `upto` (or `before`), `from` (or `after`) and `between`.

Fig. 3.5 illustrates scopes. In the figure, it can be seen that it is mandatory in SALT to specify whether the delimiting events are part of the interval (i. e., *inclusive*) or not (i. e., *exclusive)*. Furthermore it has to be stated whether the occurrence of the delimiting events is strictly required.

Scopes have been identified by Dwyer et al. [1999] as an important issue in the specification pattern system and, consequently, the BANDERA language. However, their pattern system is restricted to predefined requirements. It does not allow nested scopes, and by default only certain combinations of inclusive/exclusive and required/optional delimiters. A subset of scopes can also be expressed in SUGAR/PSL using the `next_event` and `before` operators. SALT's distinguishing feature here is

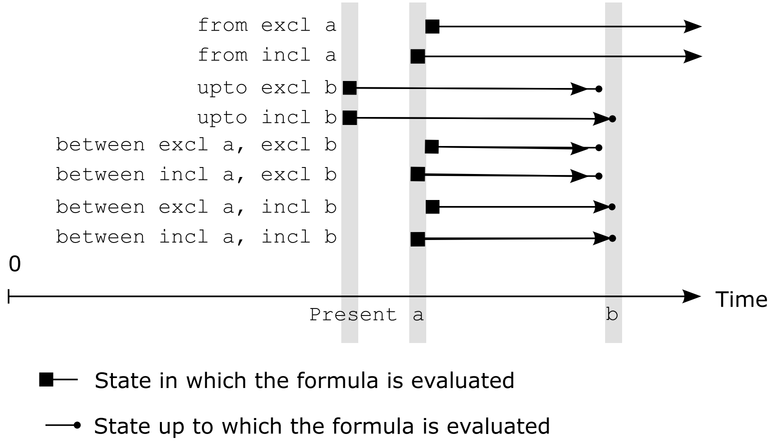


Fig. 3.5: Scopes of `upto`, `from` and `between`.

that scope operators can be used with arbitrary formulae, even with nested scope operators.

While it is relatively straightforward to implement a translation of the `from` operator into LTL, the `upto` operator proves to be more difficult, as can be seen in the following example.

A specification `always` φ `upto` b expresses that φ must always hold until the occurrence of the end condition b. A naïve translation into LTL would be $\varphi \mathbf{U}_w b$. This is in order for a purely propositional φ, but might be wrong when temporal operators are used: Consider for example $\varphi \equiv$ `p -> (eventually s)` yielding the formula $(p \Rightarrow \mathbf{F}s)\mathbf{U}_w b$, intending to say "$p$ should be followed by s before b". The sequence pbs is a model for the latter formula, although s occurs after the end condition b, which violates the intentional meaning. To meet the intentional meaning, the negated end condition b has to be inserted into the $\mathbf{U}$ and $\mathbf{X}$ statements of φ in various places, e. g., like this: $(p \Rightarrow (\neg b\mathbf{U}(\neg b \wedge s)))\mathbf{U}_w b$. Dwyer et al. [1999] describe this procedure in the notes of their specification pattern system. It is, however, a tedious and highly error-prone task if undertaken manually.

Salt supports automatic translation by internally defining a stop operator. With this operator, the above example can be formulated as $((p \Rightarrow \mathbf{F}s)$ stop $b)\mathbf{U}_w b$ with stop b expressing that $(p \Rightarrow \mathbf{F}s)$ shall not take into account states *after* the occurrence of b. It is then transformed into an LTL expression in a similar way as the `rejecton` and `accepton` operators. For details on this translation, see §3.3.4.

Regular expressions. Besides their use in programming and compilers, regular expressions provide a convenient way to express complex patterns of events, and appear also in many specification languages, e. g., such as Sugar/PSL. However, arbitrary regular languages can be defined using regular expressions, while LTL only allows the definition of so-called *star-free* languages. Thus, regular expressions have to be restricted to be usable in SALT.

The SALT regular expressions provide concatenation (`;`), union (`|`), and Kleene-star operators (`*`), but no complement. The argument of the Kleene-star is required to be a propositional formula. The advantage of this operator set (in contrast to the usual operator set for star-free regular expressions, which contains concatenation, union and complement) is that it can be translated efficiently into LTL. Note in the case of SUGAR/PSL, which also provides regular expressions without a complement operator, it is argued that many relevant properties which are interesting for verification can be expressed conveniently without it (cf. Beer et al. [2001]).

Additionally, SALT provides operators that do not increase the expressiveness of its regular expressions, but makes dealing with them more convenient for users. The overlapping sequence operator `:` is inspired by SUGAR/PSL and states that one expression follows another one, overlapping in one step. The `?` and `+` operators (optional expression and repetition at least once) are common extensions of regular expressions. The `*` operator extended with a range of natural numbers allows to specify that an expression has to hold at least, at most, exactly, or in between n and m times.

Traditional regular expressions, as used in programming and for compilers, match finite sequences (see Definition 3.1.7). In contrast, a SALT regular expression holds on a (possibly) infinite sequence, if it matches a finite prefix of the sequence.

With the help of (ω-) regular expressions, the example using exception operators, enclosed in dashes, can be reformulated as

```
assert /con_open; data*; con_close/ accepton reset.
```

Timed layer

SALT contains a timed extension that allows the specification of real-time constraints that go beyond the expressiveness of plain LTL. Basically, timed operators are translated into the logic TLTL, a timed variant of LTL introduced formally in §4.5.1 which constitutes the foundation for the dense-time runtime verification approach developed in this thesis.

Timing constraints in SALT are expressed using the modifier `timed[`$\sim$`]`, which can be used together with several untimed SALT operators in order to turn them into timed operators. $\sim$ is one of `<`, `<=`, `=`, `>=`, `>` for `next timed` and either `<` or `<=` for all other timed operators.

- `next timed[`$\sim c$`]`φ
  states that the next occurrence of φ is within the time bounds $\sim c$. This corresponds to the operator $\rhd[\sim c]\varphi$ in TLTL.
- φ `until timed[`$\sim c$`]` ψ
  states that φ is true until the next occurrence of ψ, and that this occurrence of

ψ is within the time bounds $\sim c$. The extended variants of `until` can be used as timed operators as well.

- `always timed[`$\sim c$`]` φ
  states that φ must always be true within the time bounds $\sim c$.
- `never timed[`$\sim c$`]` φ
  states that φ must never be true within the time bounds $\sim c$.
- `eventually timed[`$\sim c$`]` φ
  states that φ must be true at some point within the time bounds $\sim c$.

At this point, more detail regarding real-time specifications is not provided. The next chapter extensively deals with this topic.

Macros and parameterised expressions

SALT supports user-defined sub-expressions as *macros* and to parameterise macros and sub-expressions. Macros can be called in the same way as built-in SALT operators. Within certain limits, this allows users to extend the SALT language using their own operators. For example, the following macro is called in infix notation:

```
define respondsto(x, y) := y implies eventually x
assert always (reply respondsto request)
```

Iteration operators allow the instantiations of a parameterised sub-expression or macro with a list of values provided by the user. For example, the following specification states that either `a` or `!b` or `c` must hold forever.

```
assert someof list [a, !b, c] as i in always i
```

Parameters defined in a macro or an iteration expression can also be used to parameterise Boolean variables, as in the following example, which states that exactly one of the four variables, `state_1`, `state_2`, `state_3` and `state_4`, must be true.

```
assert exactlyoneof enumerate[1..4] as i in state_$i$
```

Macros can help to make a specification easier to understand, because complicated sub-expressions can be transparently hidden from the user, and accessed via an intuitive name that explains what the expression actually stands for. Sub-expressions that are used several times have to be written down only once.

3.3.4 Formal semantics

As outlined in §3.3.2, SALT can be translated into either LTL or TLTL; the latter only when timed operators are used in a specification. Therefore, some of SALT can be considered purely as syntactic sugaring and its formal semantics defined in terms of an inductive translation to either formalism. This translation is described in the following for some of SALT's operators, where a reductive transformation function $\mathcal{T}$ is defined to transform a SALT expression ψ into a temporal logic formula $\mathcal{T}(\psi)$, such that for every infinite word w from an alphabet of actions, $w \models \psi \Leftrightarrow w \models \mathcal{T}(\psi)$. In what follows, let ψ, φ, and φ' denote SALT specifications.

For example, the operator `never` is then translated as $\mathcal{T}(\texttt{never}\ \varphi) = \neg\mathbf{F}\mathcal{T}(\varphi)$, whereas a weak inclusive until as in φ_1 `until incl weak` φ_2 is then defined as

$$\mathcal{T}(\varphi_1\ \texttt{until incl weak}\ \varphi_2) = \mathcal{T}(\varphi_1)\ \mathbf{U}_w\ (\mathcal{T}(\varphi_1) \wedge \mathcal{T}(\varphi_2)).$$

However, not all SALT operators translate in such a straightforward inductive manner, since their translation depends on what is defined by the according sub-formulae occurring in a given expression. To guide the translation process for such operators, an artificial or helper operator, `stop`, is introduced which is inductively defined by $\mathcal{T}$ as follows:

$$\begin{aligned}
\mathcal{T}(b \text{ stop}_{\text{excl}}\ s) &= b \\
\mathcal{T}((\neg\varphi) \text{ stop}_{\text{excl}}\ s) &= \neg\mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \\
\mathcal{T}((\varphi \wedge \psi) \text{ stop}_{\text{excl}}\ s) &= \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \wedge \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s) \\
\mathcal{T}((\varphi \vee \psi) \text{ stop}_{\text{excl}}\ s) &= \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \vee \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s) \\
\mathcal{T}((\varphi\ \mathbf{U}\ \psi) \text{ stop}_{\text{excl}}\ s) &= (\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s))\ \mathbf{U}\ (\neg s \wedge \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s)) \\
\mathcal{T}((\varphi\ \mathbf{U}_w\ \psi) \text{ stop}_{\text{excl}}\ s) &= \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)\ \mathbf{U}_w\ (s \vee \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s)) \\
\mathcal{T}((\mathbf{X}\varphi) \text{ stop}_{\text{excl}}\ s) &= \mathbf{X}(\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)) \\
\mathcal{T}((\mathbf{G}\varphi) \text{ stop}_{\text{excl}}\ s) &= \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)\ \mathbf{U}_w\ s \\
\mathcal{T}((\mathbf{F}\varphi) \text{ stop}_{\text{excl}}\ s) &= (\neg s)\ \mathbf{U}\ (\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s))
\end{aligned}$$

where b denotes an atomic proposition from the action alphabet and s a well-formed suffix, possibly being atomic also.

Thus, `stop` selects certain aspects of a formula, and in $\psi \equiv \varphi_1$ `stop` φ_2, intuitively asserts that the validity of ψ does *not* depend on events occurring after φ_2 has occurred. Again, for brevity, only the exclusive variant of `stop` is considered, and only for the future fragment of SALT. The past fragment and inclusive semantics, however, are each symmetrical.

The more complicated scope operator `upto`, which was discussed earlier in §3.3.3, and whose translation depends on `stop`, is then defined as:

$$\begin{array}{lll}
\mathcal{T}(\varphi\ \texttt{upto excl req}\ b) & & = \\
& \text{if } \mathcal{T}(\varphi) = \mathbf{G}\psi\text{:} & (\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b \\
& \text{if } \mathcal{T}(\varphi) = \neg\mathbf{F}\psi\text{:} & (\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b \\
& \text{else:} & (\mathbf{F}b) \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)
\end{array}$$

$$
\begin{array}{lll}
\mathcal{T}(\varphi\ \texttt{upto excl opt}\ b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{F}\psi: & & \neg((\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{else:} & & (\mathbf{F}b) \Rightarrow (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b) \\
\mathcal{T}(\varphi\ \texttt{upto excl weak}\ b) & = & (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b) \\
\mathcal{T}(\texttt{req}\ \varphi\ \texttt{upto excl req}\ b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{G}\psi: & & \neg b \wedge ((\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{if } \mathcal{T}(\varphi) = \neg\mathbf{F}\psi: & & \neg b \wedge ((\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{else:} & & (\mathbf{F}b) \wedge \neg b \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b) \\
\mathcal{T}(\texttt{req}\ \varphi\ \texttt{upto excl opt}\ b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{F}\psi: & & \neg((\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{else:} & & (\mathbf{F}b) \Rightarrow (\neg b \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)) \\
\mathcal{T}(\texttt{req}\ \varphi\ \texttt{upto excl weak}\ b) & = & \neg b \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b) \\
\mathcal{T}(\texttt{weak}\ \varphi\ \texttt{upto excl req}\ b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{G}\psi: & & (\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b \\
\qquad \text{if } \mathcal{T}(\varphi) = \neg\mathbf{F}\psi: & & (\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b \\
\qquad \text{else:} & & (\mathbf{F}b) \wedge (b \vee (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)) \\
\mathcal{T}(\texttt{weak}\ \varphi\ \texttt{upto excl opt}\ b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{F}\psi: & & b \vee \neg((\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{else:} & & (\mathbf{F}b) \Rightarrow (b \vee (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)) \\
\mathcal{T}(\texttt{weak}\ \varphi\ \texttt{upto excl weak}\ b) & = & b \vee (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b) \\
\mathcal{T}(\varphi\ \texttt{upto incl req}\ b) & = & (\mathbf{F}b) \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{incl}}\ b) \\
\mathcal{T}(\varphi\ \texttt{upto incl opt}\ b) & = & (\mathbf{F}b) \Rightarrow (\mathcal{T}(\varphi) \text{ stop}_{\text{incl}}\ b) \\
\mathcal{T}(\varphi\ \texttt{upto incl weak}\ b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{G}\psi: & & \neg(\neg b\ \mathbf{U}\ \neg(\psi \text{ stop}_{\text{incl}}\ b)) \\
\qquad \text{if } \mathcal{T}(\varphi) = \neg\mathbf{F}\psi: & & \neg(\neg b\ \mathbf{U}\ (\psi \text{ stop}_{\text{incl}}\ b)) \\
\qquad \text{else:} & & (\mathcal{T}(\varphi) \text{ stop}_{\text{incl}}\ b)
\end{array}
$$

where, $\text{stop}_{\text{excl}}$ and $\text{stop}_{\text{incl}}$ are references to the exclusive and inclusive variants of `stop`, respectively.

Similar translation schemes are defined for SALT's exception operators, i. e., `accepton` and `rejecton`. For details, see Appendix A.

The above translation is carried out stepwise; that is, a SALT expression is first turned into the SALT *core language* consisting of the operators `until`, `upto`, `from`, `between`, `accepton`, `rejecton`, and the set of regular expressions which can be described by using only the repetition operator in the constrained form of `*[>=`n`]`.

The second step includes translation of core SALT into the subset SALT--, which basically resembles either LTL or TLTL but using also exception operators as well as exclusive and inclusive stop-operators for both the future and past fragments. The stop-operator can be thought of as a helper which is introduced during the translation of `upto` and `between`.

Finally, SALT-- gets converted in a third step into either plain LTL or TLTL, both of which may contain past operators if originally used for specification.

3.3.5 Example specifications

In this section, a concluding look at some more SALT specifications is taken, and their corresponding LTL versions examined. The examples are mostly borrowed from the survey presented by Dwyer et al. [1999], except where indicated otherwise. Note that propositions appearing in the specifications are not necessarily marked as such and are denoted in plain text only, indicating their intuitive meaning with respect to the specification.

1. The requirement that a system should operate until a queue of jobs is either empty, or an abort signal issued can be formulated in LTL as

   $$\neg((\neg(queuelength == 0 \vee abort))\mathbf{U}(\neg working \wedge (\neg(queuelength == 0 \vee abort)))).$$

   The accompanying SALT specification would be:

   ```
   assert working until weak ("queuelength == 0" | abort).
   ```

2. To specify idle behaviour, the following LTL specification could be used:

   $$\begin{aligned}\mathbf{G}(\neg return\_Execute \vee (return\_Execute \wedge ((\mathbf{F} call\_Execute) \Rightarrow \\ (\neg(\neg call\_Execute \mathbf{U}(call\_doWork \wedge \neg call\_Execute)))))).\end{aligned}$$

   It asserts that between the moment in which an execution completes, and before a new one beings, there is no work done. In SALT, this example would be written as:

   ```
   assert always
     (never call_doWork
          between inclusive optional return_Execute,
                  exclusive optional call_Execute).
   ```

3. Coming back to the initial example from the area of protocol specification, one might assert that an answer was immediately preceded by a request. In LTL this would be written as:

   $$\mathbf{G}(answer \Rightarrow (\mathbf{X} request)).$$

   Using a macro, in SALT, `precedes` can be expressed as follows:

   ```
   define precedes(x, y) := if y then once x
   assert always (request precedes answer).
   ```

4. A system with n input channels, may be using at most one at a time. Given that $n = 4$, this simple requirement would require

$$\begin{aligned}&\mathbf{G}(((in\_0 \wedge (\neg(in\_1 \vee (in\_2 \vee in\_3))))\vee\\ &\quad((in\_1 \wedge (\neg(in\_0 \vee (in\_2 \vee in\_3))))\vee\\ &\qquad((in\_2 \wedge (\neg(in\_0 \vee (in\_1 \vee in\_3))))\vee\\ &\qquad\quad(in\_3 \wedge (\neg(in\_0 \vee (in\_1 \vee in\_2)))))))\vee\\ &\qquad\qquad(\neg(in\_0 \vee (in\_1 \vee (in\_2 \vee in\_3)))))\end{aligned}$$

if specified in LTL. The shorter SALT specification appears to be less error-prone and more readable (not only because of proper indenting):

```
assert always
          (exactlyoneof enumerate [0..3] as i in in_$i$) |
          (noneof enumerate [0..3] as i in in_$i$).
```

5. To show that regular expressions can be very useful for specification purposes, in the following it is expressed that a connection signal is eventually answered by an acknowledgement, followed by at least four data packets and a close signal. Again, this is first examined in LTL:

$$\begin{aligned}&\mathbf{G}(connection \Rightarrow\\ &\quad(\mathbf{F}(answer \wedge (\mathbf{X}(data\mathbf{U}(data\wedge\\ &\qquad(\mathbf{X}(data \wedge (\mathbf{X}(data \wedge (\mathbf{X}(data \wedge (\mathbf{X}\, close)))))))))))).\end{aligned}$$

Now, consider the SALT counterpart using a regular expression:

```
assert always (if connection then
                 eventually /answer; data*[>=4]; close/)
```

6. Reconsider the elevator from p. 45. The requirement was that between the time an elevator is called at a floor and the time it opens its doors at that floor, the elevator can arrive at that floor at most twice. In SALT, this can be specified as:

```
assert always
 (occurring[<=2] atfloor
  between inclusive optional call, exclusive optional open)
```

7. This section is now concluded by extending this example further and thus, showing most of SALT's features in one use-case. The following specification describes the following behaviour: On all three floors in a building, calling the elevator at floor i implies that it may pass at most two times at that floor without opening its doors, and that it must finally open its doors at that floor within 60 seconds.

```
define max_twice_at_floor_before_open(i) :=
```

```
 always(occurring[<=2] atfloor_$i$
        between inclusive optional call_$i$,
                                   exclusive optional open_$i$)

define max_60s_before_open(i) :=
 always (call_$i$ implies
         eventually timed[<=60.0] open_$i$)

assert allof enumerate[1..3] as floor in
             max_twice_at_floor_before_open(floor)
         and max_60s_before_open(floor)
```

The modifiers `optional` in the `between`-statement make sure that `atfloor_`i is only checked provided `call_`i occurs.

Note that the equality between the LTL specifications in the above examples and their SALT counterparts, was established using the model checker SMV. For this purpose the SALT specifications were first compiled into plain LTL using the SALT compiler (see §6) and then compared with the manually written requirements.

3.4 Summary

The first part of this chapter recalls the foundations of linear time temporal logic as a means for systems specification and verification, whereas the second part introduces the custom specification and assertion language, SALT, for creating concise temporal specifications that are intuitive to read and formulate. SALT incorporates ideas of existing approaches, such as specification patterns, but also provides nested scopes, exceptions, support for regular expressions and real-time. The latter is needed in particular for verification tasks to do with reactive systems imposing strict execution times and deadlines. However, unlike other formalisms used for temporal specification of properties, SALT does not target a specific domain; that is, it can be used with any formal verification framework, besides runtime reflection, which is based on linear time temporal logic, such as model checkers, for instance. Moreover, the chapter details on the design rationale, the syntax and semantics of SALT in terms of a translation to temporal (real-time) logic, and gives a number of example use-cases.

Chapter 4

Failure detection through runtime verification

> My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed.
>
> *(Edsger W. Dijkstra,* Go To Statement Considered Harmful*)*

This chapter introduces an automata-based approach to runtime verification using a 3-valued interpretation of temporal properties, capable of monitoring real-time specifications, e. g., for event-triggered reactive or real-time systems. Such systems usually exhibit events at arbitrary points in time which are often modelled using rational or real numbers. This is then referred to as *dense time*. However, dynamic reasoning depends naturally on the resolution of the employed interrupt timers.

Although there exist various "flavours" of runtime verification, the basic underlying idea is normally the same and can be summarised as follows. A user-provided specification of intended system behaviour is compared by an observer with the actual behaviour of a system during its execution. The observer or system monitor will then come to a verdict as to whether the system behaviour satisfies the specification. Therefore, runtime verification belongs to the class of dynamic verification methods, for which system execution is mandatory. Another dynamic method is, e. g., testing as outlined in §2. Consequently, runtime verification not only aids in establishing correctness while a system is in normal operation, after it has been fully developed and shipped, but monitors are also useful to complement traditional or model-based testing (cf. Broy et al. [2005]).

It is mainly the different approaches to systems specification which have led to the differing flavours of runtime verification. Thus, one can consider a wide spectrum of approaches, ranging from predicate assertions stating properties about single states at single system or program locations, to temporal assertions stating properties about temporally separated states at multiple program locations. The latter is clearly inspired by works of model checking temporal specifications and, at the same time, the predominant approach to formal runtime verification encountered in the literature (cf. Havelund and Goldberg [2005]).

The approach described in this chapter not only covers dense-time systems, but it also has the following properties, each of which will be explained in greater detail throughout this chapter:

minimality, i. e., behavioural violations are detected as early as possible,

predictiveness, i. e., depending on the property being monitored, violations can be detected before a violating event has occurred, (This is a direct result of minimality.)

minimal space complexity, i. e., the generated monitors are minimal with respect to the language/property being monitored,

universality, i. e., monitoring is not restricted to a particular class of properties, such as safety.

4.1 A brief history of runtime verification

When compared to the area of static verification, e. g., by means of automatic model checking, runtime verification is a relatively young scientific discipline[1] with considerably fewer publications, but nonetheless various academically as well as practically interesting approaches, some of which are outlined in the following.

Basically, the term runtime verification was coined by works of Havelund and Rosu (cf. Havelund [2000], Havelund and Rosu [2001b;a; 2002; 2004]). In their work, temporal logic descriptions in LTL are used to generate monitors for Java programs to detect deadlocks and race conditions. Their efforts resulted in various (partly commercialised) tools, such as the Java PathFinder (JPF, Havelund [1999]), PathExplorer (JPaX, Havelund and Rosu [2001a]), and the Temporal Rover (Drusinsky [2000]) tools, for instance. JPaX and other early approaches were based on Maude, an efficient specification and verification framework making implementations of rewriting logic applicable (Clavel et al. [2003]). Consequently, these approaches to runtime verification were initially not based directly upon the methods and algorithms known from the area of model checking, such as generation of Büchi automata from LTL specifications (see §3.1.1), but rather on an on-the-fly formula rewriting for separating past from present and future obligations, which then have to hold in the *next* measured instant of time (Havelund and Rosu [2001b]). This is typically achieved by conversion into a normal form (see §3.1.2) and by making use of the fix-point characterisation of the logic. Emerson and Clarke [1980] introduced a fix-point

[1]The first workshop on runtime verification was held in 2001 in Paris, France, as a satellite event of the established CAV series. See Berry et al. [2001].

characterisation of branching time temporal logic, but it transfers over to LTL in a straightforward manner as can be seen in this example:

$$\mathbf{G}\varphi \equiv \varphi \wedge \mathbf{XG}\varphi \quad \text{or} \quad \mathbf{F}\varphi \equiv \varphi \vee \mathbf{XF}\varphi.$$

The same can be formulated for the remaining temporal operators of LTL. However, for brevity this topic is not developed further at this stage; a more detailed discussion of fix-point characterisations of LTL but in the context of bounded model checking is also available, e. g., from Sheridan [2002].

Other, more recent, logic rewriting approaches for runtime verification have been spawned from this work, such as those presented by Kristoffersen et al. [2003] and Håkansson et al. [2003]. Both works distinguish themselves by reflecting real-time properties using so-called *freeze quantification* (Alur and Henzinger [1989]), which basically is the addition of a (discrete) notion of time and a quantifier to standard LTL whose purpose is to "freeze" the current time instant upon evaluation of a quantified formula. For this purpose, the latter uses the notion of a so-called *disjunctive equation normal form* to guide its rewriting process. Notice, however, that the logic introduced by Alur and Henzinger [1989] is generally undecidable when considering a time domain other than discrete natural numbers.

As an alternative to rewriting, Giannakopoulou and Havelund [2001] soon tried to achieve the same results by using Büchi automata for the monitor generation. In the cited work, the authors addressed in particular the problem of dealing with LTL's infinite trace semantics in this context; that is, they paid respect to the fact that certain properties which could be specified using LTL could not be monitored in the traditional interpretation of LTL formulae when at most a finite history of system events is at hand. Their solution to this problem was to redefine the temporal operators of LTL, such that they can be evaluated over finite traces only, but on the expense of violating LTL's original semantics for some operators, as is the case with the until-operator: Let $u = a_0 a_1 \dots a_n \in \Sigma^*$ be a finite trace of system behaviours then the until-operator is defined as

$$u, 0 \models \varphi_1 \mathbf{U} \varphi_2 \Leftrightarrow \exists i \leq n : (i \geq 0 \wedge u, i \models \varphi_2 \wedge \forall l : (0 \leq l < i \Rightarrow u, l \models \varphi_1)).$$

Notice the difference in semantics! The interpretation over finite words yields wrong results if there exists a position $n' > n$ such that $u, n' \models \varphi_2 \wedge \forall l : (0 \leq l < n' \Rightarrow u, l \models \varphi_1))$; that is, the finite interpretation would yield *false* as result here, thus contradicting the standard semantics yielding *true* in this case.

Such a contradiction could lead to confusing results when systems actually adhere to their specification by showing the right behaviour in some future instant, such that a current prefix of recorded behaviours does not suffice to formally establish correctness according to the above criterion.

Moreover, Giannakopoulou and Havelund [2001] also used a next-free variant of LTL,

LTL-X. They argued that the next-operator attributes a set concept of time to the (untimed) verification process, which can be counterintuitive for some systems and specifications. Effectively, they avoid having to deal with a "next-obligation" at the end of a trace. The semantics suggested by Havelund and Rosu [2001b] gets around such compromises by simply assuming an infinite extension of the last system action observed; that is, if a last action seen is p, it would satisfy an assertion $\mathbf{X}p$. In this context, this is referred to as *stationary semantics*.

Further related approaches. Despite the fact that runtime verification as presented above is an emerging scientific discipline, the problem addressed by it has already been picked up earlier in other contexts and with modifications in the employed setup. For instance, in software and systems testing often external observers or *monitors* are employed for the online analysis of test runs (cf. Dustin et al. [1999], Pretschner [2003], Broy et al. [2005]). In the area of *synchronous systems* development (cf. Benveniste et al. [2003]), monitors are often developed in parallel with (or generated from) the actual application under scrutiny. In a nutshell, a synchronous system resembles a special class of a system which adheres to the *hypothesis of perfect synchrony*; that is, all internal computations occur instantaneously, i. e., take no time, and the results are produced at the same time inputs are received. In consequence, communication time over a network medium must also be abstracted from, in that it has to happen infinitely fast in the synchronous model (see also §4.5). Synchronous systems are usually developed in dedicated languages such as ESTEREL (Berry [1999]), LUSTRE (Halbwachs et al. [1991]), or even visually-based ones as realised, e. g., by AUTOFOCUS (cf. Huber et al. [1996], Broy et al. [1999], Bauer et al. [2005]). AUTOFOCUS in turn is formally based upon a discretely timed, synchronous subset of the more general FOCUS framework as described by Broy and Stølen [2001].

However, since these approaches do not play a major role in the remainder of this thesis, they are just referred to at this point for the sake of keeping this thesis self-contained. For instance, Halbwachs et al. [1994] first discussed the automatic generation of synchronous observers that can be executed in parallel with a synchronous application in order to detect behavioural aberrations. Their work was later picked up and extended by various authors, e. g., Westhead and Nadjm-Tehrani [1996] and Laurent et al. [2001] who applied a similar technique to the verification of a real-world avionics control system.

4.2 LTL over finite words

Many runtime verification approaches known from the literature suffer from a lack of unambiguous or intuitive semantics when interpreting specifications formulated in a temporal logic like LTL over finite traces of behaviours. The previous sections of this thesis have demonstrated that the reason lies, foremost, in the standard semantics for LTL, which asserts the validity of a property based on infinite traces, rather than prefixes.

In consequence (and besides the previously outlined approaches), various authors have settled with the following, custom interpretations of LTL over finite traces using *weak* and *strong semantics*: the weak interpretation of a formula $\varphi \in$ LTL with respect to a finite trace, denoted as u, is that if up to the point where u ends, "nothing has yet gone wrong", φ holds. In the strong view, φ holds only if it evaluates to *true* within u. Besides, there exists a *neutral view on LTL* which, basically, resembles the classical view, but uses a different interpretation for weak and strong operators and is otherwise analogue. Eisner et al. [2003] give a good overview on the topic, and discuss the benefits and drawbacks of either interpretation.

Good examples can be found for each of the interpretations and at the same time also examples that such approaches can be misleading. For instance, consider the property $\mathbf{G}\neg p$ stating that no state satisfying p must ever occur. Formally, when p is observed, a runtime monitor should raise an alarm. But as long as p does not hold, it is misleading to say that the formula is *true*, since the next observation might already violate the formula. On the other hand, consider the formula $\neg p \mathbf{U} \mathit{init}$ stating that nothing bad (i. e., p) should happen before an initialisation function, `init`, is called. If, indeed, the `init` function has been called and no p has been observed before, the formula is *true*, regardless as to what will happen in the future. For testing and verification, it is important to know whether some property is, indeed, *true* or whether the current observation is just inconclusive.

More problematic, however, is the fact that in the various finite trace interpretations discussed above, subtle *inconsistencies* can arise.

Definition 4.2.1: *A logic L is called* consistent *if there exists not a model $\mathcal{M}$ and a formula $\varphi \in L$, such that $\mathcal{M} \models \varphi$ and $\mathcal{M} \models \neg\varphi$.*

To illustrate the problem, consider the LTL formulae $\varphi \equiv \mathbf{X}p$ and $\varphi' \equiv \mathbf{X}\neg p$. Let $u \in \Sigma^*$ be a finite trace consisting of only a single element, such that $u = a_0$ with $a_0 = \{p\}$. Theoretically, in a finite trace interpretation of LTL, we could have for both $u \not\models \varphi$ and $u \not\models \varphi'$, since there is no successor action, u^1, available to satisfy either case. Vice versa, if $\mathbf{X}$ is defined differently, i. e., using a weak interpretation, one might get $u \models \varphi$ and $u \models \varphi'$ for the same reason, since there is no indication that within u the formula cannot hold.

Methodological implications. Besides a lack of unambiguous and intuitive semantics, the custom interpretation of LTL formulae is questionable from a methodological point of view as well: standard LTL is defined with rigorous, infinite trace semantics which is implemented in various verification tools, such as model checkers (see §3). Although model checking LTL properties over finite-state systems has been greatly automated over the years, these tools are still often used by verification experts only, who also have a common understanding of the underlying semantics of the tool and, in the present case, LTL.

The fact that almost each runtime verification tool currently provides its own restrictions regarding the expressiveness of LTL and the interpretation over finite traces thus hinders even (or especially) verification experts from using such a tool. None behaves like the other.

As the various examples given in this and the previous section have shown, it is even possible to define properties in a runtime verification tool that evaluate to *false*, although a model checker would yield *true* (see, e. g., the finite variant of the until-operator). Hence, the semantics of such approaches is not only unusual, but often even a contradiction to the standard semantics defined over infinite traces. In such cases, however, not only the established semantics is violated but also the intuition of the person specifying a certain temporal property. And, obviously, debugging systems and their specifications can be tedious in such a setup.

4.3 A 3-valued semantics for LTL—LTL$_3$

Mainly to overcome the difficulties in defining an adequate Boolean semantics for LTL on finite traces, this section introduces a 3-valued semantics, which naturally extends the standard infinite trace semantics. Let $w \in \Sigma^\omega$ be an infinite trace of actions, and $\varphi \in$ LTL. Naturally, in the Boolean semantics, either $w \models \varphi$, or not. However, on a truncated trace, $u \in \Sigma^*$, where $w = u\sigma$ and $\sigma \in \Sigma^\omega$, this should intuitively hold for u if and only if for all extensions, σ, $w \models \varphi$. On the other hand, if for all extensions of u, denoted σ, it holds that $w \not\models \varphi$, it is easy to see that $u \not\models \varphi$ should hold. For example, consider a simple propositional formula, $\varphi \equiv p$, which is satisfied by the finite trace consisting only of the element $a_0 = \{p\}$—regardless of any possible continuation. Hence, $u = a_0$ is, indeed, sufficient to satisfy φ, and would be clearly sufficient to falsify it, had φ been defined as $\neg p$.

The 3-valued semantics introduced in this section adds to this scheme by offering a third value, ?, to denote *inconclusive*, capturing all the remaining cases, where extensions of u, denoted σ, are not yet known. In other words, when a finite trace u is *insufficient* to satisfy or falsify a formula, as is the case with future obligations like $\mathbf{F}p$ where p has not yet occurred, its truth value with respect to the trace is denoted by the symbol ?.

Formally, the 3-valued semantics of LTL is defined and referred to in terms of LTL$_3$ over the set of truth values $\mathbb{B}_3 = \{\bot, ?, \top\}$ as follows. Notice, for the purpose of this thesis no particular ordering over the values in $\mathbb{B}_3$ needs to be defined.

Definition 4.3.1: *Let $u \in \Sigma^*$ denote a finite trace of actions. The* truth value *of an LTL$_3$ formula φ with respect to u, denoted by $[u \models \varphi]$, is an element of $\mathbb{B}_3$ and* *defined as follows:*

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases}$$

Essentially, using a 3-valued semantics, it is possible to circumvent the previously described inconsistency of Boolean semantics over finite traces in a formula such as $\mathbf{X}p$ when only one observation is at hand; that is, given $u = a_0$ where $a_0 = \{p\}$, one has $[u \models \mathbf{X}p] = ?$ and $[u \models \mathbf{X}\neg p] = ?$.

4.4 A dynamic decision procedure for LTL_3

In this section, a dynamic automata-based decision procedure for LTL_3 is developed. More specifically, for a given formula $\varphi \in \text{LTL}_3$, a finite *Moore machine*, $\bar{A}^\varphi$, is constructed that reads finite traces $u \in \Sigma^*$, and outputs $[u \models \varphi]$, thus, a value in $\mathbb{B}_3$.

Definition 4.4.1: *A (nondeterministic)* Moore machine *over Σ is a tuple $\bar{A} = (\Sigma, Q, Q_0, \delta, \lambda)$, where*

- *Σ is an input alphabet,*
- *Q a finite set of states,*
- *$Q_0 \subseteq Q$ a distinguished set of initial states,*
- *$\delta : Q \times \Sigma \to 2^Q$ a transition function,*
- *$\lambda : Q \to \Delta$ the output function, and Δ the output alphabet.*

In the following, $\Delta = \mathbb{B}_3$.

A Moore machine is *deterministic*, if and only if for all $q \in Q$, $a \in \Sigma$, $|\delta(q, a)| \leq 1$, and $|Q_0| = 1$. The outputs of a Moore machine, defined by the function λ, are thus determined by the current state $q \in Q$ alone, rather than by the input symbols leading to that state. In what follows, δ is extended to the domain of finite words in terms of $\delta' : 2^Q \times \Sigma^* \to 2^Q$ by $\delta'(Q', \epsilon) = Q'$, with $Q' \subseteq Q$, and

$$\delta'(Q', ua) = \bigcup_{q' \in \delta'(Q', u)} \delta(q', a),$$

where $a \in \Sigma$. To simplify notation, from this point forward δ is used for both δ and δ', depending on the context. For a deterministic Moore machine, by λ the function is denoted which, applied to a word $u \in \Sigma^*$, yields the output in the state reached by u rather than the sequence of outputs to that state.

As in §3, by $\mathcal{A}^\varphi$ the nondeterministic Büchi automaton which accepts all models for $\varphi \in \text{LTL}_3$ is denoted. Further for a nondeterministic Büchi automaton $\mathcal{A}$, $\mathcal{A}(q)$ is the nondeterministic Büchi automaton that coincides with $\mathcal{A}$ except for Q_0, which is defined as $Q_0 = \{q\}$.

Checking as to whether the language of a Büchi automaton $\mathcal{A}$ from a state $q \in Q$ is non-empty, is equivalent to the problem of finding a *strongly connected component* in the transition graph of $\mathcal{A}$ which contains at least one state $q' \in F$ (see Definition 3.1.10 for Büchi-acceptance). The notion of a strongly connected component is defined via the following reachability relation, $\leftrightsquigarrow$.

Definition 4.4.2: *Let $\mathcal{A}$ be a Büchi automaton defined in the usual way, and $q, q' \in Q$. $q \rightsquigarrow q'$ expresses that for a run in $\mathcal{A}$, q' is reachable from q. Let $\leftrightsquigarrow \subseteq Q \times Q$ be the* mutual reachability relation *over Q. Then $q \leftrightsquigarrow q'$ holds, if and only if both $q \rightsquigarrow q'$ and $q' \rightsquigarrow q$ hold. The set of strongly connected components in $\mathcal{A}$ is made up of the partition over Q induced by the relation $\leftrightsquigarrow$.*

For a state $q \in Q$, $scc(q)$ is used to denote the strongly connected component hosting state q. A component $scc(q)$ is said to be *accepting*, if and only if there exists a state $q' \in scc(q)$, such that $q' \in F$; this is trivially the case for a singleton component $scc(q \in F)$.

Using Tarjan's algorithm, determining all strongly connected components for $\mathcal{A}$ can be achieved in time $O(|Q| \times |E|)$, where E, in this case, is a reference to the number of edges in the state graph underlying $\mathcal{A}$ defined via the transition function δ (cf. Knuth [1998]). For the *size of an automaton*, $(|Q| + |E|)$ is used.

From this observation the following corollary can be directly derived.

Corollary 4.4.1: *Deciding emptiness for a Büchi automaton $\mathcal{A}$, defined in the usual way, is decidable in linear time with respect to the size of $\mathcal{A}$.*

Note that the above result is not optimal (cf. Bloem et al. [2000]), but for the purpose of this thesis, it shall suffice to stress merely decidability of the problem. Further, checking emptiness of automata is performed in the runtime reflection framework offline; that is, during the construction of the monitors. As such, a suboptimal approach has no negative impact on performance of the verification process. However, for a discussion of the complexity of the overall solution, see also §4.4.3.

Putting all the pieces together, a first important observation leading towards a dynamic decision procedure for LTL_3 can be summarised.

Lemma 4.4.1: *Let $\mathcal{A}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$ denote a nondeterministic Büchi automaton, such that $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$. For $u \in \Sigma^*$, let $\delta(Q_0^\varphi, u) = \{q_1, \ldots, q_l\}$. Then*

$$[u \models \varphi] \neq \bot \text{ if and only if } \exists q \in \{q_1, \ldots, q_l\} \text{ such that } \mathcal{L}(\mathcal{A}^\varphi(q)) \neq \emptyset.$$

Proof:

The proof is straightforward and follows from the semantics of LTL_3, and the definition of acceptance for Büchi automata (Definition 3.1.10).

($\Leftarrow$) If $\mathcal{L}(\mathcal{A}^\varphi(q)) \neq \emptyset$, then there exists a run in $\mathcal{A}^\varphi(q)$, $\rho \in Q^\omega$ with $\rho(0) = q$, over a word $w \in \Sigma^\omega$ such that $\mathrm{Inf}(\rho) \cap F \neq \emptyset$. Setting $w = u\sigma$, where $u \in \Sigma^*$ is an arbitrary length prefix of w and $\sigma \in \Sigma^\omega$ its continuation, one gets either $[u \models \varphi] =?$ or $[u \models \varphi] = \top$ by the semantics of LTL_3. Conversely, if $\mathcal{L}(\mathcal{A}^\varphi(q)) = \emptyset$, then, by Definition 3.1.10, there exists no run in $\mathcal{A}^\varphi(q)$, $\rho \in Q^\omega$ with $\rho(0) = q$, over a word $w \in \Sigma^\omega$ such that $w \models \varphi$. By the semantics of LTL_3, if for all $w \in \Sigma^\omega$, it holds that $w \not\models \varphi$, then $[u \models \varphi] = \bot$, where $u \in \Sigma^*$, again, denotes an arbitrary length prefix of w.

($\Rightarrow$) Follows directly from Definition 4.3.1 and acceptance of Büchi automata (see Definition 3.1.10). $\square$

Since the emptiness check per state merely indicates as to whether there exists a run in the automaton from a state q to an accepting component $scc(q' \in F)$, it is necessary to additionally determine whether there exists also a run from q to a component $scc(q'' \notin F)$ with $q \neq q''$, where none of the states are accepting. If that is the case, the conclusion follows that both an accepting run as well as a violating run from q are possible. The foundation for this test is given in the following lemma, which is the dual to the previous one.

Lemma 4.4.2: *Let $\mathcal{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, F^{\neg\varphi})$ denote a nondeterministic Büchi automaton, such that $\mathcal{L}(\mathcal{A}^{\neg\varphi}) = \mathcal{L}(\neg\varphi)$. For $u \in \Sigma^*$, let $\delta(Q_0^{\neg\varphi}, u) = \{q_1, \ldots, q_l\}$. Then*

$$[u \models \varphi] \neq \top \textit{ if and only if } \exists q \in \{q_1, \ldots, q_l\} \textit{ such that } \mathcal{L}(\mathcal{A}^{\neg\varphi}(q)) \neq \emptyset.$$

Proof:
Correctness follows directly from Lemma 4.4.1 by substitution of $\neg\varphi$ for φ. $\square$

For $\mathcal{A}^\varphi$ and $\mathcal{A}^{\neg\varphi}$, a function $\mathcal{F}^\varphi : Q^\varphi \to \mathbb{B}$, respectively for $\mathcal{F}^{\neg\varphi}$, $\mathcal{F}^{\neg\varphi} : Q^{\neg\varphi} \to \mathbb{B}$, can be defined, assigning to each state q whether the language of the respective automaton starting in state q is not empty.

Therefore, using $\mathcal{F}^\varphi$ and $\mathcal{F}^{\neg\varphi}$, two nondeterministic finite automata can be defined, $\hat{A}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, \hat{F}^\varphi)$ and $\hat{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \hat{F}^{\neg\varphi})$ where

$$\hat{F}^\varphi = \{q \in Q^\varphi \mid \mathcal{F}^\varphi(q) = \top\} \quad \text{and} \quad \hat{F}^{\neg\varphi} = \{q \in Q^{\neg\varphi} \mid \mathcal{F}^{\neg\varphi}(q) = \top\}.$$

Obviously, $\hat{A}^\varphi$, respectively $\hat{A}^{\neg\varphi}$ accept the finite traces $u \in \Sigma^*$ for which $[u \models \varphi]$ evaluates to $\neq \bot$ and, respectively, $\neq \top$. This is summed up briefly in the following lemma.

Lemma 4.4.3: *Using the notation as before, for all $u \in \Sigma^*$ the following holds.*

$$u \in \mathcal{L}(\hat{A}^\varphi) \textit{ if and only if } [u \models \varphi] \neq \bot \textit{ and } u \in \mathcal{L}(\hat{A}^{\neg\varphi}) \textit{ if and only if } [u \models \varphi] \neq \top.$$

Therefore, $[u \models \varphi]$ can be evaluated according to Lemma 4.4.3 as follows.

Lemma 4.4.4: *Let $u \in \Sigma^*$ be a finite trace of actions and let $\varphi \in LTL_3$ be a property. If $\hat{A}^{\varphi} = (\Sigma, Q^{\varphi}, Q_0^{\varphi}, \delta^{\varphi}, \hat{F}^{\varphi})$ and $\hat{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \hat{F}^{\neg\varphi})$ are the nondeterministic finite automata as defined for Lemma 4.4.3, then the following holds.*

$$[u \models \varphi] = \begin{cases} \top & \text{if } u \notin \mathcal{L}(\hat{A}^{\neg\varphi}) \\ \bot & \text{if } u \notin \mathcal{L}(\hat{A}^{\varphi}) \\ ? & \text{if } u \in \mathcal{L}(\hat{A}^{\varphi}) \text{ and } u \in \mathcal{L}(\hat{A}^{\neg\varphi}). \end{cases}$$

The lemma yields a simple procedure to evaluate the semantics of $\varphi \in \mathrm{LTL}_3$ for a given finite trace u: both $u \in \mathcal{L}(\hat{A}^{\neg\varphi})$ and $u \in \mathcal{L}(\hat{A}^{\varphi})$ are evaluated, and Lemma 4.4.4 is used to determine $[u \models \varphi]$.

4.4.1 Monitor construction

The actual monitor component used for runtime verification in the framework can be constructed by defining a deterministic finite state machine $\bar{A}^{\varphi}$, which is a Moore-type, that outputs for each finite string u its associated 3-valued semantical evaluation with respect to some LTL_3 formula φ. Using the steps outlined in the previous section, this is possible due to the following well-known (cf. Hopcroft and Ullman [1979]) correspondence between deterministic and nondeterministic finite automata.

Theorem 4.4.1 (Folklore): *If $\hat{A} = (\Sigma, Q, q_0, \delta, F)$ is a nondeterministic finite automaton, then there exists a deterministic finite automaton $A' = (\Sigma, Q', \{q_0\}, \delta', F')$, such that $\mathcal{L}(\hat{A}) = \mathcal{L}(A')$.*

Proof:
For brevity, merely the proof idea for this theorem is sketched as follows. Without loss of generality, complete automata can be assumed. Further, the extended transition function over words is used as introduced in §4.4.

The proof then proceeds by induction over a finite word $u \in \Sigma$ and by showing that $\delta'(\{q_0\}, u) = \delta(q_0, u)$. Notice in this context that each of the transition functions return a set of states from Q, but δ' interprets this set as one of the states of Q', whereas δ interprets this set as a subset of Q. Since Q' is based on the power-set of Q, conversion can be *exponential*. For a complete proof of this theorem, see, e. g., the standard works of Hopcroft and Ullman [1979]. □

Without loss of generality, let $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$ be deterministic variants of $\hat{A}^{\varphi}$, respectively $\hat{A}^{\neg\varphi}$. Recall further that the product of two finite automata A and A', denoted $A \times A'$, is defined as a tuple $(\Sigma, Q^A \times Q^{A'}, \{(q_0^A, q_0^{A'})\}, \delta, F^A \times F^{A'})$, where for two states $p \in Q^A, q \in Q^{A'}$, and a symbol $a \in \Sigma$, $\delta((p,q),a) = (\delta^A(p,a), \delta^{A'}(q,a))$.

The Moore machine in question can then be defined as the product of $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$ as follows.

Definition 4.4.3: *Let $\tilde{A}^\varphi = (\Sigma, Q^\varphi, \{q_0^\varphi\}, \delta^\varphi, \tilde{F}^\varphi)$ and $\tilde{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, \{q_0^{\neg\varphi}\}, \delta^{\neg\varphi}, \tilde{F}^{\neg\varphi})$ be the deterministic finite automata which correspond to the two nondeterministic finite automata $\hat{A}^\varphi$ and $\hat{A}^{\neg\varphi}$ as defined for Lemma 4.4.3. Then the* monitor *$\bar{A}^\varphi = \tilde{A}^\varphi \times \tilde{A}^{\neg\varphi}$ is defined as a finite Moore state machine $(\Sigma, \bar{Q}, \bar{q}_0, \bar{\delta}, \bar{\lambda})$, where*

- *Σ is the finite input alphabet,*
- *$\bar{Q} = Q^\varphi \times Q^{\neg\varphi}$ a finite set of states,*
- *$\bar{q}_0 = (q_0^\varphi, q_0^{\neg\varphi})$ an initial state,*
- *$\bar{\delta}((q, q'), a) = (\delta^\varphi(q, a), \delta^{\neg\varphi}(q', a))$ a transition function, and*
- *$\bar{\lambda} : \bar{Q} \to \mathbb{B}_3$ the output function defined as*

$$\bar{\lambda}((q, q')) = \begin{cases} \top & \textit{if } q' \notin \tilde{F}^{\neg\varphi} \\ \bot & \textit{if } q \notin \tilde{F}^\varphi \\ ? & \textit{if } q \in \tilde{F}^\varphi \textit{ and } q' \in \tilde{F}^{\neg\varphi}. \end{cases}$$

The above can now be formulated in terms of a theorem as follows.

Theorem 4.4.2: *Let $\varphi \in LTL_3$ and let $\bar{A}^\varphi = (\Sigma, \bar{Q}, \bar{q}_0, \bar{\delta}, \bar{\lambda})$ be the corresponding monitor. Then, for all $u \in \Sigma^*$ it holds that $[u \models \varphi] = \bar{\lambda}(\bar{\delta}(\bar{q}_0, u))$.*

Proof:
Follows directly from Definition 4.4.3. □

4.4.2 Example: The C++ static initialisation order fiasco

Now a simple but comprehensive real-world example is examined in more detail, which also highlights most of the features described above.

In a program written in C++, all static objects of an executable are initialised before the `main` method is entered, however, their order is undefined, and initialisation thus nondeterministic (cf. Stroustrup [2000]). In consequence, if threads get spawned before executing `main`, it is difficult to ensure that all resources necessary to synchronise those threads are already initialised, such as globally available and statically initialised mutex objects. This problem is generally known as the *static initialisation order fiasco* (cf. Dewhurst [2002]). The "fiasco" is an especially complicated one when large applications are built from a number of different frameworks which must remain independent from each other.

Using the entire runtime reflection framework as outlined in §1.1 with a C++ logging layer such as the Apache Software Foundation's library, log4cxx[2], for gaining access to signals emitted by the application's threads, it is possible to construct a monitor over an alphabet $\Sigma = 2^{AP}$, where $\{spawn, init\} \subseteq AP$, for a property $\varphi \equiv \neg spawn \mathbf{U} init$. In other words, the monitor ensures that no thread gets spawned before the application under scrutiny has properly finished initialisation.

[2]See `http://www.apache.org/`.

This example further illustrates the need for having three truth values, instead of two when monitoring a running system: Intuitively, a monitor for φ should raise an alarm only, if a thread was spawned before *init* occurred; should eventually switch itself off, if *init* occurred before *spawn*; and until either happens should return ?, indicating the necessity for further observation.

Using the translation algorithm from formulae of LTL to Büchi automata as proposed by Fritz [2003], one obtains for φ, respectively $\neg\varphi$, the Büchi automata depicted in Fig. 4.1.

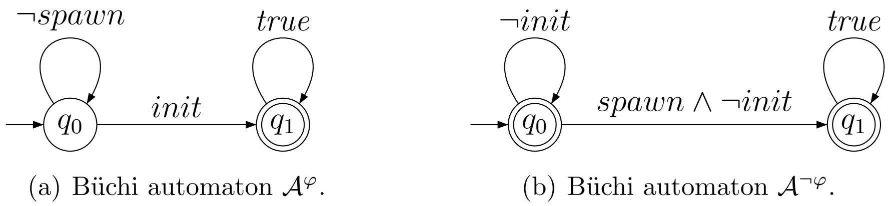


Fig. 4.1: The Büchi automaton $\mathcal{A}^{\varphi}$ accepts all words not in $\mathcal{L}(\mathcal{A}^{\neg\varphi})$, and vice versa.

Using the tools described in greater detail in §6, then two nondeterministic finite automata $\hat{A}^{\varphi} = (\Sigma, Q^{\varphi}, Q_0^{\varphi}, \delta^{\varphi}, \hat{F}^{\varphi})$ and $\hat{A}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \hat{F}^{\neg\varphi})$ are constructed, where the accepting states $\hat{F}^{\varphi}$ and $\hat{F}^{\neg\varphi}$ are defined in accordance with Lemma 4.4.3. Without loss of generality, $\hat{A}^{\varphi}$ gets extended (the same needs to be done for $\hat{A}^{\neg\varphi}$) with an additional state q indicating a violation, thus, being a "sink"; that is, a strongly connected component where $scc(q) \cap \hat{F} = \emptyset$. Note that this step is not mandatory, as the semantic expressiveness of a complete and a non-complete finite automaton is equal (see Theorem 3.1.1). Which type of automaton is preferred depends solely on the proper interpretation of the presence or absence of such redundant transitions, when executing the automaton (cf. Hopcroft and Ullman [1979]). The result of this step is depicted in Fig. 4.2. In it, the respective states q_2 represent the rejecting states, since there exists no path back to an accepting state.

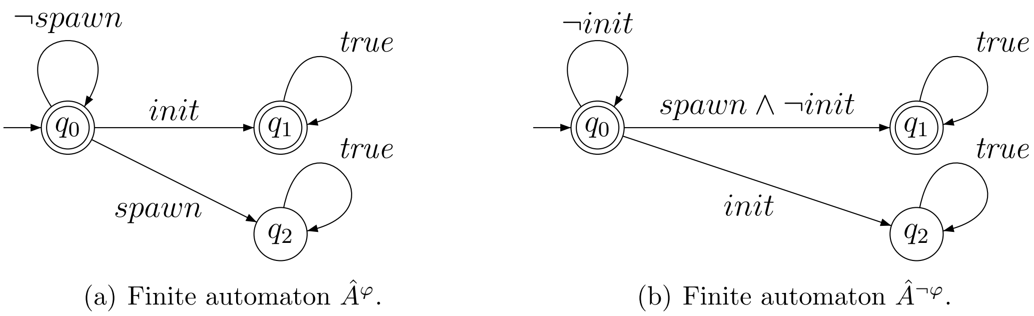


Fig. 4.2: The corresponding finite automata derived from $\mathcal{A}^{\varphi}$ and $\mathcal{A}^{\neg\varphi}$.

Note that in this particular case, the finite automata are already deterministic. Further, the truth value of a propositional symbol, which is not part of the transition label, is irrelevant with respect to whether the according transition is to be triggered. Hence, it holds in this example that $\hat{A}^{\varphi} = \tilde{A}^{\varphi}$ and $\hat{A}^{\neg\varphi} = \tilde{A}^{\neg\varphi}$.

After these preliminaries and according to Definition 4.4.3, the Moore machine which will then serve as monitor for the static initialisation order fiasco can now be constructed. For this purpose, basically, the product of $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$ yielding the automaton depicted in Fig 4.3 is built. This automaton now corresponds with the user's intuition, and yields ? while neither event occurred, and either $\top$ or $\bot$, otherwise. Notice, the respective output symbols of the Moore machine are denoted below the state labels.

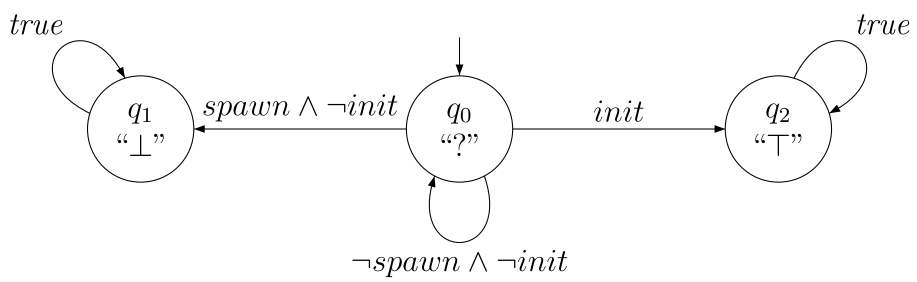


Fig. 4.3: The corresponding deterministic finite Moore state machine.

Execution of the Moore machine happens by evaluating a bit-vector encoding the presence or absence of an event, supplied by the logger. Consider the following example observation violating φ as in $u = \{\neg spawn, \neg init\}, \{spawn, \neg init\}, \{spawn, init\}, \ldots$ whose corresponding run, ρ, through the Moore machine would be as follows: $\rho = q_0, q_1, q_1, \ldots$. Once q_1 is reached for the first time, the monitor outputs $\bot$, and the user knows that the property is violated, regardless as to what the future will bring. Theoretically, at this stage, the user could turn the monitor off and restart or modify the program under scrutiny.

On the contrary, for an observation $u = \{\neg spawn, \neg init\}, \{\neg spawn, init\}, \{spawn, init\}, \ldots$, the run $\rho = q_0, q_2, q_2, \ldots$ is observed, which means that u satisfies φ, regardless as to what the future will bring. Again, the monitor can be switched off as soon as q_2 is entered for the first time and $\top$ is returned.

Note that the transition from q_0 to q_1 would be omitted in the actual monitor, and the automaton then interpreted accordingly.

4.4.3 Complexity

The entire procedure for constructing a monitor, i. e., a finite state machine (FSM), from a SALT specification (see §3.3) is summarised in Fig. 4.4, where NBA is an abbreviation for nondeterministic Büchi automaton, NFA for nondeterministic finite automaton, and DFA for deterministic finite automaton.

Input (1) Formula (2) NBA (3) Emptiness per state (4) NFA (5) DFA (6) FSM

$$\begin{tikzcd}
& \varphi \arrow[r] & A^{\varphi} \arrow[r] & \mathcal{F}^{\varphi} \arrow[r] & \hat{A}^{\varphi} \arrow[r] & \tilde{A}^{\varphi} \arrow[dr] & \\
\textsc{Salt} \arrow[ur] \arrow[dr] & & & & & & \bar{A} \\
& \neg\varphi \arrow[r] & A^{\neg\varphi} \arrow[r] & \mathcal{F}^{\neg\varphi} \arrow[r] & \hat{A}^{\neg\varphi} \arrow[r] & \tilde{A}^{\neg\varphi} \arrow[ur] &
\end{tikzcd}$$

Fig. 4.4: The procedure for getting $[u \models \varphi]$ for a given specification.

Now the size of the resulting FSM is studied in more detail. Since the translation of a Salt specification to an LTL formula φ usually does not add redundancy to the resulting formula (see §6), it is safe to assume this step as constant with respect to the size of the formula, without loss of generality. Given φ, step 1 requires replication of φ and negation, i. e., it is linear in the original size. In step 2, the construction of the nondeterministic Büchi automata is worst-case exponential in the size of the formula, denoted $|\varphi|$ (Fritz [2003]). Steps 3 and 4, leading to $\tilde{A}^{\varphi}$ and $\tilde{A}^{\neg\varphi}$, do not change the size of the original automata. Then, computing the deterministic automata of step 5 might again involve an exponential "blowup" with respect to the size of the corresponding NFAs (see Theorem 4.4.1). Hence, in total, the Moore machine of step 6 may, in the worst case, experience a double exponential "blowup".

At first sight this result may appear to be unemployable for the purpose of runtime verification. However, it is possible to show that this leads, in fact, to an *optimal* solution for the underlying finite state machine, when reconsidering the following theoretical results.

The foundation for this lies in the fact that monitors are but deterministic finite state machines accepting words of the language $L = \mathcal{L}(\hat{A}^{\varphi} \times \hat{A}^{\neg\varphi})$. For the languages accepted, the following definition of *equivalence* applies:

Definition 4.4.4: *Let $L \in \Sigma^*$ be a regular language, and $\sim_L \subseteq \Sigma^* \times \Sigma^*$. Two words $u, u' \in \Sigma^*$ are called L-*equivalent*, denoted $u \sim_L u'$, if and only if $uv \in L \Leftrightarrow u'v \in L$, where $v \in \Sigma^*$. The equivalence class of u is defined as $[u]_{\sim_L} = \{u' \in \Sigma^* \mid u \sim_L u'\}$.*

The Myhill-Nerode theorem (cf. Hopcroft and Ullman [1979]) provides a necessary and sufficient condition for a language to be regular, and is based upon the *equivalence classes of regular languages* and the correspondence to automata. From the theorem, various minimisation algorithms for finite automata have been derived that combine states according to their L-equivalence, and that naturally apply also for the generated monitors. Minimisation then implies that any other method, in the worst case, must have the same complexity as the previously constructed monitor in the runtime reflection framework.

Hence, better complexity results in other approaches are either due to using a restricted fragment of LTL or otherwise imply that the chosen temporal operators might not limit the expressive power of LTL but sometimes impose long formulas for encoding the desired behaviour.

4.4.4 Discussion: Informativeness vs. minimality

Various approaches to runtime verification and reasoning about systems based on truncated paths have been based upon a seminal paper by Kupferman and Vardi [2001]. The authors of that paper have introduced the concept of *informativeness* and that of *pathological safety formulae*. Intuitively, a finite bad prefix for a formula $\varphi \in \mathrm{LTL}$ is called informative, if and only if it is a bad prefix for all of φ's sub-formulae given by its *closure*. That is, all sub-formulae in the original formula have a reason for not holding.

Definition 4.4.5: *The* closure *of a formula $\varphi \in LTL$ is given by the set $cl(\varphi)$ and defined as follows:*

- $\varphi \in cl(\varphi)$,
- $\neg\varphi \in cl(\varphi)$,
- $\varphi_1 \ op\ \varphi_2 \in cl(\varphi) \Rightarrow \varphi_1, \varphi_2 \in cl(\varphi)$,
- $\varphi_1 \mathbf{U} \varphi_2 \in cl(\varphi) \Rightarrow \varphi_1, \varphi_2 \in cl(\varphi)$,
- $\mathbf{X}\varphi_1 \in cl(\varphi) \Rightarrow \varphi_1 \in cl(\varphi)$,

where op defines the usual Boolean operations.

A safety property is called pathologically safe if there is a prefix that violates φ which has no informative bad prefix. For instance, the formula $(\mathbf{G}(q \vee \mathbf{GF}p) \wedge \mathbf{G}(r \vee \mathbf{GF}\neg p)) \vee \mathbf{G}q \vee \mathbf{G}r$ is pathologically safe (Kupferman and Vardi [2001]). Formally, this is captured in the following definition over the term syntax of φ.

Definition 4.4.6: *Let $\varphi \in LTL$ be in negative normal form, and $u = a_0 a_1 \ldots a_n \in \Sigma^*$. u is* informative, *if and only if there exists a mapping $L : \{0, \ldots, n+1\} \rightarrow 2^{cl(\neg\varphi)}$, such that the following conditions hold:*

- $\neg\varphi \in L(0)$,
- $L(n+1)$ *is empty, and*
- *for all $0 \leq i \leq n$ and $\psi \in L(i)$, the following hold.*
  - *If ψ is an atomic proposition, then $\psi \in a_i$.*
  - *If $\psi = \psi_1 \ op\ \psi_2$, then $\psi_1 \in L(i)\ op\ \psi_2 \in L(i)$.*
  - *If $\psi = \psi_1 \mathbf{U} \psi_2$, then (i) $\psi_2 \in L(i)$ or (ii) ($\psi_1 \in L(i)$ and $\psi_1 \mathbf{U} \psi_2 \in L(i+1)$).*
  - *If $\psi = \mathbf{X}\psi_1$, then $\psi_1 \in L(i+1)$,*

  where, as in the previous definition, op denotes the set of standard Boolean connectives.

In the same paper, a worst-case double exponential construction with respect to the size of the underlying formula for a finite state machine is given which, given a (non-) pathological safety property $\varphi \in \mathrm{LTL}$ in negative normal form, accepts all of its bad prefixes. Latvala [2002] extended this work by introducing a single exponential construction for a finite state machine which accepts, given a non-pathological safety property $\varphi \in$ LTL in negative normal form, all its informative bad prefixes. Moreover, Geilen [2001] also introduced an on-the-fly monitoring algorithm for this particular set of LTL properties.

Notably, these procedures differ in two ways from the previously presented approach to monitoring properties using a 3-valued semantics for LTL. Firstly, they are based on the syntactic structure of the underlying property as is the concept of informativeness. Secondly, they are defined only for safety properties, and sometimes work only for a subclass of safety properties, such as monitoring non-pathological safety properties.

On the other hand, monitoring in the runtime reflection framework, although optimally used for safety properties (see Definition 3.2.1), works equally with properties other than safety, even liveness. However, depending on the liveness property (see Definition 3.2.3), this could involve having infinitely many ? in the monitor's output, as a liveness property can only be refuted by an infinite counterexample. Moreover, monitoring in the runtime reflection framework is not restricted to minimal informative bad prefixes, but it detects *all* minimal bad prefixes, if these exist and the system under scrutiny shows the according behaviour. Essentially, this is achieved by the different approach to monitoring, which relies on automata analysis, rather than the term structure of the properties. Notice, predictiveness, as is achieved by detection of minimal bad prefixes, is also not given in full generality in the other works cited above. However, for completeness it should be pointed out that d'Amorim and Rosu [2005] have now addressed in particular the predictiveness problem in their approaches by adding to each state in the monitor an OBDD encoding possible future runs of the monitor. Although feasible due to the efficiency of OBDD look-up procedures, it involves handling additional data structures, analyses, and tasks to perform at runtime.

4.5 Reflecting real-time

In order to be able to reason about real-time properties of distributed and reactive systems, such as strict execution deadlines and the timeliness of communication patterns or events, LTL as introduced in the previous section, is generally not expressive enough. That is, LTL is well suited for expressing qualitative constraints about the ordering of actions along a trace, but not for expressing quantitative constraints. Although often not explicitly stated, with standard LTL there is an underlying assumption associated that the systems under scrutiny behave somewhat *synchronous*, i. e., that there is a fixed (discrete) notion of a step where actions can be observed. In the context of this thesis, this class of systems is referred to as *quasi-synchronous* to not confuse them with the established class of *time-synchronous* systems adhering to the more strict *perfect synchrony hypothesis* (cf. Halbwachs et al. [1991], Berry [2000], Benveniste et al. [2003]).

Reactive real-time systems as they are used increasingly, e. g., in the embedded domain, adhere more often to a strictly *event-triggered* nature (cf. Romberg and Bauer [2004]). In other words, actions are not only performed in uniform cycles or steps, but occur spontaneously and at arbitrary intervals, although depending naturally on the resolution of timer hardware or bus arbitration, for instance. In other words, when reasoning about real-time properties, not only the order of events is important, but also the exact time when they occurred. Consequently, for LTL, real-time constraints have been taken into account, but often at the expense of desirable LTL properties, such as decidability, for instance; this, however, depends strongly on *how* real-time is actually added to the formalism. A good discussion of this particular topic is available, e. g., from Alur and Henzinger [1991] and Maler et al. [2005].

4.5.1 The real-time logic TLTL

For the formulation of real-time properties that go beyond the expressiveness of standard LTL, this section presents the logic TLTL (timed linear-time temporal logic, sometimes also referred to as state-clock logic) as introduced originally by Raskin and Schobbens [1997], but in the form used by D'Souza [2003]. The language expressible by a TLTL formula can be defined by *event-clock automata*, a decidable and determinisable subclass of *timed automata* (see §4.5.2). More specifically, D'Souza [2003] showed that TLTL corresponds exactly to the class of languages, definable in the first-order fragment of monadic second order logic, interpreted over timed words. Thus, it can be considered to be the natural counterpart of LTL in the timed setting (see also §3.3.2).

Definition 4.5.1: *A (possibly) infinite* timed word w *over an alphabet* Σ *is a (possibly) infinite sequence of* timed actions $(a_0, t_0)(a_1, t_1)\dots$ *consisting of symbols* $a_i \in \Sigma$*, and non-negative numbers* $t_i \in \mathbb{R}^{\geq 0}$*, such that*

1. *for each* $i \in \mathbb{N}$*,* $t_i < t_{i+1}$*, and* (strict monotonicity)
2. *for all* $t \in \mathbb{R}^{\geq 0}$ *there is an* $i \in \mathbb{N}$ *such that* $t_i > t$*.* (progress)

For reasons outlined above, timed actions are also referred to as *events*.

To simplify notation, $(\Sigma \times \mathbb{R}^{\geq 0})$ is also abbreviated by $T\Sigma$. Thus, a finite timed string or word is an element of $T\Sigma^*$ and the domain of infinite timed words is denoted by $T\Sigma^\omega$. Furthermore, for w as given above, its string of actions (i. e., the projection to the first component) is called the *untimed word* of w, denoted by $ut(w)$.

TLTL is a so-called *dense* real-time logic, in a sense that every symbol $a \in \Sigma$ is associated with an *event-recording clock*, x_a, and an *event-predicting clock*, y_a, from $\mathbb{R} \cup \{\perp\}$. Before getting back to this in more detail in Definition 4.5.3, first the notion of an *event-clock* is formally introduced.

Definition 4.5.2: *Given an alphabet,* Σ*, the set of* event clocks *associated with the elements in* Σ *is the set* $\mathbb{C}_\Sigma = \mathbb{H}_\Sigma \cup \mathbb{P}_\Sigma$*, where* $\mathbb{H}_\Sigma = \{x_a \mid a \in \Sigma\}$ *is the set of* event-recording clocks, *i. e., an event-recording clock is associated with each element* $a \in \Sigma$, *and where* $\mathbb{P}_\Sigma = \{y_a \mid a \in \Sigma\}$ *is the set of* event-predicting clocks, *i. e., an event-predicting clock is associated with each* $a \in \Sigma$.

In what follows, $x \in \mathbb{H}_\Sigma$ denotes any event-recording clock of $\mathbb{C}_\Sigma$, and $y \in \mathbb{P}_\Sigma$ any event-predicting clock of $\mathbb{C}_\Sigma$.

Definition 4.5.3: *A* clock valuation function *is a function over a (possibly) infinite timed word* $w \in T\Sigma^\omega$ *with* $w_i = (a_i, t_i)$ *for each* $i \in \mathbb{N}$, $\gamma_i : \mathbb{C}_\Sigma \to \mathbb{R}^{\geq 0} \cup \{\perp\}$, *which assigns a positive real, or undefined value,* $\perp$, *to each clock variable corresponding to position* i *as follows:*

$$\begin{aligned}\gamma_i(x_a) &= \begin{cases} t_i - t_j & \textit{if } \exists j < i : a_j = a \textit{ and } \forall k : j < k < i \Rightarrow a_k \neq a \\ \perp & \textit{otherwise.} \end{cases} \\ \gamma_i(y_a) &= \begin{cases} t_j - t_i & \textit{if } \exists j > i : a_j = a \textit{ and } \forall k : i < k < j \Rightarrow a_k \neq a \\ \perp & \textit{otherwise.} \end{cases}\end{aligned}$$

Intuitively, the above definition asserts that, given an (infinite) timed word w, the value of the event-recording clock variable x_a at position i of w equals $t_i - t_j$, where j represents the *last* position preceding i such that $a_j = a$. If no such position exists, then the value of x_a remains undefined, denoted by $\perp$. The event-predicting clock variable y_a equals $t_j - t_i$, where j represents the next position *after* i such that $a_i = a$. If no such position exists, again, the variable remains undefined, denoted by $\perp$.

Example. For instance, consider a timed word $w = (a,1)(a,3)(a,5)(b,8)(c,9)\ldots$, and two clock variables, x_a and y_a, denoting the last occurrence of a and the next occurrence of a with respect to an index $i \in \mathbb{N}$, respectively. Fig. 4.5 shows the values of γ_i for x_a and y_a along the given prefix of w.

Syntax

The logic TLTL introduces dedicated real-time operators to LTL, $\lhd_a$ and $\rhd_a$, which span the time interval upon which an action a occurred last, or upon which an action a will occur in the future. The set of *intervals* used for this purpose, $\mathcal{I}$, contains all intervals of the form (l, r), $[l, r)$, $(l, r]$, or $[l, r]$, where $l, r \in \mathbb{R}^{\geq 0} \cup \{\infty\}$.

Without loss of generality, assume $l < r$, except for $[l, r]$, and for intervals $(l, r]$, or $[l, r]$ that $r \neq \infty$. To simplify notation, [(and)] is used for interval borders which can either be (or [, respectively),]. For instance, the interval $[1, \infty)$ denotes a set $\{t \in \mathbb{R}^{\geq 0} \mid 1 \leq t\}$.

Definition 4.5.4: *Let* Σ *denote a finite set of actions. The set of well-formed TLTL formulae over alphabet* Σ *is denoted by* $TLTL(\Sigma)$, *and given by the following abstract syntax:*

$$\varphi ::= \textit{true} \mid a \mid \lhd_a \in I \mid \rhd_a \in I \mid \neg\varphi \mid \varphi \; op \; \varphi \mid \varphi \, \mathbf{U} \varphi \mid \mathbf{X}\varphi \quad (a \in \Sigma),$$

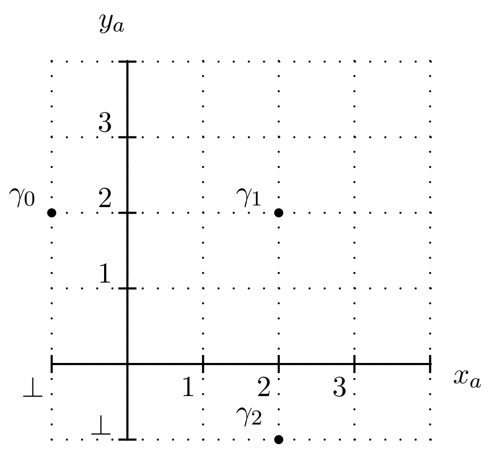


Fig. 4.5: Evaluations of x_a and y_a for $w = (a,1)(a,3)(a,5)(b,8)(c,9)\ldots$.

with $\varphi \in TLTL(\Sigma)$, and where op represents a binary Boolean operator defined by the set $op \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$, and where $\lhd_a$ is the operator which measures the time elapsed since the last occurrence of a, and $\rhd_a$ the operator which predicts the next occurrence of a within a timed interval $I \in \mathcal{I}$.

As in the untimed case, if the parameterisation of the set of formulae is clear from the context, one can abstain from naming the concrete alphabet in its name, and simply write TLTL, rather than TLTL(Σ), when making reference to the set of well-formed TLTL formulae.

Semantics

Definition 4.5.5: *Let $\varphi \in TLTL(\Sigma)$, and $i \in \mathbb{N}^{\geq 0}$ denote a position. The* semantics of TLTL(Σ) *is defined inductively over infinite timed words $w \in T\Sigma^\omega$, where $w = (a_0, t_0)(a_1, t_1)\ldots$, as follows:*

$$\begin{array}{lll}
w, i \models true & & \\
w, i \models \neg\varphi & \Leftrightarrow & w, i \not\models \varphi \\
w, i \models a & \Leftrightarrow & a_i = a \\
w, i \models \lhd_a \in I & \Leftrightarrow & \gamma_i(x_a) \in I \\
w, i \models \rhd_a \in I & \Leftrightarrow & \gamma_i(y_a) \in I \\
w, i \models \varphi_1 \; op \; \varphi_2 & \Leftrightarrow & (w, i \models \varphi_1 \; op \; w, i \models \varphi_2) \\
w, i \models \varphi_1 \mathbf{U} \varphi_2 & \Leftrightarrow & \exists k \geq i : (w, k \models \varphi_2 \wedge \forall l : (0 \leq l < k \Rightarrow w, l \models \varphi_1)) \\
w, i \models \mathbf{X}\varphi & \Leftrightarrow & w, i+1 \models \varphi
\end{array}$$

Further, let $w \models \varphi$, if and only if $w, 0 \models \varphi$.

Like in the untimed case, $w \in T\Sigma^\omega$ is said to satisfy (alternatively, is a model of) the formula $\varphi \in$ TLTL(Σ), if and only if $w \models \varphi$ holds. The set given by $\mathcal{L}(\varphi) = \{w \in$

$T\Sigma^\omega \mid w \models \varphi\}$ of all models of φ is called the (timed) *language* of φ. The formula φ is satisfiable if $\mathcal{L}(\varphi) \neq \emptyset$ and unsatisfiable, otherwise. A formula φ is valid, if and only if $\neg\varphi$ is unsatisfiable.

Basically, TLTL enriches untimed or standard LTL by adding two operators for recording and predicting the occurrence of actions. These operators are referred to as the *timed operators*, to the remaining ones as *untimed operators*. The semantics of the untimed TLTL-operators is, therefore, analogous to LTL. Consequently, the typical syntactic derivations of operators (see Definition 3.1.15) transfers over to TLTL in a natural way. In other words, every valid LTL formula is also a valid TLTL formula using only the set of untimed operators.

Clock constraints

Formulae in TLTL which make use of timed operators, implicitly define constraints over values of clock variables. Basically, a *clock constraint* compares the value of a clock variable to a natural number.

Definition 4.5.6: *Let $\Psi(\mathbb{C}_\Sigma)$ be the set of clock constraints over $\mathbb{C}_\Sigma$. A* clock constraint *$\psi \in \Psi(\mathbb{C}_\Sigma)$ is a conjunction of atomic formulae of the form $z \bowtie c$, where $z \in \mathbb{C}_\Sigma$, $\bowtie \in \{<, \leq, \geq, >\}$, and $c \in \mathbb{N}$.*

Definition 4.5.7: *Let $z \in \mathbb{C}_\Sigma$ be a clock variable, and $\gamma_i(z)$ the valuation function for z over a (possibly) infinite timed word w. The timed word w* satisfies *a clock constraint $\psi = z \bowtie c$ at position i with $i, c \in \mathbb{N}$, according to the following rules:*

- *$w, i \models z \bowtie c$ if and only if $\gamma_i(z) \bowtie c$,*
- *$w, i \models \neg\psi$ if and only if $w, i \not\models \psi$,*
- *$w, i \models \psi_1 \vee \psi_2$ if and only if $w, i \models \psi_1 \vee w, i \models \psi_2$,*

where $\bowtie$ is evaluated as usual in non-negative real numbers, and $\bot \bowtie c$ always evaluates to false.

Given a clock constraint ψ and a clock valuation function γ, one can also write $\gamma \models \psi$ to denote that according to γ, constraint ψ is fulfilled, where $\bot \bowtie c$ for $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, \geq, >\}$ does not hold, and the remaining cases are defined in the expected manner. For example, one could have $\gamma(x_a) = 3.2 \models x_a \leq 5$.

Some example properties

The expressive power of TLTL stems from formulae like $\mathbf{G}(\rhd_a \in [0, 5])$ requiring the re-occurrence of an action a every five time units. For reasons outlined in §3.2, the formula also constitutes a classical safety-property, i. e., the property specified is always finitely refutable. The formal basis upon which to decide this is formally introduced in the next section.

Further examples include:

- $\mathbf{G}(\neg valid \Rightarrow \rhd_{valid} \in [0,5])$: always if a an event decoded by proposition *valid* has not occurred, it has to occur at least within the next five time units.
- $\mathbf{G}((\lhd_q \in [0,3]) \Rightarrow p)$: asserts that always if q occurred within the last three time units, then proposition p has to hold now.
- $\mathbf{G}((p \Rightarrow \rhd_p \in [0,5]) \vee (\mathbf{XF}\neg p))$: always every event p is either followed by another p event distant more than five time units, or never followed by another p event again.

4.5.2 Verification of timed systems

While formulae of LTL are translatable to Büchi automata, formulae of TLTL (like other real-time logics) require a certain type of a *timed automaton* (Alur and Dill [1990]) for translation and verification, which not only accepts actions but entire events, i. e., actions with time-stamps. The class used for verification of TLTL-properties are the so-called *event-clock automata* (Alur et al. [1999]), a derivation from the more general "Alur-Dill automata", which are going to be introduced first.

Timed automata

The concept of timed automata was introduced by Alur and Dill [1990] for modelling and reasoning about the behaviour of real-time systems. Their definition provides a simple, and yet general, way to annotate the states of state-transition graphs with timing constraints using finitely many real-valued clock variables. Thus, timed automata are finite-state automata augmented with clocks and constraints over their possible values. It is assumed that the transitions of a timed automaton are instantaneous. However, time can elapse when the automaton is in a state or a location. When a transition occurs, some of the clocks may be reset to zero. The value of a clock then equals the time that has elapsed since it was last reset. In order to prevent pathological behaviours and to adhere to the definition of timed words as given in Definition 4.5.1, only automata are considered that are *non-Zeno*; that is, only a finite number of transitions can be triggered in a non-empty interval of time.

Definition 4.5.8: *A* timed automaton *(also referred to as Alur-Dill automaton) is defined by a tuple $\mathcal{A} = (\Sigma, Q, Q_0, E, I)$, where*

- *Σ is a finite input alphabet,*
- *Q a finite set of locations,*
- *$Q_0 \subseteq Q$ is a distinguished set of initial locations,*
- *$E \subseteq Q \times \Sigma \times \Psi(\mathbb{C}_\Sigma) \times 2^{\mathbb{C}_\Sigma} \times Q$ a set of transitions, and*
- *$I : Q \rightarrow \Psi(\mathbb{C}_\Sigma)$ is a mapping from locations to clock constraints, called* location invariants.

An edge $e = (q, a, \psi, \lambda, q') \in E$ represents a transition from source location $q \in Q$ upon symbol $a \in \Sigma$ to destination $q' \in Q$, where the clock constraint $\psi \in \Psi(\mathbb{C}_\Sigma)$ specifies when this transition is enabled, and where $\lambda \subseteq \mathbb{C}_\Sigma$ is a finite set of clocks that are reset when the transition is actually taken.

For brevity, the notation $q \xrightarrow{a,\psi,\lambda} q'$ is used, when $(q, a, \psi, \lambda, q') \in E$.

The *semantics of a timed automaton* $\mathcal{A}$ is defined by associating a labelled transition system $S_\mathcal{A}$ with it as follows.

Definition 4.5.9: *Let $S_\mathcal{A}$ denote a labelled transition system for a timed automaton $\mathcal{A} = (\Sigma, Q, Q_0, E, I)$ with a set of states $Q_{S_\mathcal{A}}$ from $Q \times \mathbb{R}^{\geq 0} \cup \{\perp\}$. A state from $S_\mathcal{A}$ is represented by a tuple (q, γ), where $q \in Q$ and $\gamma : \mathbb{C}_\Sigma \rightarrow \mathbb{R}^{\geq 0} \cup \{\perp\}$ is a clock valuation. A state (q, γ) is an initial state if q is an initial location of $\mathcal{A}$ and $\gamma : \mathbb{C}_\Sigma \rightarrow \{0\}$. The set of clocks, λ, reset (and thus, evaluating) to zero is denoted as $\gamma[\lambda := 0]$. The transition system $S_\mathcal{A}$ then possesses two different types of transitions:*

Delay transitions *correspond to the elapsing of time while staying at some location. This is denoted by $(q, \gamma) \xrightarrow{d} (q, \gamma + d)$, where $d \in \mathbb{R}^{\geq 0}$, provided that for every $0 \leq e \leq d$, the invariant $I(q)$ holds for $\gamma + e$.*

Action transitions *correspond to the execution of a transition taken from (q, γ) upon switch $(q, a, \psi, \lambda, q')$, such that γ satisfies ψ, $(q, \gamma) \xrightarrow{a} (q', \gamma')$, where $a \in \Sigma$ and $\gamma' = \gamma[\lambda := 0]$.*

Definition 4.5.10: *An (infinite)* run *θ* of a timed automaton *$\mathcal{A}$ is defined over a transition system, $S_\mathcal{A}$, with an initial state (q_0, γ_0), where $\gamma_0 : \mathbb{C}_\Sigma \rightarrow \{0\}$ over a timed word $w = (a_1, t_1)(a_2, t_2)(a_3, t_3) \ldots \in T\Sigma^\omega$ is a sequence of transitions*

$$\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \ldots$$

such that $t_i = t_{i-1} + d_i$ for all $i \in \mathbb{N}^{\geq 1}$ with $t_0 = 0$.

Definition 4.5.11: *The* timed language *L expressed by $\mathcal{L}(\mathcal{A})$ is the set of all timed words $w \in T\Sigma^\omega$ for which there exists a run of $\mathcal{A}$ over w.*

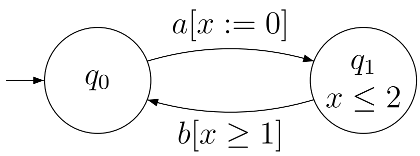


Fig. 4.6: Graphical representation of a timed automaton.

Example. Fig. 4.6 illustrates a timed automaton with two states, q_0 and q_1, a clock x, and alphabet of actions $\Sigma = 2^{AP}$, with $\{a, b\} \subseteq AP$. The initial state, q_0, is not associated with an invariant, which means the system can spend an arbitrary amount of time in q_0. Upon reading symbol a, the clock associated with x gets reset, and state q_1 is entered. Its invariant asserts that at most two time units must be spent in that state, i. e., $x \leq 2$. However, the transition back to q_0 is only enabled when at least one time unit is spent, i. e., $x \geq 1$. Thus, the automaton asserts a delay between a and b of at least one time unit and at most two.

In analogy with Büchi automata, an acceptance condition can be coupled with timed automata and transition tables to define timed languages.

Definition 4.5.12: *A* timed Büchi automaton *is a tuple $\mathcal{A} = (\Sigma, Q, Q_0, E, I, F)$, where (Σ, Q, Q_0, E, I) resembles the constituents of an ordinary timed automaton, and $F \subseteq Q$ a set of accepting states. An infinite run θ of a timed Büchi automaton over a timed word $w \in T\Sigma^\omega$ is called accepting, if and only if Inf$(\theta) \cap F \neq \emptyset$.*

Note, like in the untimed case (see Definition 3.1.12), Inf(θ) is the set of infinitely often reoccurring states by executing θ.

Definition 4.5.13: *A timed language L is also called a* regular timed language *if and only if $L = \mathcal{L}(\mathcal{A})$, where $\mathcal{A}$ is a timed Büchi automaton.*

However, as Alur and Dill [1990] have shown, the class of regular timed languages is not closed under determinsation nor complementation. In consequence, timed (Büchi) automata cannot generally be made deterministic as would be required for fully general model checking (see §3.1).

Theorem 4.5.1 (Alur and Dill [1990]): *The class of timed regular languages is closed under union and intersection.*

Proof:
For a detailed proof of this theorem, see the above cited source.

Timed transition systems constitute the foundation for the analysis of many interesting properties of real-time systems, which are usually based on the notion of *reachability*; that is, given a set of initial states Q_0, one wants to compute the set of all states $q \in Q$ reachable from Q_0 by the transition relation E. Formally, for a timed transition system with initial state (q_0, γ_0), (q_i, γ_i) is said to be *reachable* if and only if $(q_0, \gamma_0) \rightarrow^* (q_i, \gamma_i)$, where $\rightarrow^*$ denotes an arbitrary number of taken transitions from E. However, the reachability problem is non-trivial, since the transition systems can have an infinite number of states due to the presence of implicit delay transitions. Hence, in order to reason about reachability, a finite representation of the state space is necessary.

Clock regions

The foundation for decidability of the above stated problem, i. e., reachability, lies in a separate partitioning of the state space of timed automata in terms of *clock regions*. Hence, this section formally introduces clock regions to meet this objective.

Basically, a clock region is made up of a set of clock valuations. In order to obtain a finite state representation for the analysis of the per se infinite state space of a timed automaton, clock regions and an according equivalence relation are defined. The index of this equivalence relation is given by a so-called *clock ceiling*.

Definition 4.5.14: *A* clock ceiling *is a function* $k : \mathbb{C}_\Sigma \to \mathbb{N}$ *mapping each clock* $z \in \mathbb{C}_\Sigma$ *to a natural number* $k(z)$*, i. e., the ceiling of* z*.*

Formally, the notion of (clock) region equivalence for a timed automaton is then defined as follows.

Definition 4.5.15: *For a number* $d \in \mathbb{R}$*, let* $\{d\}$ *denote the fractional part of* d*, and* $\lfloor d \rfloor$ *its integer part. Two clock valuations* γ, γ' *are* region equivalent*, denoted* $\gamma \mathrel{\dot\sim_k} \gamma'$*, where* $k : \mathbb{C}_\Sigma \to \mathbb{N}$ *is the ceiling function, if and only if*

1. $\forall z \in \mathbb{C}_\Sigma : \lfloor \gamma(z) \rfloor = \lfloor \gamma'(z) \rfloor \vee (\gamma(z) > k(z) \wedge \gamma'(z) > k(z))$
2. $\forall z \in \mathbb{C}_\Sigma : \gamma(z) \leq k(z) \implies (\{\gamma'(z)\} = 0 \Leftrightarrow \{\gamma(z)\} = 0)$
3. $\forall w, z \in \mathbb{C}_\Sigma : \gamma(z) \leq k(z) \wedge \gamma'(w) \leq k(w) \implies (\{\gamma(z)\} \leq \{\gamma(w)\} \Leftrightarrow \{\gamma'(z)\} \leq \{\gamma'(z)\})$

If k maps clocks to the maximum clock constants used in the given timed automaton, one can write $\gamma \mathrel{\dot\sim} \gamma'$ instead of $\gamma \mathrel{\dot\sim_k} \gamma'$. In other words, regions constitute the *equivalence classes* induced by $\dot\sim$. Further, let $[\gamma]$ denote the set of clocks, region-equivalent to γ, and $\mathcal{R}$ the set of all regions.

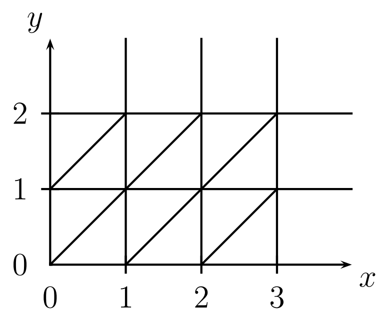


Fig. 4.7: Clock regions for a system with two clock variables, x and y.

Example. The regions, $\mathcal{R}$, of an automaton can be visualised as in Fig. 4.7, which depicts all regions for the following setup: $\mathbb{C}_\Sigma = \{x, y\}$, $k(x) = 3$, and $k(y) = 2$. Notice, x and y not necessarily denote recording or predicting clocks according to

Definition 4.5.2. In this example, however, a total of 60 regions is obtained: 12 corner points, 6 closed diagonal line segments, 17 closed horizontal and vertical line segments, 12 closed regions, and 13 open regions.

Definition 4.5.16: *Let $\mathcal{A} = (\Sigma, Q, Q_0, E, I)$ be a timed automaton. A* region graph *(or* region automaton*) $\mathcal{R}_\mathcal{A}$ is built on states of the form $(q, [\gamma])$ and transitions defined as follows:*

- $(q, [\gamma]) \twoheadrightarrow (q, [\gamma'])$ *if* $(q, \gamma) \xrightarrow{d} (q, \gamma')$ *for some* $d \in \mathbb{R}^{\geq 0}$*, and*
- $(q, [\gamma]) \twoheadrightarrow (q, [\gamma'])$ *if* $(q, \gamma) \xrightarrow{a} (q, \gamma')$ *for some* $a \in \Sigma$*,*

where $q \in Q$ and $\gamma, \gamma' : \mathbb{C}_\Sigma \to \mathbb{R}^{\geq 0} \cup \{\bot\}$ are clock valuation functions defined in the usual way.

The transition relation $\twoheadrightarrow$ is finite, and the foundation for many interesting verification tasks, such as the previously mentioned reachability analysis, i. e., to check whether a state $q \in Q$ is reachable from another state $q' \in Q$ in a timed automaton. However, one problem is that the number of regions, though finite, is exponential both in the number of clocks as well as the maximal clock constants:

Proposition 4.5.1 (Alur and Dill [1990]): *The number of clock regions is bounded by $|\mathbb{C}_\Sigma|! \cdot 2^{|\mathbb{C}_\Sigma|} \cdot \prod_{z \in \mathbb{C}_\Sigma}(2k(z) + 2)$, where $k(z)$ is the clock ceiling for z.*

Thus, this proposition implicitly gives an answer to the initial question of decidability, in that the reachability problem has been shown PSPACE-complete in the cited source.

From clock regions to clock zones

A sometimes more efficient state-space representation for timed automata is made possible by using so-called *clock zones*. In a nutshell, a clock zone describes a number of clock regions, in that it represents (the solutions of) a finite number of conjunctions of clock constraints $z \bowtie c$, where $z \in \mathbb{C}_\Sigma$, $\bowtie \in \{<, \leq, \geq, >\}$ and $c \in \mathbb{N}$.

Definition 4.5.17: *Let ϕ and ϕ' be clock zones and $\lambda \subseteq \mathbb{C}_\Sigma$ a set of clocks. The* operations over clock zones *(used for reachability analysis) are inductively defined as follows:*

Intersection: *$\phi \wedge \phi'$ is called the intersection of two clock zones. (Since both ϕ and ϕ' are clock zones, they can be expressed as conjunctions of clock constraints.)*

Elapsing of time: *For a clock zone ϕ, $\phi^\uparrow$ denotes the elapsing of time and is defined as follows: $\phi^\uparrow = \{\gamma + d \mid \gamma \in \phi, d \in \mathbb{R}^{\geq 0}\}$.*

Reset: *For a finite subset of clocks λ, and a zone ϕ, $r(\phi) = \phi[\lambda := 0]$ denotes the set of clock valuations $\gamma[\lambda := 0]$ for $\gamma \in \phi$.*

Clock zones are closed under all three operations (Bengtsson and Yi [2004]), but unlike regions, do not necessarily form a *stable* equivalence relation over a timed automaton's state space.

Definition 4.5.18: *Let $S = (\Sigma, Q, Q_0, E)$ and $S' = (\Sigma, Q', Q'_0, E')$ be two (abstract) transition systems with the same alphabet Σ. An (equivalence) relation $\sim \subseteq Q \times Q'$ is a* stable (equivalence) relation *(also called a* bisimulation*), if and only if $q \sim p$ and $q \xrightarrow{a} q' \in E$ implies that there exists a p' such that $p \xrightarrow{a} p' \in E'$ and $q' \sim p'$, with $a \in \Sigma$.*

For region equivalence, Alur and Dill [1996] observed:

Lemma 4.5.1: *Let a timed automaton $\mathcal{A} = (\Sigma, Q, Q_0, E, I)$, a state $q \in Q$, and two clock valuations, γ_1 and γ_2 with $\gamma_1 \dot{\sim} \gamma_2$ be given. If for some $a \in \Sigma$, $(q, \gamma_1) \xrightarrow{a} (q', \gamma'_1)$, then there exists a clock valuation γ'_2 such that $\gamma'_1 \dot{\sim} \gamma'_2$ and $(q, \gamma_2) \xrightarrow{a} (q', \gamma'_2)$.*

Proof:
For a proof, see Alur and Dill [1996].

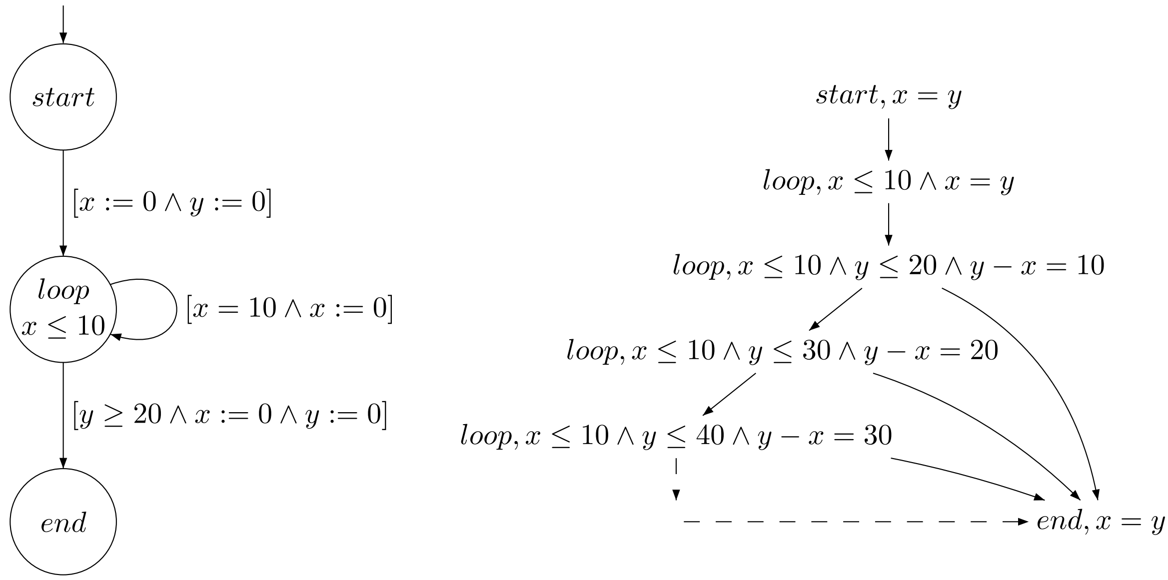
Since the purpose of the region equivalence relation, $\dot{\sim}$, is to quotient an infinite transition system into a finite one, this relation must be generally stable and finite. Using the same notation as in Definition 4.4.4, the following provides for a finiteness criterion.

Definition 4.5.19: *Given a transition system $S = (\Sigma, Q, Q_0, E)$ with a stable equivalence relation and class $[q]_\sim$, where $q \in Q$, then $\sim$ is called a* stable equivalence relation of finite index, *if and only if $\{[q]_\sim \mid q \in Q\}$ is a finite set.*

Together with Lemma 4.5.1 and Proposition 4.5.1, it is easy to see that $\dot{\sim}$ does, indeed, satisfy the criteria implied by Definition 4.5.19.

Example. To illustrate why these requirements are important, consider Fig. 4.8. It shows a timed automaton with transition constraints and invariants (a), as well as the according zone graph (b) which is *infinite*, and thus, not suitable for forward analysis: it contains an infinite number of zones fulfilling the constraints once the state with label *loop* has been reached.

However, it should be pointed out that, in practice, this problem does not seem to occur very frequently, since it applies only to certain time constraints. For instance, the timed verification tool, UPPAAL (Behrmann et al. [2002]), is solely based upon zone graph analysis, and uses a *normalisation function* over the constraints to avoid such problems. Although normalisation can be used to achieve finiteness, this does not guarantee that the zone graph is, under all circumstances, smaller than the region graph. The up- and down-sides of either representation (when normalised accordingly), however, are not topic of this chapter. For a brief discussion, see also §7.



(a) Timed automaton. (b) Zone graph.

Fig. 4.8: A timed automaton and its (infinite) zone graph.

Like a region automaton (see Definition 4.5.16), clock zones of a timed automaton can be captured and processed by a so-called (normalised) *zone automaton* which is defined as follows.

Definition 4.5.20: *Let $\mathcal{A} = (\Sigma, Q, Q_0, E, I)$ be a timed automaton. A* zone graph *(or* zone automaton*) $\mathcal{Z}_{\mathcal{A}}$ is built on symbolic states of the form (q, ϕ) and symbolic transitions defined as follows:*

- $(q, \phi) \rightsquigarrow (q, \phi^{\uparrow} \wedge I(q))$
- $(q, \phi) \rightsquigarrow (q', r(\phi \wedge \psi) \wedge I(q'))$ *if* $l \xrightarrow{a, \psi, \lambda} q'$

where $q, q' \in Q$, $a \in \Sigma$, $\psi \in \Psi(\mathbb{C}_{\Sigma})$, $\lambda \subseteq \mathbb{C}_{\Sigma}$, and ϕ is a clock zone.

This *symbolic semantics* is then a sound and complete characterisation of the operational semantics of timed automata (Bengtsson and Yi [2004]). Notice a very efficient, in terms of time and space complexity, realisation and representation of zones can be achieved using so-called *difference bound matrices* (DBM, Dill [1989]). Many timed verification systems, such as UPPAAL use DBMs internally. Since DBMs will not play a significant role in this thesis, they are not developed further at this point, and instead a reference to the relevant literature is given.

Event-clock automata

Systems verification using model checking, as outlined in §3.1.1, involves the complementation of the automaton which represents the specified property of a system, which in turn relies upon determinisation. However, timed automata as described in the previous section are not generally determinisable, and important verification problems (e. g., language inclusion as in checking whether $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$) are undecidable using this framework (see also Theorem 4.5.1).

Event-clock automata as proposed by Alur et al. [1999] tackle this problem, in that they represent a class of timed automata which are determinisable, i. e., their non-deterministic variants are equally expressive as their deterministic counterparts. Moreover, Raskin [1999] gave a translation of TLTL to event-clock automata, and provided solutions to the model checking problem of this logic. In a nutshell, this is based upon a translation of event-clock automata to ordinary Alur-Dill automata and then performing the main analysis on them by means of building region or (stable) zone graphs.

Definition 4.5.21: *Let $\mathbb{C}_\Sigma$ be a finite set of clocks. An* event-clock automaton *is defined as a finite-state automaton whose edges are annotated both with input symbols and with clock constraints as $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$, where*

- *Σ is a finite input alphabet,*
- *Q a finite set of locations,*
- *$Q_0 \subseteq Q$ is a distinguished set of initial locations,*
- *$E \subseteq Q \times \Sigma \times \Psi(\mathbb{C}_\Sigma) \times Q$ a set of transitions, and*
- *$\mathcal{F} \subseteq 2^Q$ is a set of accepting components.*

An edge $e = (q, a, \psi, q') \in E$ represents a transition from source location $q \in Q$ upon symbol $a \in \Sigma$ to destination $q' \in Q$, where the clock constraint $\psi \in \Psi(\mathbb{C}_\Sigma)$ specifies when this transition is enabled.

For an event-clock automaton $\mathcal{A}_{ec}$, let $K_{\mathcal{A}_{ec}}$ denote the biggest constant, i. e., clock ceiling, appearing in some constraint of $\mathcal{A}_{ec}$; in this thesis, simply K is used whenever $\mathcal{A}_{ec}$ is clear from the context.

Without loss of generality, event-clock automata using only history clocks are referred to as *event-recording automata*, whereas automata using only predicting clocks are referred to as *event-predicting automata*. Alur et al. [1999] have shown that both classes are strictly less expressive than fully general event-clock automata, and can not be generally converted into one another.

Event-clock automata as used here also differ from Alur-Dill automata, in that they possess accepting states, but lack clock constraints as state invariants. However, as Alur et al. [1999] point out, the latter does not influence the expressiveness of the computational model. Moreover, the event-clock automata used throughout this thesis are defined over the domain $T\Sigma^\omega$ as also used by D'Souza [2003], Raskin and Schobbens [1999], instead of $T\Sigma^*$.

Definition 4.5.22: *An (infinite)* timed run θ *of an automaton* $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, \mathcal{F})$ *over a timed word* $w \in T\Sigma^\omega$ *starting in* (q_0, γ_0) *is an (infinite) sequence of state-valuation tuples and transitions as follows:*

$$\theta : (q_0, \gamma_0) \xrightarrow{\alpha_1} (q_1, \gamma_1) \xrightarrow{\alpha_2} \dots$$

with $q_i \in Q$*, and* γ_i *being the evaluation function assigning for every element from* Σ *the value of the recording and predicting event clocks corresponding to* α_i*, where* $\alpha_i \in T\Sigma$ *is a timed event of the form* $(a_i \in \Sigma, t_i \in \mathbb{R}^{\geq 0})$*, and for all* $i \geq 1$ *there is a transition in* E *of the form* $(q_{i-1}, a_i, \psi, q_i)$ *such that* $\gamma_i \models \psi$*.*

Definition 4.5.23: *An infinite run* θ *of* $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, \mathcal{F})$ is accepting*, if and only if for all accepting components* $F_i \in \mathcal{F}$: *Inf*$(\theta) \cap F_i \neq \emptyset$ *(generalised Büchi acceptance).*

γ_0 is *initial* (with respect to w) if $\gamma_0(x_a) = \bot$ and $\gamma_0(y_a) = t_i$ if $\alpha_i = (a, t_i)$ and for $j < i$ and $\alpha_j = (a_j, t_j)$, $a_j \neq a$, and $\gamma_0(y_a) = \bot$ if a does not occur in w. Then, the timed language accepted by $\mathcal{A}_{ec}$, denoted as $\mathcal{L}(\mathcal{A}_{ec})$, is the set of timed words for which an accepting run of $\mathcal{A}_{ec}$ exists starting in (q_0, γ_0), for some $q_0 \in Q_0$ and the initial γ_0.
For brevity, the translation algorithms of the real-time logic TLTL to event-clock automata are not fully restated; these are given, e. g., by Raskin [1999].

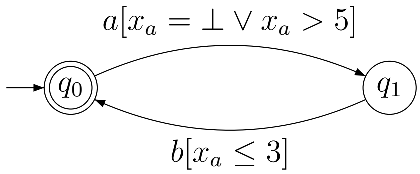


Fig. 4.9: Example event-clock automaton.

Example. Fig. 4.9 depicts an example event-clock automaton. It is easy to recognise it, more precisely, as an event-recording automaton using only x_a as a history clock. It accepts timed words $w \in T\Sigma^\omega$ of the form $L = [\![(a + b)^\omega]\!]$, where the Kleene-star has been replaced with the ω-operator to express infinity (see also example on p. 37), such that the time difference between each consecutive a and b is within three time units (i. e., $x_a \leq 3$), and that two a's are separated by at least five time units (i. e., $x_a = \bot \vee x_a > 5$). The $\bot$ symbol is necessary to deal with the initial case.

4.5.3 A 3-valued semantics for TLTL—TLTL$_3$

In order to develop a monitoring procedure for real-time systems which works schematically similar to the one described in §4.3, it is necessary to introduce a 3-valued semantics for TLTL first. Hence, in this section TLTL_3 is formally introduced, a 3-valued interpretation of TLTL.

Definition 4.5.24: *Let $u \in T\Sigma^*$ denote a finite timed trace. The truth value of a TLTL_3 formula φ with respect to u, denoted $[u \models \varphi]$, is an element of $\mathbb{B}_3$ and defined as follows:*

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \text{ such that } u\sigma \in T\Sigma^\omega,\ u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \text{ such that } u\sigma \in T\Sigma^\omega,\ u\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases}$$

4.5.4 Dynamic decision procedure

This section develops the dynamic, automata-based decision procedure for the logic introduced in the previous section, TLTL_3. The automata used are basically event-clock automata as introduced formally in §4.5.2, but evaluated *symbolically* over the set $\mathbb{B}_3$ instead of $\mathbb{B}$. The alternative symbolic evaluation of event-clock automata becomes a necessity in the runtime reflection framework, because clock constraints over predicting clocks cannot be evaluated in a straightforward manner when only finite prefixes of behaviours are at hand.

Recall, a run through an event-clock automaton resembles a stream of state-valuation tuples (q, γ) (see Definition 4.5.23), where γ assigns to the event-recording and predicting variables, each associated with a symbol a from the event alphabet, either the time when it was seen last (i. e., $\gamma(x_a)$), or when it has to be seen next (i. e., $\gamma(y_a)$). While the past valuations can be determined or $\bot$ assigned, future valuations obviously pose a problem: the exact time when the next appropriate event occurs is not yet known.

A *symbolic evaluation of clock constraints* over $\mathbb{B}_3$ circumvents this dilemma.

Symbolic execution of event-clock automata

Definition 4.5.25: *Let $\Gamma : \mathbb{C}_\Sigma \to \mathbb{R}^{\geq 0} \cup \{\bot\} \cup \mathcal{I}$ be a* symbolic clock evaluation, *which assigns a real or undefined value, $\bot$, to each recording clock variable $x_a \in \mathbb{C}_\Sigma$, and an interval from $\mathcal{I}$ or undefined value to each predicting clock variable $y_a \in \mathbb{C}_\Sigma$. The operations defined over Γ are:*

Elapsing of time: *Given elapsed time, $t \in \mathbb{R}^{\geq 0}$, $\Gamma' = \Gamma + t$, where $\Gamma'(x_a) = \Gamma(x_a) + t$ and for $\Gamma(y_a) = [(l, r)]$, let $\Gamma'(y_a) = [(l \dot{-} t, r - t)]$, where $\dot{-}$ yields at least 0; if $r - t < 0$, then Γ'* is invalid.

Reset: Γ reset *by action a, denoted as $\Gamma \downarrow a$, sets $x_a = 0$, and removes all constraints on y_a, setting $\Gamma'(y_a) = [0, \infty)$ and $\Gamma'(z_b) = \Gamma(z_b)$ for all $b \neq a$ and $z \in \mathbb{C}_\Sigma$.*

Conjunction: *The* conjunction *of Γ with constraint $\psi \in \Psi(\mathbb{C}_\Sigma)$ yields $\Gamma' = \Gamma \wedge \psi$, where each predicting clock y_a is combined with the constraints of ψ which* *involve y_a (i. e., $\Gamma'(y_a) = \Gamma(y_a) \wedge \bigwedge\{y_a \bowtie c \subseteq \psi\}$). Γ' is* invalid, *if for some y_a, $\Gamma'(y_a)$ is not satisfiable.*

Intuitively, the symbolic evaluation of clock constraints occurring in an event-clock automaton is meant to assign, instead of concrete values to a predicting clock variable y_a, the valid intervals, concrete values of y_a can have, such that an accepting run through the automaton is still possible.

A transition $(q, a, \psi, q') \in E$ of an event-clock automaton defined in the typical manner, is referred to as *applicable* to a pair (q, Γ), if and only if the constraints $x_b \bowtie c$ in ψ are satisfied by Γ for all $b \in \Sigma$ and $0 \in \Gamma(y_a)$. For instance, $\Gamma(y_a) = [0, 5]$ would *allow* an action a to occur, say, in 3 or 4 time units from the current instant of time, whereas $\Gamma(y_a) = [0, 0]$ would make the occurrence *mandatory* at the current instant of time. Additionally, if a transition (q, a, ψ, q') is applicable, then the successor of the corresponding (q, Γ) is (q', Γ'), where $\Gamma' = (\Gamma \downarrow a) \wedge \psi$ (i. e., reset and conjunction).

A *symbolic timed run* which is used for the purpose of monitoring, is then defined as follows.

Definition 4.5.26: *A* symbolic timed run Θ *of an event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, \mathcal{F})$ over a timed word $w \in T\Sigma^\omega$ starting in (q_0, Γ_0) is an infinite sequence of state-symbolic-valuation tuples and transitions as follows:*

$$\Theta : (q_0, \Gamma_0) \xrightarrow{\alpha_1} (q_1, \Gamma_1) \xrightarrow{\alpha_2} \ldots$$

with $q_i \in Q$, and Γ_i being a symbolic valuation function, where for each $(q_{i-1}, \Gamma_{i-1}) \xrightarrow{(a_i,t_i)} (q_i, \Gamma_i)$, there exists some transition $(q_{i-1}, a_i, \psi, q_i)$ applicable to $(q_{i-1}, \Gamma_{i-1} + t_i)$ and (q_i, Γ_i) is the result of this application. Γ_0 is called initial *if $\Gamma_0(x_a) = \perp$ and $\Gamma_0(y_a) = [0, \infty)$.*

Note that the notion of acceptance for symbolic runs corresponds to that of runs, i. e., for each $F_i \in \mathcal{F}$ there is some $q \in F_i$ occurring infinitely often.

This connection defined between symbolic timed runs and actual timed runs of an event-clock automaton is summed up again formally in the following theorem.

Theorem 4.5.2: *If $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, \mathcal{F})$ is an event-clock automaton and $w \in T\Sigma^\omega$, then there is an accepting run on w starting in (q_0, γ_0), if and only if there is a symbolic accepting run on w starting in (q_0, Γ_0).*

The important fact about this theorem is that γ_0 is dependent on w since the clock variable y_a has to be initialised to match the first occurrence of a, while Γ_0 is independent of w. Thus, symbolic runs form are a suitable device for runtime verification.

Region automata construction

In what follows, it is assumed (due to a construction given by Raskin [1999]) that for a formula φ as well as its negation, an event-clock automaton is given, accepting precisely the models, respectively counterexamples, of φ and $\neg\varphi$.

In accordance with the scheme developed for LTL_3, it may be tempting to merely check for every state q of an event-clock automaton, whether the accepted language from that state is empty. However, this would yield wrong conclusions, as can be seen in the following example.

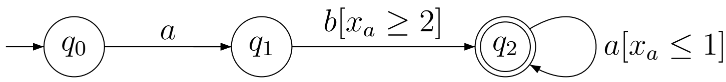


Fig. 4.10: An event-clock automaton.

Example. While the language accepted in state q_2 is non-empty and, despite, state q_2 is reachable, the automaton does not accept any word when starting in state q_0. The constraint when passing from q_1 to q_2 requires the clock x_a to be at least q_2. This, however, restricts the loop in state q_2 to be taken.

The proposed solution to this problem is to work on the region automaton of a given event-clock automaton instead. The key property of region equivalence is *stability*; that is, given a state s and two equivalent valuations, γ_1 and γ_2, then (s', γ') is an a-successor of (s, γ_1) if and only if (s', γ'') is one of (s, γ_2) for a suitable γ'' equivalent to γ' (see also Definition 4.5.1). This can also be "lifted" to infinite runs according to the following lemma.

Lemma 4.5.2: *Let $\mathcal{A}_{ec}$ be an event-clock automaton. Let q be some state of $\mathcal{A}_{ec}$ and γ_1, γ_2 two valuations with $\gamma_1 \dot{\sim} \gamma_2$. Let $\bar{w} \in \Sigma^\omega$. Then, there exists an accepting run on some infinite timed word $w_1 \in T\Sigma^\omega$ with $ut(w_1) = \bar{w}$ starting in (q, γ_1), if and only if there exists an accepting run on some infinite timed word $w_2 \in T\Sigma^\omega$ with $ut(w_2) = \bar{w}$ starting in (q, γ_2).*

Proof:
Stability as defined above is a property over pairs of state-valuation tuples, respectively. Thus, this lemma can be shown by induction over $n \in \mathbb{N}$, where n denotes the position in $\bar{w}$. □

Again, notice, that zone graphs do not generally possess the stability criterion. Hence, they have not been chosen as a vehicle for monitoring TLTL_3 properties (see §4.5.2). Region automata were chosen mainly to keep the presentation focused on the basic concepts. However, since the key property of the monitor construction is stability of the region equivalence, the approach could, in fact, be improved by taking a coarser but *stable* partition of the underlying timed transition system into account as well. Examples of such stable partitions have been studied extensively by Tripakis and Yovine [2001].

The following construction of a region automaton as defined in Definition 4.5.16 from an event-clock automaton as defined in Definition 4.5.21 is based upon the one presented by Raskin and Schobbens [1999].

Definition 4.5.27: *Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, \mathcal{F})$ be an event-clock automaton defined in the usual way, and $\mathcal{R}$ denote its regions. The* region automaton *of $\mathcal{A}_{ec}$ is the (generalised) Büchi automaton $R(\mathcal{A}_{ec}) = (\Sigma^r, Q^r, Q_0^r, E^r, \mathcal{F}^r)$, where*

- $Q^r = \{(l, \kappa, \zeta) \mid l \in Q, \kappa \in \mathcal{R}, \zeta \in \{t, d\}\}$ *is the set of states,*
- $Q_0^r = \{(l, \kappa, \zeta) \in Q^r \mid l \in Q_0, \forall a \in \Sigma : \kappa(x_a) = \bot, \zeta = d\}$ *is the set of initial states,*
- $\Sigma^r = \Sigma \cup \{\epsilon\}$
- $E^r = E_d^r \cup E_t^r$ *is the union of untimed and timed transitions, where*
  - $E_d^r = \{((l_1, \kappa_1, t), (l_2, \kappa_2, d), a) \mid (l_1, a, \psi, l_2) \in E$ *and* $\exists \kappa_3$ *such that* $\kappa_1 = \kappa_3[y_a := 0], \kappa_2 = \kappa_3[x_a := 0],$ *and* $\kappa_3 \models \psi\}$
  - $E_t^r = \{((l, \kappa_1, d), (l, \kappa_2, t), \epsilon) \mid \kappa_2 \in TS(\kappa_1)\}$
- $\mathcal{F}^r = \{F_i^r \mid F_i \in \mathcal{F}\} \cup \{F_{x_a} \mid \lhd_a \in I \in cl(\varphi)\} \cup \{F_{y_a} \mid \rhd_a \in I \in cl(\varphi)\}$,
  - *where for* $F_i \in \mathcal{F}$, $F_i^r = \{(l, \kappa, \zeta) \mid l \in F_i\}$
  - $F_{x_a} = \{(l, \kappa, \zeta) \mid \forall \gamma \in \kappa : \gamma(x_a) = 0 \vee \gamma(x_a) > c \vee \gamma(x_a) = \bot\}$
  - $F_{y_a} = \{(l, \kappa, \zeta) \mid \forall \gamma \in \kappa : \gamma(y_a) = 0 \vee \gamma(y_a) = \bot\}$,

and $TS(\kappa_1)$ is a time successor *of a clock region κ_1, denoted $\kappa_2 \in TS(\kappa_1)$, if and only if for all $\gamma \in \kappa_1$ there is some $t \in \mathbb{R}^{\geq 0}$ such that $\gamma + t \in \kappa_2$.*

Since the region automaton as defined here is, basically, a Büchi automaton, the accepted language by it is a sequence of (untimed) words over Σ. Thus, it is straightforward to compute for every state, whether the accepted (untimed) language is empty or not in the way as done in §4.4, i. e., the procedure is analogous. For every state (l, κ, ζ) with a non-empty language, stability now guarantees that for each $\gamma \in \kappa$, there is some accepting run of the corresponding event-clock automaton starting in (l, γ) for some timed word w. Dually, if the accepted language is empty, then the corresponding event-clock automaton has no accepting run starting in (l, γ) for any $\gamma \in \kappa$, and any w.

Monitoring algorithm

Using the symbolic execution scheme and the derivation of region automata as described above, an actual monitoring procedure for TLTL_3 can now be derived, i. e., execution of a finite state machine that reads timed words and decides whether further events might yield and accepting run, or not.

The monitoring procedure is based on both the event-clock automaton as well as the region automaton for a formula in TLTL_3. It follows the possible *symbolic* computations for the given input along the lines of the event-clock automaton. However, to decide, whether future events might contribute to an accepting run, the region automaton is consulted according to Algorithm A (see below).

Unlike otherwise noted, for the remainder, an event-clock automaton $\mathcal{A}^{\varphi}_{ec}$ and its region automaton $R(\mathcal{A}^{\varphi}_{ec})$ both defined in the usual way are assumed. Moreover, a timed word $w = (a_0, t_0)(a_1, t_1) \cdots \in T\Sigma^{\omega}$ is assumed, where (a_0, t_0) denotes the first action a_0 occurring at time t_0.

Algorithm A (*Automata execution*). Let Γ_0 be the initial symbolic valuation of $\mathcal{A}^{\varphi}_{ec}$ and l_0 one of the initial states of $\mathcal{A}^{\varphi}_{ec}$.

A1. [Compute successor set.] For the first event (a_0, t_0), the set of successors with respect to $\mathcal{A}^{\varphi}_{ec}$ is computed.

A2. [Set empty?] If this set is empty, the underlying formula is obviously violated, and *false* issued. If not, go to step A3.

A3. [Check emptiness.] Each successor is a pair (l, Γ) and corresponds to a set of states in the region automaton. If and only if for all of them the accepted language is empty, the underlying property is violated, and *false* issued (see Theorem 4.5.2 and Lemma 4.5.2).

A4. [Process next event.] Issue *true*, and continue procedure from A2 with each successor state (l, Γ) for which a corresponding accepting state of $R(\mathcal{A}^{\varphi}_{ec})$ exists, reading a new input event. ∎

Thus, the generated procedure keeps a set of possible state-symbolic valuation pairs that represent the possible current states of $\mathcal{A}^{\varphi}_{ec}$ (giving credit to the non-deterministic nature of $\mathcal{A}^{\varphi}_{ec}$). Furthermore, the transition table of $\mathcal{A}^{\varphi}_{ec}$ and the states of $R(\mathcal{A}^{\varphi}_{ec})$ enriched with emptiness per state information can be stored as look-up tables.

However, according to the procedure used in the untimed case, this algorithm captures only half of the steps actually required to determine $[u \models \varphi]$, where $w = u\sigma$ and $\sigma \in T\Sigma^{\omega}$, and $\varphi \in \mathrm{TLTL}_3$. To determine validity of a property, the negated property needs to be checked as well in the same manner as described by the algorithm.

The overall monitoring procedure for TLTL_3 is thus, summarised in Fig. 4.11, where ECA denotes an event-clock automaton. Notice, the main difference compared to the untimed case lies in that no nondeterministic finite automaton is constructed explicitly, nor is determinsation included explicitly. Instead, the actual monitor implementing the above steps for $\mathcal{A}^{\varphi}_{ec}$, respectively $\mathcal{A}^{\neg\varphi}_{ec}$, evaluates $R(\mathcal{A}^{\varphi}_{ec})$, respectively $R(\mathcal{A}^{\neg\varphi}_{ec})$, in an *on-the-fly manner*; that is, both automata are evaluated by dynamic power-set construction as described in more detail in §6 of this work (or, alternatively,

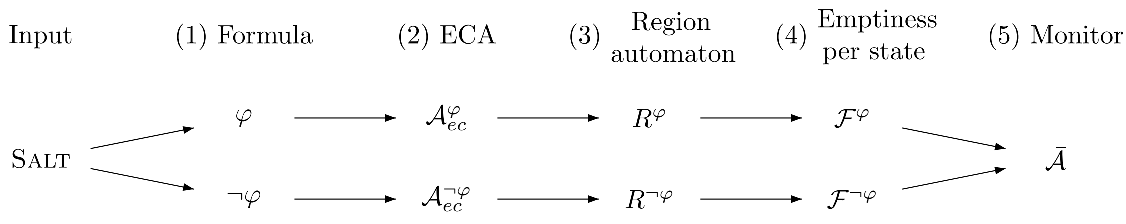


Fig. 4.11: The procedure for getting $[u \models \varphi]$ for a given $\varphi \in \mathrm{TLTL}_3$ derived from a SALT specification.

see also Hopcroft and Ullman [1979] and Aho et al. [1988] for an explanation of the general power-set construction).

Remark. To enhance the practical applicability, the procedure can be adjusted as follows: the formal framework described above requires the monitor to raise an alarm if and only if for some prefix $(a_0, t_0) \dots (a_i, t_i)$ no accepting run exists. In particular, it is assumed that "a digital watch is consulted only when some action occurs". But the time transitions yielding the subsequent regions in the region automaton actually (often) constrain the possible occurrence of some future event a. For each current valuation Γ corresponding to a set of regions, $R(\mathcal{A})$ is consulted for the possible accepting time successors and to compute a maximal time bound before some event has to occur for reaching an accepting state. Thus, in practice, a timer *interrupt* can be set, when such a bound exists, and a trace can be rejected, when a timeout occurs before a suitable action has been read.

4.5.5 Complexity

Complexity evaluation largely follows §4.4.3; that is, considering Fig. 4.11 again, step 1 merely requires replication of φ and negation. Thus, it is linear in the original size of the formula. According to Proposition 4.5.1, the region automaton of $\mathcal{A}_{ec}^{\varphi}$, respectively $\mathcal{A}_{ec}^{\neg\varphi}$, is worst-case exponential with respect to the length of the underlying formula φ as well as the largest constant K appearing in φ.

Although, so far, the procedure is "only single exponential", a penalty is introduced by the on-the-fly power-set construction to determine successor states of the automata; that is, following all the different paths in a nondeterministic automaton for a given prefix involves further space requirements exponential in the size of the according automaton. Further details and a discussion of possible improvements are available in §6.

4.6 Summary

This chapter introduces a construction for runtime monitors that work with timed and untimed systems using LTL, and TLTL for the timed case. Due to D'Souza's results, TLTL can be considered a natural extension of LTL towards real-time. As such, a typical obstacle in runtime verification is solved both for untimed and timed formulae, in that standard models of linear temporal logic are infinite traces, whereas in runtime verification only finite system behaviours are at hand. Therefore, a 3-valued semantics (*true*, *false*, ?) for LTL and TLTL on finite traces is defined that resembles the infinite trace semantics in a suitable and intuitive manner (i. e., not conflicting with the traditional 2-valued semantics). Further, this chapter describes how to construct, given a formula in LTL, respectively TLTL, an optimal deterministic monitor with three output symbols that reads a finite trace and yields its according 3-valued semantics. Notably, the monitor rejects a trace as early as possible, in that any minimal bad prefix results in *false* as a return value.

Chapter 5

Fault detection using model-based diagnosis

> How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?
>
> *(Sherlock Holmes in* The Sign of Four*)*

This chapter explores model-based diagnosis as a means of deducing from observed behavioural deviations, explanations for them; that is, an efficient realisation of diagnosis is presented that uses the results of the monitors to detect actual faults in a system.

Although it is not explicitly stated by the communities focussed on the static verification of systems, such as model checking, their focus rests on the detection of failures, rather than the question of what to do, once a symptom or a failure has been found. This chapter proposes diagnosis as a subsequent task, in order to perform a plausibility check, and to determine whether an observed symptom for failure hints to the actual fault in a system, or whether the fault must be looked for elsewhere, probably even outside the scope of the system, in its environment.

The procedure proposed in this chapter aims to remedy this downside which, in its present form, also exists in most runtime verification approaches that do not distinguish between an observed symptom of a fault (e. g., violation of a safety property), and the actual fault itself (e. g., communication failure between components or a deadlock in a single component). Neither the employed models (i. e., LTL formulae), nor the theory behind runtime verification currently cater for such a differentiation.

Like runtime verification, model-based diagnosis also relies upon complementary system models which form a suitable foundation for this kind of analysis. However, different types of diagnosable systems as well as means of specification have also led to different approaches to model-based diagnosis; a term which was first used in this context in a seminal paper by de Kleer and Williams [1987]. The underlying system model used by de Kleer and Williams captures both the behaviour of a system, and, more importantly for the approach developed in this chapter, the causality between its constituents or individual components.

The task of model-based diagnosis is summarised in Fig. 5.1: From the system model a specific behaviour is predicted, which is compared to a set of observations that are obtained from the actual system during its execution. From the (possibly empty) set of discrepancies, the diagnoses, i. e., explanations for the discrepancies, are then derived; if empty, the system should work as expected.

$$
\begin{tikzcd}
\text{System model} \arrow[d] & \text{Diagnosis} & \text{System} \arrow[d] \\
\text{Derived behaviour} \arrow[r] & \text{Discrepancy} \arrow[u] & \text{Observation} \arrow[l]
\end{tikzcd}
$$

Fig. 5.1: The principle of model-based diagnosis.

The rest of this chapter is structured as follows. In §5.1 a formalisation of the model-based diagnosis problem in first-order logic is given. It also outlines the two most well-known approaches to diagnostic problem solving in first-order logic, namely the *Diagnose*-algorithm proposed by Reiter [1987], and the *General Diagnostic Engine*, proposed by de Kleer and Williams [1987]. In §5.2 an alternative and practically more efficient solution to the diagnosis problem is presented, which follows the idea depicted in Fig. 5.1, but uses propositional logic instead. As such, it provides means for efficient realisation in terms of mapping the diagnosis problem to a Boolean satisfiability problem. Hence, this section discusses the mapping from first-order to propositional diagnosis as well as a straightforward implementation based on the well-known DPLL-algorithm (see §5.2.2). Then, in §5.2.3 an optimisation for this implementation is proposed which not only determines a diagnosis based on a satisfiability check, but also caters for determination of *all* potentially possible diagnoses, based on the cardinality of system faults found. For systems diagnosis this is essential, since more than one diagnosis for an observed failure may be required to explain it. Related work is summarised in §5.3, whereas §5.4 provides a brief summary of the chapter.

5.1 Preliminaries

The term model-based diagnosis as used throughout the remainder of this thesis reflects the theory introduced first by Reiter [1987], under the name “diagnosis from first principles” or consistency-based diagnosis, and independently but almost at the same time by de Kleer and Williams [1987] as *model-based diagnosis*. In the following, the basic concepts of model-based diagnosis are introduced which have their background in first-order reasoning, followed by a discussion and comparison of two established algorithms for solving the formal problem of model-based diagnosis.

5.1.1 Languages of first-order logic

To keep this chapter self-contained and to substantiate the transition from first-order to propositional diagnosis models, this section briefly recalls the formal semantics of *first-order logic languages*, and introduces some essential notation.

The *syntax of a first-order logic language* is defined over a *first-order alphabet* consisting of the following classes of symbols.[1]

Definition 5.1.1: *A* first-order logic language, $\mathcal{L}_{FO}(R, F, C)$, *may consist of the following classes of symbols:*

- *A countably infinite set of variables*

$$V = \{u, v, w, x, y, z, U, V, W, X, Y, Z, \ldots\}.$$

- *A set of logical operators* $\{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$.
- *A finite set R of relation symbols. For each $P \in R$, there exists an $n \in \mathbb{N}$ denoting the arity of P.*
- *A finite set F of function symbols. For each $f \in F$, there exists an $n \in \mathbb{N}$ denoting the arity of f.*
- *A finite set C of constants.*
- *A set $\{\exists, \forall\}$ of quantifiers, where $\exists$ is called the existential and $\forall$ the universal quantifier.*

In the rest of this chapter some abbreviations will be used. $\mathcal{L}_{FO}$ is used instead of $\mathcal{L}_{FO}(R, F, C)$, when R, F, C are clear from the context. The notation $\mathcal{F}(\mathcal{L}_{FO})$ is used to denote the set of function symbols appearing in $\mathcal{L}_{FO}$, $\mathcal{R}(\mathcal{L}_{FO})$ for the relation symbols, $\mathcal{C}(\mathcal{L}_{FO})$ for the constants, and $\mathcal{V}(\mathcal{L}_{FO})$ for all the occurring variables.

Given a set of functions F, then $f^n \in F$ denotes a function f with arity n. Respectively, given a set of relation symbols R, then $P^n \in R$ denotes an n-ary relation P.

The languages of first-order logic are expressible by *first-order (logic) sentences* and *terms* as defined below.

Definition 5.1.2: *A* first-order term *is inductively defined as follows:*

- *A variable $v \in V$ is a term.*

[1] Standard Edinburgh Prolog notation (Clocksin and Mellish [1987]).

- *Let* $f^n \in F$ *be a function symbol, and* $t_1, \ldots, t_n$ *be terms, then* $f(t_1, \ldots, t_n)$ *is also a term.*

However, in the remainder, some mathematical functions are written using *infix* instead of *prefix* notation, e. g., $u + v \times x$ is used for $+(u, \times(v, x))$.

Definition 5.1.3: *A* first-order sentence *is inductively defined as follows:*
- *Let* $P^n \in R$ *be a relation symbol, and* $t_1, \ldots, t_n$ *be terms, then* $P(t_1, \ldots, t_n)$ *is a sentence.*
- *Let* t_1 *and* t_2 *be terms, then* $t_1 = t_2$ *is a sentence.*
- *Let* $v \in V$ *and* F, G *be sentences, then so are* $\neg F$, $F \wedge G$, $F \vee G$, $F \Rightarrow G$, $F \Leftrightarrow G$, $\exists v(F)$, *and* $\forall v(F)$.

The *semantics of a first-order logic language* is given by a *structure* defined below, and a function which assigns to each variable occurring in a sentence an element from that structure.

Definition 5.1.4: *A* structure *for the first-order language* $\mathcal{L}_{FO}$ *is a pair* $\mathcal{M} = (D, I)$, *where*
- D *is a nonempty set called the* domain, *and*
- I *an* interpretation *over* D *which provides*
  - *for every* $c \in \mathcal{C}(\mathcal{L}_{FO})$, *an* $I(c) \in D$,
  - *for every* $f \in \mathcal{F}(\mathcal{L}_{FO})$, *a mapping* $I(f) : D^n \to D$, *where* $n \in \mathbb{N}$ *is the arity of* f,
  - *for every* $P \in \mathcal{R}(\mathcal{L}_{FO})$, *an* $I(P) \subseteq D^n$, *where* n *is the arity of the relation.*

For brevity, the concept of interpretation is not expanded any further at this point. Details are available, e. g., in Fitting [1996] and Harrison [TBA]. In these works, first-order structures are also referred to as "models" of first-order languages. However, to not confuse a first-order model of a language with the behavioural model of a computer system, the term "structure" is used here. For brevity, also the notations c^I, f^I, P^I will be used instead of $I(c)$, $I(f)$, and $I(P)$.

5.1.2 First-order diagnosis

The system models used in model-based diagnosis as introduced by Reiter [1987] are first-order sentences encoding all relevant aspects of a system with respect to its diagnosability. The system itself is defined in a domain-independent manner as follows.

Definition 5.1.5: *A* system *is a pair* $(SD, COMP)$, *where*
- SD *is the* system description, *encoded by first-order sentences;*

- *$COMP$ is the set of* system components, *encoded by a finite set of constants.*

In practice, the set of components can be of almost arbitrary granularity. Depending on the properties of the system to be diagnosed, $COMP$ may, for example, refer to Java threads, user session objects within a Web application, or even physical entities such as sensors, actuators, or entire nodes in a computer network. This definition makes no assumption on the purpose of a system as it is domain-independent.

However, in all intended applications, the system description will contain an unary predicate, $AB(\cdot)$, encoding that something is "abnormal". The intuitive interpretation of AB-predicates in a system description follows that of McCarthy [1987] who rests his formalisation of circumscription upon this predicate; he writes $AB(c)$ to denote an abstract component $c \in COMP$ with an abnormal behaviour or characteristic (see also §5.3).

In the classical theory of diagnosis, the diagnostic task is, given a set of observations and a system description with AB-predicates over a set of components, to determine whether the system under scrutiny is malfunctioning, which components may be faulty or abnormal, and what additional information needs to be gathered (if any) to identify the faulty components with relative certainty (cf. de Kleer and Kurien [2003]). To formalise this, the notion of an *observation* needs to be introduced.

Definition 5.1.6: *An* observation *of a system is a finite set of first-order sentences. With $(SD, COMP, OBS)$ the system $(SD, COMP)$ with observation OBS is denoted.*

The next example brings these concepts, i. e., that of a system description and observations, together.

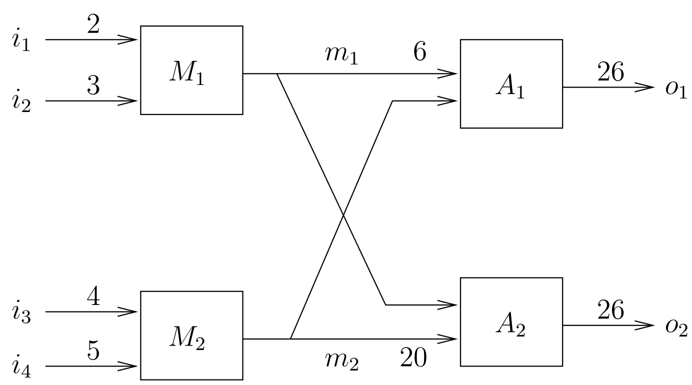


Fig. 5.2: A network with four arithmetic components.

Example. The distributed system depicted in Fig. 5.2 contains four arithmetic components, i. e., two multiplicators, M_1 and M_2, and two adders, A_1 and A_2. The following schematic system description applies:

$$SD = \left\{ \begin{array}{l} mult(X) \wedge \neg AB(X) \Rightarrow (output_1(X) = input_1(X) \times input_2(X)), \\ add(X) \wedge \neg AB(X) \Rightarrow (output_1(X) = input_1(X) + input_2(X)), \\ mult(M_1), mult(M_2), add(A_1), add(A_2), \\ output_1(M_1) = input_1(A_1), \\ output_1(M_1) = input_1(A_2), \\ output_1(M_2) = input_1(A_1), \\ output_1(M_2) = input_1(A_2) \end{array} \right\},$$

where the input and output "channels" of the system have been given shortcuts in the diagram to make the example, and coming references to it more applicable. The first two sentences in SD are referred to as *behavioural model*, whereas the remaining ones are referred to as *structural model* capturing only the causality of the overall system.

The set of components is defined as

$$COMP = \{M_1, M_2, A_1, A_2\}.$$

In this example, the set of observations is defined as:

$$OBS = \left\{ \begin{array}{l} input_1(M_1) = 2, input_2(M_1) = 3, output_1(M_1) = 6, \\ input_1(M_2) = 4, input_2(M_2) = 5, output_1(M_2) = 20, \\ output_1(A_1) = 26, output_1(A_2) = 26 \end{array} \right\}.$$

Here, OBS can also be expressed by a mapping of signals, exchanged on channels, denoted by SIG, as $OBS : SIG \rightarrow \mathbb{N}$, such that an assignment $i_1 \mapsto 2$ is a shortcut for $(input_1(M_1) = 2)$. In the remainder of this chapter, both notations will be used interchangeably where fit. $\neg AB(M_1)$ denotes that, given M_1 is working as expected, its output corresponds exactly to the product of its inputs, respectively for M_2, A_1, and A_2. Hence, statements like $\neg AB(M_1)$ appearing in a system description express *assumptions* regarding the state of components. Furthermore, the current observation OBS leads to the conclusion that all components work as expected.

As is the case in the above example, a diagnosis not necessarily needs to come to the conclusion that a component is faulty. Formally, Reiter's diagnosis is defined as follows.

Definition 5.1.7: *A* diagnosis *for* $(SD, COMP, OBS)$ *is a minimal set* $\Delta \subseteq COMP$ *such that*

$$SD \cup OBS \cup \{AB(c) \mid c \in \Delta\} \cup \{\neg AB(c) \mid c \in COMP \backslash \Delta\}$$

is consistent, i. e., has a satisfying structure.

In other words, a diagnosis determines whether a logical diagnosis model given by $SD \cup \{\neg AB(c_1), \ldots, \neg AB(c_n)\} \cup OBS$ is inconsistent and, more importantly, allows one to interpret why. The latter is achieved by retracting some of the assumptions made in SD regarding the abnormal-predicates, $AB(\cdot)$, such that consistency can be restored. Minimality in the above definition asserts that only those satisfying structures are diagnoses where the assumption that one or many components are abnormal, together with the assumption of *all* the other components behaving normally is consistent with the system description and the corresponding observations. Reiter refers to this as the *Principle of Parsimony for diagnosis*. This principle is an important prerequisite from a practical point of view, because retracting all of the abnormal-predicates in SD makes the logical model consistent in any case, and would yield an otherwise meaningless diagnosis.

From the above definition it can easily be derived that:

Proposition 5.1.1: *$\emptyset$ is a diagnosis (and the only diagnosis) for $(SD, COMP, OBS)$, if and only if*

$$SD \cup OBS \cup \{\neg AB(c) \mid c \in COMP\}$$

is consistent.

Proposition 5.1.1 gives a formal explanation for why, in the above example, the only obvious diagnosis is that no component is faulty; that is, if no failure was observed, the system must function normally.

Example (continued). Setting $o_1 \neq 26$ in the above example, leads to a diagnosis $\Delta = \{A_1\}$, i. e., $AB(A_1)$. This is also a minimal diagnosis, in that the assumption that this component is abnormal, together with the assumption of *all* the other components behaving correctly, is consistent with the system description and the observation.

Although in the previous example it is sufficient to assume a single component responsible for an observed symptom, there exist symptoms which are only explicable by assuming multiple abnormal components. Moreover, model-based diagnosis with *multiple faults* is in the worst case exponential in the number of components (cf. Bylander et al. [1991]); that is, every element of the power-set of the set of components may be a potential hypothesis for explaining observed inconsistencies. This was first formally (and independent of Reiter) acknowledged by de Kleer and Williams [1987].

Example (continued). Using the notation with signals, let, in the example from Fig. 5.2, $OBS = \{i_1 \mapsto 2, i_2 \mapsto 3, i_3 \mapsto 4, i_4 \mapsto 5, m_1 \mapsto 6, m_2 \mapsto 20, o_1 \mapsto 32, o_2 \mapsto 32\}$, i. e., o_1, and o_2 are not as expected. Then the possible diagnoses are $\{M_1, A_1, A_2\}$, $\{M_1, A_1, A_2\}$, and $\{A_1, A_2\}$. All these diagnoses can be used as hypotheses to make the system description consistent with the observations given by the set OBS.

Determination of diagnoses

This section details on how to formally determine a diagnosis for a given first-order system model $(SD, COMP, OBS)$. For brevity, only the two most widespread approaches from the literature of model-based diagnosis are discussed, the computation of minimal *conflict sets* according to Reiter [1987], and the algorithm of de Kleer and Williams [1987] for the *General Diagnostic Engine* (GDE). Other related approaches, but not necessarily bound to first-order reasoning, are briefly compared in §5.3.

The conflict set method. In what follows, a system and an according observation $(SD, COMP, OBS)$ are assumed. In order to determine useful diagnoses for the system, the concept of a *conflict set* is employed, which is formally defined as:

Definition 5.1.8: *A* conflict set *C is a set $\{c_1, \ldots, c_n\}$ with $c_i \in COMP$, such that*

$$SD \cup OBS \cup \{\neg AB(c_i) \mid c_i \in C\}$$

is inconsistent. C is minimal *if and only if there is no strict subset which is also a conflict set.*

Definition 5.1.7 of a diagnosis can now be reformulated in terms of Proposition 5.1.2.

Proposition 5.1.2: *$\Delta \subseteq COMP$ is a diagnosis, if and only if Δ is a minimal set such that $COMP \backslash \Delta$ is not a conflict set.*

Reiter uses the notion of a *hitting set* to determine conflicts.

Definition 5.1.9: *Let $\mathcal{C}$ be a set of sets, i. e., a collection. A* hitting set *for $\mathcal{C}$ is a set $H \subseteq \bigcup_{S \in \mathcal{C}} S$, such that $H \cap S \neq \emptyset$ for each $S \in \mathcal{C}$. A hitting set for $\mathcal{C}$ is* minimal, *if and only if no strict subset of it is a hitting set for $\mathcal{C}$.*

The central theorem to compute diagnoses is defined over the collection of conflict sets as follows.

Theorem 5.1.1: *$\Delta \subseteq COMP$ is a (minimal) diagnosis, if and only if Δ is a (minimal) hitting set for the collection of conflict sets.*

Proof:
For brevity, only the connection between diagnoses and hitting sets is shown, but not minimality. For a complete proof, see Reiter [1987].

($\Rightarrow$) By Proposition 5.1.2, $COMP \backslash \Delta$ is not a conflict set. It follows that every conflict set must contain an element from Δ, such that Δ is a hitting set for the collection of conflict sets.

($\Leftarrow$) Assume $COMP \backslash \Delta$ is not a conflict; if it was, then Δ would not hit it, and thus, contradict the fact that Δ is a hitting set. $\square$

From the validity of the above theorem, Reiter deduces an algorithm, in the following, referred to as Diagnose, which performs a *lattice* exploration over the possible diagnoses.

Definition 5.1.10: *A* lattice *is a non-empty partially ordered set* $(S, \subseteq)$, *such that for all elements* $a, b \in S$ *there exists an infimum, denoted* $inf(a, b)$, *and a supremum, denoted* $sup(a, b)$.

The diagnosis lattice which is associated with the example shown in Fig. 5.2, is depicted in Fig. 5.3. It contains 16 ($= 2^{|COMP|}$) elements.

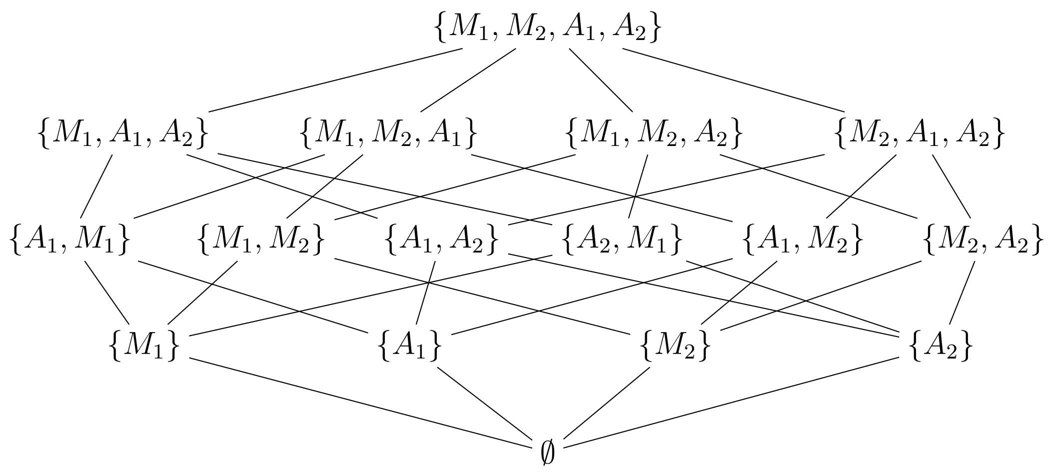


Fig. 5.3: Diagnostic lattice for the system depicted in Fig. 5.2.

Basically, the algorithm determines all minimal hitting sets for the set of conflict sets. Note that finding minimal hitting sets is also known as the *transversal problem*, which is one of the key problems in the combinatorics of finite sets (cf. Berge [1989] and Khachiyan et al. [2005]). Eiter and Gottlob [1995] give a good overview on the complexity of this and related problems, and, more precisely, identify the hitting set problem as described here to be $\mathcal{NP}$-complete.

Algorithm D (*Diagnose*). Let $\mathcal{C}$ be the collection or set of conflict sets. The algorithm Diagnose then performs a *breadth-first search* (cf. Knuth [1998]) on $\mathcal{C}$ starting with node $\emptyset$.

D1. [Next node.] Let C be the current node of the breadth-first search.

D2. [Conflict set?] Call a first-order theorem prover to determine whether $COMP \backslash C$ denotes a conflict set; if it is, continue with step D3, otherwise with step D4.

D3. [Eliminate non-diagnoses.] Eliminate all nodes C' from $\mathcal{C}$ for which it holds that $C' \cap (COMP \backslash C) = \emptyset$. C' cannot be a minimal diagnosis.

D4. [Eliminate direct descendants.] C is a minimal diagnosis. Eliminate all its descendants from $\mathcal{C}$. ∎

A discussion of the employed theorem prover is omitted at this point as it is beyond the scope of this thesis. First-order theorem provers, as required in this context, have been automated over the years, and a good discussion of their concepts is available, e. g., from Fitting [1996], Gabbay et al. [1994], or Harrison [TBA].

General Diagnostic Engine. de Kleer and Williams [1987] developed the *General Diagnostic Engine* (GDE) specifically for diagnosing systems which contain more than one fault simultaneously. Their method is similar to Reiter's, i. e., based on determination of minimal conflict and hitting sets, although the actual term "hitting set" is not used in their work.

To compute diagnoses, GDE exploits the formal correlation between hitting and conflict sets as given in Theorem 5.1.1; that is, it first determines the set of all conflict sets, of which the minimal hitting sets then constitute a system's diagnoses. Unlike Reiter, who explores the entire diagnosis lattice in a breadth-first manner, GDE uses an *inference engine*, and a so-called *assumption-based truth maintenance system* (ATMS) that outputs the minimal sets of faulty components. The inference engine can be chosen arbitrarily, as long as it finds the components that an observation depends on and can calculate values of variables in the system. The task of the ATMS is to record dependencies between observable parameters, i. e., sets of assumptions supporting the parameters. Again, for brevity the implementation details of both the inference engine and the ATMS (see de Kleer [1986], Reiter and de Kleer [1987]) are not discussed in detail. However, assumption sets as they appear in GDE correspond directly to Reiter's notion of minimal conflict sets (see Definition 5.1.8). If the value predicted for a parameter contradicts the observed value, at least one of the assumptions must be wrong, which is then, in the process of a stepwise simulation, updated in the ATMS.

The following algorithm, describes how GDE infers diagnoses, which in this structured form does not appear in the original work of de Kleer and Williams [1987].

Algorithm G (*GDE*). Let $\mathcal{C}$ be the collection of conflict or assumption sets for a system $(SD, COMP, OBS)$. The following then proceeds in a stepwise manner over all variables or parameters appearing in SD for which value assignments need to be determined.

G1. [Infer variable assignment.] Use the inference engine to determine a variable assignment for the current parameter (e. g., an input/output value of a component) based on the normal behaviour defined by SD.

G2. [Make assumption.] Tag the inferred value with assumptions from the system description (comprising AB-predicates over $COMP$) and add assumptions to the ATMS.

G3. [Compare values.] Compare the inferred value with the actually observed value from OBS.

G4. [Discrepancy?] If there is a discrepancy between the predicted and the actual value, use ATMS to retract previous assumptions in terms of negating AB-predicates, add affected components to $\mathcal{C}$, and update the ATMS.

G5. [Next value.] Repeat from step G1 until all values are determined.

G6. [Determine diagnoses.] Compute the minimal hitting sets for the collected sets in $\mathcal{C}$. ∎

There exist various optimisations and generalisations of GDE in the literature, e. g., as proposed by Struss and Heller [2001] in terms of the $G^{+}DE$ framework to add therapeutic measures into the reasoning process, or also by de Kleer and Williams [1992] who extended the algorithm by reflecting not only normal, but also faulty behaviour, resulting in the so-called *Sherlock* framework.

5.2 Diagnosis as a Boolean satisfiability problem

This section gives an efficient implementation of model-based diagnosis as introduced in §5.1. It is based on a translation of the diagnosis problem into a Boolean satisfiability problem, which is due to the use of monitors in the runtime reflection framework, determining whether monitored components are considered abnormal, or not. The proposed method aims at complementing the monitoring of systems, in that it can either confirm a monitor pointing to the root cause of a failure, or suggest a fault located outside the scope of a monitor or, possibly outside the scope of the entire system, then called an external fault (see §2).

5.2.1 From first-order to propositional diagnosis models

Although model-based diagnosis as described above is a useful tool for reasoning about systems, and various practical applications and implementations exist (cf. de Kleer and Williams [1987], Poole [1988], Baumgartner et al. [1996], Struss and Price [2003]), there are some limiting factors:

- Model-based diagnosis is laid out for the static component-based analysis of systems, where behavioural patterns over time are not considered.
- Despite the focus on static component-based analysis, the computational complexity is still high. The number of diagnoses can be exponential in the number of components, and diagnosis inference relies on first-order reasoning frameworks, such as theorem provers, which operate either interactively, or if automated, possibly do not terminate (cf. Fitting [1996], Harrison [TBA]).

Due to such reasons, model-based diagnosis as presented above, has been applied mostly to hardware systems, where components resemble physical parts (cf. Mikaelian et al. [2005]), rather than abstract entities such as communicating threads of a reactive system exhibiting complex behaviour over time. However, the runtime reflection framework analyses such behavioural patterns by using dedicated monitor components (see §4), such that failure and fault detection are handled by two separate entities. The connection of both are the results of the monitors, which are reflected by a special predicate as follows.

Definition 5.2.1: *Let* $(SD, COMP)$ *be a system description and* SIG *be a set of input and output signals. An* observation *for* $(SD, COMP)$ *is then defined by the set*

$$OBS = \{ok(s) \mid s \in SIG\} \mathbin{\dot\cup} \{\neg ok(s) \mid s \in SIG\},$$

where $\dot\cup$ *denotes the pairwise disjoint union of the subsets.*

Intuitively, the *ok*-predicate evaluates to *true* if an observed signal conforms with a specification, i. e., is “ok”. In the runtime reflection framework, given a signal $s \in SIG$, a monitor determines whether $ok(s)$ or $\neg ok(s)$ holds, since the stream of signals (or, actions) are evaluated by the monitoring layer. The *ok*-predicate can thus be considered as the counterpart for the *AB*-predicate, but applied to signals instead of components. It is assumed that, unlike components, signals cannot fail, but usually are the result of a faulty component, or some environmental influence outside the scope of the monitored system.

Rather than specifying the concrete behaviour of a component in terms of arithmetics and equations, it is abstracted from in this form by making use of the *ok*-predicates. The diagnosis problem imposed by this form of modelling is then referred to as *monitoring-based diagnosis problem.*

Definition 5.2.2: *A* (monitoring-based) diagnosis problem *is given by a tuple* $\mathcal{M} = (SD, COMP, OBS, SIG)$, *where*

- SD *is a system description consisting of a finite set of first-order sentences (with cardinality* $|COMP|$*), referred to as* abstract normality axioms, *which, for each* $c \in COMP$ *and* $I_c, O_c \subset SIG$, *are of the form*

$$\forall i \in I_c : ok(i) \land \neg AB(c) \Leftrightarrow \forall o \in O_c : ok(o)$$

- $COMP = \{c_1, \ldots, c_n\}$ *is a finite set of components,*
- $OBS = \{ok(s) \mid s \in SIG\} \mathbin{\dot\cup} \{\neg ok(s) \mid s \in SIG\}$ *is a finite set of observations, and*
- $SIG = \{s_1, \ldots, s_m\}$ *a finite set of signals used for communication inside the system.*

Note that the sets of variables I_c and O_c, denoting the input and output signals of a component c, are not always disjoint in order to preserve the system's causality. Moreover, this definition also caters for a more formal definition of some of the terms introduced informally in §2. For instance, a negatively evaluated *ok*-predicate corresponds to the notion of a symptom. Formally:

Definition 5.2.3: *For a monitoring-based diagnosis problem, given by the tuple* $\mathcal{M} = (SD, COMP, OBS, SIG)$*, let* $\{\neg ok(s) \mid s \in SIG\} \subseteq OBS$ *denote the set of* symptoms*. A system is* symptom-free*, if and only if the set of symptoms is empty, i. e., it holds that* $\forall s \in SIG : ok(s)$*.*

This implies a natural link between a symptom and the presence of a fault, which is exploited by model-based diagnosis and that follows as a consequence of the above definitions. Moreover, as is argued in §2, it is important to differentiate between two types of faults: internal and external. In the context of this chapter, let an externally induced fault be formally defined as follows.

Definition 5.2.4: *Let* $\mathcal{M} = (SD, COMP, OBS, SIG)$ *denote a monitoring-based diagnosis problem, for which an* external (or non-diagnosable) fault *can be deduced, if there exists a symptom but no responsible component, such that*

$$\exists s \in SIG\, \forall c \in COMP : \neg ok(s) \wedge \neg AB(c)$$

holds.

This definition intuitively asserts that, if a symptom is observed, and no component is abnormal, then its cause must be induced by the system's environment. External faults cannot be explained in terms of a faulty component, since they are beyond the scope of the diagnosis model. They are, therefore, also referred to as *non-diagnosable faults*. The other case, where a symptom is observed, and both an internal as well as a non-diagnosable fault are present, is not considered, since it trivially holds that for a symptom either an abnormal component exists, or not.

Remark. In Reiter's theory of diagnosis, the retraction of *all AB*-predicates constitutes a formal solution to the diagnosis problem, although not a minimal one for a given set of observations. With the substitution of "$\Rightarrow$" for "$\Leftrightarrow$" in the abstract normality axioms of Definition 5.2.2, this is no longer universally true. The motivation for this change is, foremost, a methodological one rather than a theoretical one; that is, instead of comparing isolated reference values with observed values as it is done in the original theory of model-based diagnosis, the monitors of the runtime reflection framework, essentially, interpret entire *series* of values over time to come to a verdict. It is therefore safe to assume that if the output values of a system component have not violated a specified property, then also the component's input values must have been according to that specification, and the component considered normal ($\Leftarrow$). In other words, in many settings, it can be thought unlikely that a faulty component would yield an entire *series* of correct output values, despite receiving a *series* of incorrect input values. This may be coincidentally observed for an isolated pair of input-output values, but most likely not if sufficiently long series of value pairs are considered. However, from a purely theoretical point of view, this assumption is not mandatory.

In the remainder, for abnormal system components, the symptom-fault-causality is thus given as follows.

Proposition 5.2.1: *Let $\mathcal{M} = (SD, COMP, OBS, SIG)$ be defined as usual, and Δ be a diagnosis for $\mathcal{M}$. If there exists not a non-diagnosable fault, then for a symptom $s \in SIG : \neg ok(s)$ the following holds*

$$\neg ok(s) \Rightarrow \exists c \in COMP : AB(c),$$

where $AB(c)$ denotes a system fault, i. e., component c is assumed abnormal, and $\{c\} \subseteq \Delta$.

Proof:
Follows as a direct consequence of the above definitions. □

What remains, when using this concept of a symptom, is a strictly causal model of the system under scrutiny which consists only of sentences with *ok* and *AB*-predicates. Monitors observe the *concrete behaviour* and indicate whether or not symptoms are currently present in the system under scrutiny.

Example. Reconsider the example system depicted in Fig. 5.2. The original system description, SD, can be reformulated using abstract normality axioms as follows:

$$SD = \left\{ \begin{array}{l} ok(i_1) \wedge ok(i_2) \wedge \neg AB(M_1) \Leftrightarrow ok(m_1), \\ ok(i_3) \wedge ok(i_4) \wedge \neg AB(M_2) \Leftrightarrow ok(m_2), \\ ok(m_1) \wedge ok(m_2) \wedge \neg AB(A_1) \Leftrightarrow ok(o_1), \\ ok(m_1) \wedge ok(m_2) \wedge \neg AB(A_2) \Leftrightarrow ok(o_2) \end{array} \right\}, \tag{5.1}$$

with the sets of component input and output signals being $I_{M_1} = \{i_1, i_2\}$, $O_{M_1} = \{m_1\}$, $I_{A_1} = \{m_1, m_2\}$, $O_{A_1} = \{o_1\}$, and so on. Notice that causal links between different components are preserved using this notation, for the sets of input and output variables are not always disjoint, e. g., $O_{M_1} \cap I_{A_1} = \{m_1\}$.

A monitoring-based diagnosis problem defines a first-order language, $\mathcal{L}_{FO}$, with a structure, $\mathcal{M} = (D, I)$ (see Definition 5.1.4), where $\mathcal{R}(\mathcal{L}_{FO}) = \{ok, AB\}$ with $ok^I = AB^I = \{(n) \mid n \in D\}$, and $D = \{i_1, i_2, i_3, i_4, m_1, m_2, o_1, o_2\}$. This strictly causal system model in the above sense, using only the *ok* and *AB*-predicates, is expressible in propositional logic by substitution of each predicate with a distinct, signed Boolean variable.

Propositional logic: some essential notation and elements

Let V be a finite set of *variables* with n elements. V gives rise to 2^n *literals* from $V \dot{\cup} \overline{V}$, where $\overline{V} = \{\neg v \mid v \in V\}$ is the set of *negations* of all the elements in V. Elements from V are the *positive literals*, and elements from $\overline{V}$ are the *negative literals*. Negation of literals is defined as follows:

$$\bar{l} = \begin{cases} \overline{v} & \text{if } l = v \in V, \text{ and} \\ v & \text{if } l = \overline{v} \in \overline{V}. \end{cases}$$

Both notations $\bar{l} \in \overline{V}$ as well as $\neg l \in \overline{V}$ will be used interchangeably throughout the remainder of this text. Moreover, variables will be usually denoted by the letters v, v', v'', whereas corresponding literals will be denoted by the letters l, l', l'', and so on.

A *clause* C with respect to a set V is a subset of pairwise distinct literals from $V \cup \overline{V}$. For brevity, the set of variables occurring in a clause C, given by $\{v \in V \mid v \in C \text{ or } \overline{v} \in C\}$, is also denoted by $vbl(C)$. The function vbl returning a set of distinct variables extends to sets of clauses in a natural manner.

The *valuation* or *assignment* of a Boolean variable is defined by a function $\alpha : V \to \mathbb{B}$, mapping truth values to variables in the expected manner. α extends to $V \cup \overline{V}$ by $\alpha(\overline{v}) = \neg\alpha(v)$ for $v \in V$. An assignment satisfies a clause C, if and only if $\alpha(l) = \mathit{true}$ for at least one literal $l \in (V \cup \overline{V}) \cap C$.

Reduction of SD

For the remainder of this chapter, a set of variables V is fixed, and let the tuple $\mathcal{M} = (SD, COMP, OBS, SIG)$ denote a monitoring-based diagnosis problem over V, defining the first-order language $\mathcal{L}_{FO}$. Further, let I be an interpretation, and $\mathcal{T} : I(\mathcal{R}(\mathcal{L}_{FO})) \to (V \cup \overline{V})$ be a partial translation function, which maps interpreted predicate symbols to literals as follows. Let $AB^I, ok^I \in \mathcal{R}(\mathcal{L}_{FO})$, and P_{SD} be the set of predicates as they are used in SD, then

$$\mathcal{T}(AB(c \in COMP)) = \begin{cases} c & \text{if } AB(c) \in P_{SD} \\ \overline{c} & \text{if } \neg AB(c) \in P_{SD}, \end{cases}$$

and

$$\mathcal{T}(ok(s \in SIG)) = \begin{cases} s & \text{if } ok(s) \in P_{SD} \\ \overline{s} & \text{if } \neg ok(s) \in P_{SD}, \end{cases}$$

with $COMP, SIG \subseteq V$, and $COMP \cap SIG = \emptyset$. As before (see, e. g., Eq. 5.1), SIG denotes the set of observable signals (or, actions) for the system. For every ok-predicate in SD, $\mathcal{T}$ yields a signed variable from SIG, and for every interpreted AB-predicate in SD, a signed variable from $COMP$, thus, mapping the first-order diagnosis model to a strictly propositional one.

Applying $\mathcal{T}$ to the system description given by Eq. 5.1, the following propositional system description $SD_{\mathcal{T}}$ is obtained, with $SIG = \{i_1, i_2, i_3, i_4, m_1, m_2, o_1, o_2\}$ and $COMP = \{A_1, A_2, M_1, M_2\}$, and all previous predicate symbols substituted accordingly:

$$SD_{\mathcal{T}} = \left\{ \begin{array}{l} i_1 \wedge i_2 \wedge \overline{M}_1 \Leftrightarrow m_1, \\ i_3 \wedge i_4 \wedge \overline{M}_2 \Leftrightarrow m_2, \\ m_1 \wedge m_2 \wedge \overline{A}_1 \Leftrightarrow o_1, \\ m_1 \wedge m_2 \wedge \overline{A}_2 \Leftrightarrow o_2 \end{array} \right\}. \tag{5.2}$$

Instead of specifying the concrete behaviour of the system as is done by Reiter, now axioms of the form

$$i_1 \wedge i_2 \wedge \overline{M}_1 \Leftrightarrow m_1$$

constitute the system description; in this case, denoting an observation m_1 which is due, given component M_1 is not faulty, and i_1, i_2 are ok (and vice versa). A positive literal $s \in SIG$ encodes conforming, and a negative literal, $s \in \overline{SIG}$, non-conforming system behaviour. This is symmetrical for literals denoting system components. A set of system observations for $SD_{\mathcal{T}}$ is thus given as $OBS_{\mathcal{T}} \subseteq SIG \,\dot{\cup}\, \overline{SIG}$.

The formal problem of diagnosis as given in Definition 5.1.7 transfers over to the propositional setup in a straightforward manner:

Definition 5.2.5: *Let $(SD_{\mathcal{T}}, COMP, OBS_{\mathcal{T}})$ be a propositional system model with observations. A diagnosis $\Delta \subseteq COMP$ in the propositional domain is a minimal set, such that $SD_{\mathcal{T}} \cup OBS_{\mathcal{T}} \cup \Delta \cup \overline{COMP} \backslash \Delta$ is consistent.*

The notion of a monitoring-based diagnosis problem (see Definition 5.2.2) transfers over in a similar manner, but is, for brevity, not elaborated on in detail as it is self-explanatory.

Example. Let us now reconsider the system depicted in Fig. 5.2 in the propositional domain, with $SD_{\mathcal{T}}$ defined by Eq. 5.2, where $COMP = \{A_1, A_2, M_1, M_2\}$, and $OBS_{\mathcal{T}} = \{i_1, i_2, i_3, i_4, \overline{m}_1, m_2, \overline{o}_1, \overline{o}_2\}$, where $\overline{m}_1$, $\overline{o}_1$, and $\overline{o}_2$ denote observations that are not ok. Finding a consistent assignment of variables for $SD_{\mathcal{T}} \cup OBS_{\mathcal{T}} \cup \Delta \cup \overline{COMP} \backslash \Delta$ in terms of elements from $COMP$, leaves the possible diagnoses $\{M_1, A_1, A_2\}$, $\{M_1, A_1\}$, $\{M_1, A_2\}$, and $\{M_1\}$.

How to formally, as well as efficiently, determine such a result for less obvious propositional diagnosis problems, is the subject of the next section.

5.2.2 Computing diagnoses using Boolean satisfiability

This section outlines a first algorithm for formally solving the diagnosis problem in the propositional domain. The algorithm is based upon the well-known DPLL-algorithm for checking the satisfiability of Boolean formulae in normal form. The DPLL-algorithm needs to have its input in a normal form.

Clause normal form

The diagnosis algorithm processes clauses which are in the *clause normal form*, sometimes also referred to as *conjunctive normal form* (CNF), since it has to be the conjunction of disjunctions of literals, i. e., of the form

$$\bigwedge_{i=1}^{m}(\bigvee_{j=i}^{k} l_{i,j}) = (l_{1,1} \vee \ldots \vee l_{1,k}) \wedge \ldots \wedge (l_{m,1} \vee \ldots \vee l_{m,k}).$$

For every Boolean formula F there exists an equivalent formula in CNF, denoted for brevity by $CNF(F)$, which can be computed in polynomial time in the size of F, when certain prerequisites are fulfilled, such as allowing the introduction of new variables; standard algorithms apply (cf. Nonnengart and Weidenbach [2001]). Let $A = \{\alpha_1, \ldots, \alpha_n\}$ be a set of assignments, then A is a satisfying assignment for a formula $F' = CNF(F)$, denoted as $A \models F'$, if and only if it satisfies all clauses in F'. For example, the CNF-representation of the previous $SD_{\mathcal{T}}$ in Eq. 5.2 is given as

$$\begin{aligned} CNF(SD_{\mathcal{T}}) = \quad & (\bar{i}_1 \vee \bar{i}_2 \vee M_1 \vee m_1) \wedge (\overline{m}_1 \vee i_1) \wedge (\overline{m}_1 \vee i_2) \wedge (\overline{m}_1 \vee \overline{M}_1) \wedge \\ & (\bar{i}_3 \vee \bar{i}_4 \vee M_2 \vee m_2) \wedge (\overline{m}_2 \vee i_3) \wedge (\overline{m}_2 \vee i_4) \wedge (\overline{m}_2 \vee \overline{M}_2) \wedge \\ & (\overline{m}_1 \vee \overline{m}_2 \vee A_1 \vee o_1) \wedge (\overline{o}_1 \vee m_1) \wedge (\overline{o}_1 \vee m_2) \wedge (\overline{o}_1 \vee \overline{A}_1) \wedge \\ & (\overline{m}_1 \vee \overline{m}_2 \vee A_2 \vee o_2) \wedge (\overline{o}_2 \vee m_1) \wedge (\overline{o}_2 \vee m_2) \wedge (\overline{o}_2 \vee \overline{A}_2). \end{aligned}$$

For brevity, the $\wedge$-symbols will be omitted in future references to CNF-formulae, and a comma or newline-separated list of clauses used instead, where each clause is, in turn, a comma-separated set of literals. It formally holds that a set of assignments A satisfies $CNF(SD_{\mathcal{T}})$, if and only if $A \models SD_{\mathcal{T}}$.

Boolean satisfiability

In the literature, checking satisfiability of Boolean formulae in CNF is known as *SAT-solving*. Consequently, a SAT-solver is a program, which takes a (Boolean) formula in CNF, F, and determines whether an assignment exists, such that F is satisfiable. The main algorithm used for this purpose was first recursively described by Davis and Putnam [1960], and revised by Davis et al. [1962] (in short, DPLL to acknowledge the original authors) as follows.

Algorithm S (*Satisfiability of a Boolean formula*). Let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be a set of clauses over a set of Boolean variables V, and A an initially empty set of variable assignments for the elements $vbl(\mathcal{C})$.

S1. [A valid for $\mathcal{C}$?] If the clauses in $\mathcal{C}$ evaluate to *true* using A, return *true* and terminate.

S2. [A invalid for $\mathcal{C}$?] If the clauses in $\mathcal{C}$ evaluate to *false* using A, return *false* and terminate.

S3. [Positive unit clause in $\mathcal{C}$?] If there exists a positive unit clause $C \in \mathcal{C}$, such that $C = \{l \in V\}$, set $A := A \cup \{l \mapsto 1\}$, and invoke Algorithm S recursively again.

S4. [Negative unit clause in $\mathcal{C}$?] If there exists a negative unit clause $C \in \mathcal{C}$, such that $C = \{l \in \overline{V}\}$, set $A := A \cup \{l \mapsto 0\}$, and invoke Algorithm S recursively again.

S5. [Positive pure literal in $\mathcal{C}$?] If there exists a positive literal $l \in V$ which only appears exactly once in all of $\mathcal{C}$, set $A := A \cup \{l \mapsto 1\}$, and invoke Algorithm S recursively again.

S6. [Negative pure literal in $\mathcal{C}$?] If there exists a negative literal $l \in \overline{V}$ which only appears exactly once in all of $\mathcal{C}$, set $A := A \cup \{l \mapsto 0\}$, and invoke Algorithm S recursively again.

S7. [Otherwise.] Let $v \in V$ be an unassigned variable that appears in $\mathcal{C}$, but not yet in A. Then there are two cases:

1. If Algorithm S applied recursively to $\mathcal{C}$ and $A \cup \{v \mapsto 0\}$ returns true, return *true* and terminate.
2. Otherwise, invoke Algorithm S with $\mathcal{C}$ and $A \cup \{v \mapsto 1\}$, trying the other assignment for v. ∎

When calling the algorithm recursively in one of the above steps, the individually gathered assignments in A are stored in a special data structure, the *runtime stack*. For brevity, the runtime stack is not part of or reflected in the algorithm. Moreover, since the DPLL-procedure tries out, in the worst case, all the possible variable assignments appearing in a given set of clauses, it is also known as a *backtracking algorithm* that "unrolls" the information stored on the stack, whenever an assignment is encountered that does not lead to a satisfying overall solution.

The DPLL-algorithm provides a solution to the *3-SAT* or k-SAT problem if feasible, where clauses are considered that contain 3 or $k > 3$ literals with positive or negative occurrence. The k-SAT problem can be mapped to a 3-SAT problem, and since Cook [1971] it is well-known that k-SAT is $\mathcal{NP}$-complete. For subclasses, such as *2-SAT* where only clauses with at most two literals are considered or Horn clauses, the SAT-problem is in the complexity class $\mathcal{P}$; that is, there exist deterministic algorithms for solving it in polynomial time with respect to the number of variables (cf. Gallo and Scutella [1988]).

The correspondence between diagnosis and Boolean satisfiability

In the following, a first algorithm is presented for solving the diagnosis problem, as given by Definition 5.2.5, based on satisfiability checking of Boolean formulae in CNF. For this purpose, a variant of the DPLL-algorithm is used, which not only determines satisfiability, but also the set and number of *all* satisfying assignments for a set of clauses. This extension is essential, since observations for a system may be explained by more than one diagnosis (see, e. g., the above examples). In other words, considering the SAT-problem merely in terms of a Boolean decision problem is insufficient for this purpose. For diagnosis, all solutions may be relevant.

The complexity class $\#\mathcal{P}$. Counting the solutions for decision problems is not only interesting in face of model-based diagnosis. Various other applications can be found in mathematics (cf. Vadhan [2001]) and, for instance, in automated reasoning (cf. Roth [1996]). In a seminal paper, Valiant [1979] has introduced, especially for the purpose of counting the number of solutions for decision problems, the complexity class $\#\mathcal{P}$, $\#\mathcal{P}$-complete, and $\#\mathcal{P}$-hard. Formally, a problem is in $\#\mathcal{P}$, if there is a nondeterministic, polynomial-time Turing machine that, for each instance of the problem, has a number of accepting computations, equal to the number of solutions. Naturally, a $\#\mathcal{P}$ problem is at least as hard as the corresponding $\mathcal{NP}$ problem: once the number of solutions has been determined, it can be compared to zero to solve the $\mathcal{NP}$ decision problem. It follows that every $\#\mathcal{P}$ problem, corresponding to an $\mathcal{NP}$-complete problem, is $\mathcal{NP}$-hard. A problem is $\#\mathcal{P}$-complete, if and only if it is in $\#\mathcal{P}$, and every other problem in $\#\mathcal{P}$ can be reduced to it in polynomial time. With respect to $\#\mathcal{P}$, the problem of determining all satisfying clauses to a Boolean formula in CNF is referred to as #SAT. #SAT in turn is known to be $\#\mathcal{P}$-complete (cf. Papadimitriou [1994]).

One possible, but naive way for solving the #SAT (as well as the propositional diagnosis) problem is captured by the following Algorithm #S, which relies upon a SAT-solver (or, Algorithm S) for solving instances of the SAT-decision problem. In the following, without loss of generality, it is assumed that a solver based on Algorithm S not only returns *true* or *false*, but also after termination the set A of satisfying assignments, if any. DPLL-based solvers such as Chaff (Moskewicz et al. [2001]) are able to do this.

Algorithm #S (*Iterative DPLL for #SAT*). Let $\mathcal{C}$ be a set of clauses over a set of Boolean variables V, and $\mathcal{A}$ an initially empty collection of variable assignments for the elements $vbl(\mathcal{C}) \subseteq V$.

S1. [Determine set of assignments.] Use Algorithm S to obtain a set of satisfying assignments for $\mathcal{C}$, $A = \{\alpha(l_1), \ldots, \alpha(l_n) \mid l_i \in vbl(\mathcal{C})\}$.

S2. [Assignment set not empty?] If $A \neq \emptyset$, go to step S4.

S3. [Terminate.] If no further assignments could be determined, i. e., $A = \emptyset$, $\mathcal{A}$ must be the output and two cases differentiated:

1. If $\mathcal{A} \neq \emptyset$, return *true*.
2. Otherwise, return *false*.

S4. [Add assignment to collection and continue.] Set $\mathcal{A} := \mathcal{A} \cup \{\alpha(l_1), \ldots, \alpha(l_n)\}$ and $\mathcal{C} := \mathcal{C} \cup \{\bar{l}_1, \ldots, \bar{l}_n\}$, and go back to step S1. ∎

Algorithm #S constructs a collection $\mathcal{A}$ of satisfying assignments for $\mathcal{C}$, and the total number of assignments is then given by $|\mathcal{A}|$. The algorithm obviously terminates since in each step the negative set of assignments is added to the set of clauses $\mathcal{C}$ to disregard it as a possible future solution. When no more assignments can be found, step S3 applies and the algorithm returns, besides $\mathcal{A}$, depending on whether $|\mathcal{A}| > 0$, either *true* or *false*.

The following example illustrates how Algorithm #S can be used in a straightforward manner to compute diagnoses.

Example (continued). Let $\mathcal{C}$ be a set of clauses in CNF whose elements are given as follows:

$$\mathcal{C} = \left\{ \begin{array}{l} \{\bar{i}_1, \bar{i}_2, M_1, m_1\}, \{\overline{m}_1, i_1\}, \{\overline{m}_1, i_2\}, \{\overline{m}_1, \overline{M}_1\}, \\ \{\bar{i}_3, \bar{i}_4, M_2, m_2\}, \{\overline{m}_2, i_3\}, \{\overline{m}_2, i_4\}, \{\overline{m}_2, \overline{M}_2\}, \\ \{\overline{m}_1, \overline{m}_2, A_1, o_1\}, \{\overline{o}_1, m_1\}, \{\overline{o}_1, m_2\}, \{\overline{o}_1, \overline{A}_1\}, \\ \{\overline{m}_1, \overline{m}_2, A_1, o_2\}, \{\overline{o}_2, m_1\}, \{\overline{o}_2, m_2\}, \{\overline{o}_2, \overline{A}_2\}, \\ \{i_1\}, \{i_2\}, \{i_3\}, \{i_4\}, \{\overline{m}_1\}, \{m_2\}, \{\overline{o}_1\}, \{\overline{o}_2\} \end{array} \right\}. \tag{5.3}$$

The clauses contained in $\mathcal{C}$ resemble the system description and accompanying observations as used in the previous example. The first clauses encode the system description, whereas the singletons correspond to the accompanying observations. Application of Algorithm #S yields a set $\mathcal{A}$ with the following satisfying assignments:

$$\mathcal{A} = \left\{ \begin{array}{l} \{i_1, i_2, i_3, i_4, \overline{m}_1, m_2, \overline{o}_1, \overline{o}_2, M_1, \overline{M}_2, A_1, A_2\}, \\ \{i_1, i_2, i_3, i_4, \overline{m}_1, m_2, \overline{o}_1, \overline{o}_2, M_1, \overline{M}_2, A_1, \overline{A}_2\}, \\ \{i_1, i_2, i_3, i_4, \overline{m}_1, m_2, \overline{o}_1, \overline{o}_2, M_1, \overline{M}_2, \overline{A}_1, A_2\}, \\ \{i_1, i_2, i_3, i_4, \overline{m}_1, m_2, \overline{o}_1, \overline{o}_2, M_1, \overline{M}_2, \overline{A}_1, \overline{A}_1\} \end{array} \right\}.$$

That is, the solution to the #SAT problem given by $\mathcal{C}$ is determined by $|\mathcal{A}|$, whereas the elements in $\mathcal{A}$ determine, at the same time, the diagnoses for the previous example. In particular, the diagnoses are given by the following subsets from $\mathcal{A}$: $\{M_1, A_1, A_2\}$, $\{M_1, A_1\}$, $\{M_1, A_2\}$, $\{M_1\}$.

Efficiency considerations

Although feasible, this approach to propositional diagnosis is not very efficient. Writing a program that implements the above means having to traverse iteratively an exponentially large search space. Even for small propositional diagnosis problems involving no more than 30 variables, this brute-force method needs to try, in the worst case, 2^{30} cases each time (not regarding the observed symptoms, if used for diagnosis). Usually, for modern solvers, such as Chaff, such instances pose no great challenge. However, when implementing Algorithm #S by using efficient standard solvers like Chaff, another purely practical problem arises: the implementation is likely to be runtime inefficient, due to the time required for iteratively starting, ending, and re-starting the external SAT-solver. Often mechanical parts are involved, such as the moving of the hard-disk heads in I/O operations, necessary to exchange data between the processes, or to swap memory for solving large instances of individual SAT-problems involving hundreds or thousands of variables.

How to efficiently realise the above method, and how to circumvent the sketched efficiency problems, are thus the subjects of the coming sections.

5.2.3 Optimisation and determination of all minimal diagnoses

The algorithm described in this section caters for both the *computationally efficient* determination of the total number of solutions for a given SAT-problem, i. e., an answer to the #SAT-problem, as well as the determination of the set of all satisfying assignments, which in the context of this chapter resemble system diagnoses. However, in diagnosis often not all the solutions are of interest, but only those sufficiently meaningful to explain a failure (i. e., Principle of Parsimony). The following algorithm caters for that, in that it determines diagnoses where the cardinality of those literals corresponding to $\neg AB(\cdot)$ is at most $n \in \mathbb{N}^{\geq 0}$, where n can be chosen by the user of that system. This is also referred to as the *n-fault assumption* (cf. Baumgartner et al. [1996]). As this section will show, the n-fault assumption constitutes an effective pruning criterion for the overall problem search space and yields fast execution times for the algorithm even for very large satisfiability or diagnostic problems. It will be reflected in the algorithm by counting in each set of satisfying variable assignments, those positive variable assignments, where the variables correspond to individual components in the diagnostic model.

The algorithm will be referred to as LSAT, short for *light SAT-solver*, and has been described in the context of model-based diagnosis by Bauer [2005]. Like other algorithms for testing satisfiability of Boolean or quantified Boolean formulae (cf. Samulowitz and Bacchus [2005]), LSAT consists of two main procedures, `branch` and `refute`. The purpose of `branch` is to open a new "branch" of possible assignments for the input clauses by selecting a previously unassigned variable. `refute` determines which clauses are satisfiable by either the positive interpretation or the negative interpretation of that variable. The satisfying assignments are stored, and conflicting ones resolved using backtracking.

In what follows, let V denote a set of Boolean variables, and without loss of generality, $\mathcal{C} = CNF((SD, COMP, OBS))$ a propositional system model over V in CNF.[2] Each element in $\mathcal{C}$ is a tuple (C, pos, neg) containing a clause $C = \{l_1, \ldots, l_n \mid l_i \in V \dot{\cup} \overline{V}\}$, and two entries $pos, neg \in \mathbb{N}^{\geq 0}$, where pos is equal to the number of positive literals in C, and neg equal to the number of negative literals in C, such that $|C| = pos + neg$.

[2] Any other set of clauses in CNF is possible, but then, naturally, the n-fault assumption has no effect on the efficiency of the algorithm.

Given a set $\mathcal{C}$ defined as above, the procedure implementing `branch` is then defined as follows.

Algorithm B (*Branch*). Let $\mathcal{A} = \{A_1, \dots, A_n\}$ be an initially empty collection of satisfying variable assignments for the clauses stored in $\mathcal{C}$. Moreover, any variable given by $vbl(\mathcal{C})$ can appear marked or unmarked, indicating whether or not a truth assignment for the variable was previously made.

B1. [Find next unmarked variable.] Look for any unmarked variable $v \in vbl(\mathcal{C})$. If an unmarked v could be found, go to step B2. Otherwise, go to step B3.

B2. [Branch.] Branch off by invoking Algorithm R (`refute`) twice: the first time, using v, the second time, using $\overline{v}$. Algorithm R and Algorithm B use a common set A to collect satisfying variable assignments. A variable assignment is added to A by Algorithm R, if it is not a conflicting assignment (see the description of Algorithm R below).

B3. [Add assignment set to $\mathcal{A}$.] If no unmarked v could be found, i. e., there are no further variable assignments to be made, then set $\mathcal{A} := \mathcal{A} \cup A$, and terminate. A may also be the empty set. ∎

Algorithm R which is invoked by the relatively short Algorithm B is described next. The algorithm controls when variables are marked or unmarked, and, basically, implements the `refute` procedure as outlined above. It uses a projection function $\Pi : 2^{\mathcal{C}} \times (V \cup \overline{V}) \rightarrow 2^{\mathcal{C}}$; if called with an argument $\overline{v} \in \overline{V}$, those clauses from $\mathcal{C}$ are returned in which v occurs negatively, respectively positively if called with $v \in V$.

Algorithm R (*Refute*). Let V, $\mathcal{C}$, and $\mathcal{A}$ be defined as in Algorithm B. Moreover, both algorithms operate on common or shared sets of variables, clauses, and satisfying variable assignments. (In terms of an actual implementation, such as given by LSAT, this is realised via global heap variables.) Algorithm R is then invoked with a parameter $l \in V \cup \overline{V}$, defining a previously unassigned variable with either a positive sign, if $l \equiv v \in V$, or negative sign, if $l \equiv \overline{v} \in \overline{V}$.

R1. [Initialise.] Let $\mathcal{C}_{curr} := \Pi(\mathcal{C}, \bar{l})$, and $\mathcal{C}_{true} := \Pi(\mathcal{C}, l)$. The literal l constitutes a satisfying assignment for all the elements in $\mathcal{C}_{true}$, whereas the elements from $\mathcal{C}_{curr}$ need to be examined further.

R2. [Decrement counter.] If $l \in \overline{V}$, then decrement for each element $(C, pos, neg) \in \mathcal{C}_{curr}$, pos by one. Otherwise, decrement neg by one.

R3. [Push assignment.] Set $A := A \cup \{l\}$, i. e., add l to the set of satisfying assignments.

R4. [Conflict?] Whether or not further variable assignments are required depends on two conditions:

1. $\mathcal{C}_{curr}$ does *not* contain an "empty" tuple, i. e., a tuple of the form $(C, pos = 0, neg = 0)$, indicating that $\bar{l}$ is a conflicting assignment for C, and

2. $|A'| \leq n$, where $A' = \{l \mid l \in COMP\} \subseteq A$, and $n \in \mathbb{N}$ is the maximum number of faults as defined by the n-fault assumption.

If both conditions hold, continue with R5. Otherwise, continue with step R8.

R5. [Mark and subsume clauses.] Mark l (as being evaluated), and set $\mathcal{C} := \mathcal{C} \backslash \mathcal{C}_{true}$.

R6. [Branch and expand.] Invoke Algorithm B again for a stepwise expansion of the overall search space.

R7. [Unmark and resume clauses.] Unmark l, and set $\mathcal{C} := \mathcal{C} \cup \mathcal{C}_{true}$.

R8. [Increment counter.] If $l \in \overline{V}$, then increment for each element $(C, pos, neg) \in \mathcal{C}_{curr}$, pos by one. Otherwise, increment neg by one.

R9. [Pop assignment.] Set $A := A \backslash \{l\}$. ▮

Together, Algorithm B and Algorithm R form the foundation for LSAT's implementation. If applied to a set of clauses defined as above, LSAT performs the following, based on the n-fault assumption: If the user chooses to ignore n, i. e., $n = \infty$, LSAT determines all possible satisfying assignments for $\mathcal{C}$, thus also an answer to the #SAT problem. However, if in a diagnosis problem, $n \leq |COMP|$, only those satisfying assignments are determined which contain at most n positive literals corresponding to components. In other words, LSAT gives explanations for an observed failure which are based on at most n faulty components. In consequence, if $n < \infty$, LSAT may no longer be able to compute an answer to the #SAT problem, but it does filter out practically useless diagnoses, such as those where all components are assumed faulty.

LSAT's computation is space efficient, but naturally, has the same worst-case time complexity as any other #SAT algorithm known from the literature. Space efficiency is due to the use of "shared" data structures representing clauses and variables, i. e., invocations of Algorithm B and Algorithm R simply mark or unmark on the same data set, variables and clauses to denote whether or not these should be evaluated further. This yields a linear space requirement with respect to the size of the input formula in CNF. As with Algorithm #S, in the worst case, still an exponential number of steps with respect to the number of variables is required before the algorithm has determined all satisfying assignments. However, the n-fault assumption provides a heuristic on how to keep the number of steps minimal, and the diagnoses meaningful (see also §6 and example on p. 128).

Note that in order to keep the presentation of the algorithm for LSAT comprehensible and the following analysis of it simple, optimisations such as determining *unit clauses* or *pure literals*, suggested in the original DPLL-algorithm, have not been considered. However, §6 shows how such optimisations can be added to achieve additional runtime performance and with little extra implementation effort.

Analysis

Semantic trees, as originally introduced by Kowalski and Hayes [1969], form a suitable foundation for an analysis of LSAT, since there exists a direct correspondence between LSAT's algorithmic steps and the stepwise creation of a semantic tree: Let T be a semantic tree for a set of clauses $\mathcal{C}$ with $vbl(\mathcal{C}) \subseteq V$. Let $v \in vbl(\mathcal{C})$, then $v \in T \Leftrightarrow v \in A$, where A is the set of satisfying assignments for $\mathcal{C}$ used by LSAT. The individual branches of T are formally defined as follows.

Definition 5.2.6: *Let V be a set of Boolean variables.* A branch *(of a semantic tree), M, is defined by a partial function $\sigma : V \rightarrow \{\top, \bot\}$, where*

$$\sigma(v) = \begin{cases} \top & \textit{if } v \in M, \\ \bot & \textit{if } \overline{v} \in M. \end{cases}$$

For a variable $v \in V$, if neither $v \in M$ nor $\overline{v} \in M$, $\sigma(v)$ is undefined. In the case of LSAT, this corresponds to a variable which has not yet been added to the set A throughout its computation.

Definition 5.2.7: *Formally,* a semantic tree *T is a labelled, unordered* binary tree *built of branches, whose nodes are sets of clauses from $\mathcal{C}$. For T, the following holds:*

- *Let $v \in M$, then for two edges in M who share the same parent node, the labels are $\sigma(v) = \top$ and $\sigma(v) = \bot$, respectively, if and only if v does not appear elsewhere on M.*
- *Each leaf node of a branch may be labelled with a clause $(\{l_1, \ldots, l_m\}, pos, neg) \in \mathcal{C}$, if for every $l_i : \overline{l}_i \in M$.*

Definition 5.2.8: *A branch M of a semantic tree is* closed*, if its literals contradict a clause $(C, pos, neg) \in \mathcal{C}$, such that $\forall l \in C : \overline{l} \in M$.*

In other words, a closed branch is marked with the clause that is violated by the literals stored in that branch (see Definition 5.2.7).

Proposition 5.2.2: *A branch M which is not closed and cannot be expanded any further, i. e., $\forall v \in vbl(\mathcal{C}) : \sigma(v) \in \{\top, \bot\}$, constitutes a satisfying assignment for $\mathcal{C}$.*

A closed branch M is given in LSAT, if its leaf contains an empty clause, such that for a clause $C : pos + neg = 0$. The empty clause corresponds to a clause, whose literals appear complementary on M (see step R4).

Definition 5.2.9: *A semantic tree is* closed*, if and only if all its branches are closed.*

Proposition 5.2.3 (Termination): LSAT *always terminates (given enough run-time).*

Proof:
Upon each step in the algorithm that expands the search tree by an additional level, LSAT performs two things:

1. It decrements the *pos*, respectively *neg* counters for each clause where a previously unassigned literal l appears either positively or negatively, not satisfying the clause (steps R1 to R3).
2. It adds all clauses satisfied by l to the set $\mathcal{C}_{true}$ of satisfied clauses (and deactivates those, such that they are not reconsidered in future steps (steps R1 and R5)).

This expansion of the tree is bound to terminate, since there is only a finite number of literals available for selection, i. e., $|vbl(\mathcal{C})|$, and all clauses contain finitely many positive or negative occurrences of literals, i. e., for each $(C, pos, neg) : pos + neg \leq |vbl(\mathcal{C})|$, and in each step either $pos = pos - 1$ or $neg = neg - 1$. Thus, expansion of a branch in the semantic tree is interrupted upon any of the following three criteria each of which are checked in step R4:

1. A branch is closed, i. e., for a clause $C : pos + neg = 0$.
2. A branch is fully expanded and not closed, i. e., no more unassigned literals can be found, and $\mathcal{C}_{true} = \mathcal{C}$.
3. A branch contains more than $n \in \mathbb{N}^{\geq 0}$ positively assigned literals that correspond to abnormal components in the diagnosis model (and violate the n-fault assumption).

By Proposition 5.2.2, if the branch is fully expanded and not closed, then it contains a satisfying assignment for the set of input clauses. If it is closed, or violates the n-fault assumption, LSAT backtracks by incrementing the clause counters accordingly (steps R7 to R9).

LSAT terminates expansion when all branches are either closed, violate the n-fault assumption, or are fully expanded, which results in a semantic tree with worst-case $2^{|vbl(\mathcal{C})|}$ nodes, if all assignments have to be made before a fully expanded branch exists, or all branches are closed. □

Proposition 5.2.4 (Soundness): *If there is a semantic tree for $\mathcal{C}$, which is closed, then $\mathcal{C}$ is unsatisfiable.*

Proof:
By contradiction: Assume there exists a closed semantic tree for a satisfiable $\mathcal{C}$. Then, by definition of semantic trees, there exists for $\mathcal{C}$ a branch in its tree, which is fully expanded and not closed. However, since the tree of $\mathcal{C}$ is closed, all of its branches must also be closed. The existence of a fully expanded branch which is not closed is thus a contradiction to the initial assumption. □

Proposition 5.2.5 (Completeness): *Let $\mathcal{C}$ be unsatisfiable. Then there exists a closed semantic tree for $\mathcal{C}$.*

Proof:
By contradiction: Let T be the fully expanded semantic tree for $\mathcal{C}$. Assume there exists a branch in T, M, that is not closed. By Proposition 5.2.2, if M is fully expanded and not closed, it contains a satisfying assignment for $\mathcal{C}$. However, since $\mathcal{C}$ is not satisfiable, there cannot be a satisfying assignment for it, thus, contradicting the assumption. □

The following example illustrates how LSAT works when using a 2-fault assumption.

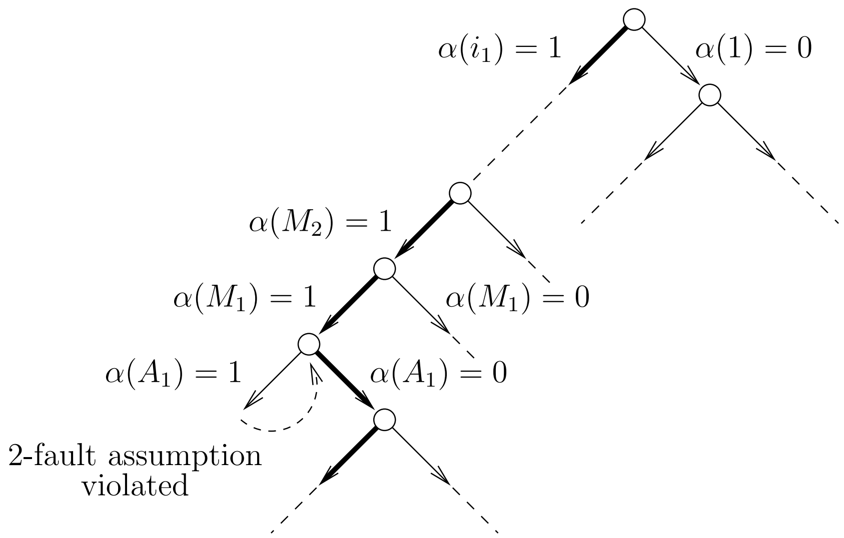


Fig. 5.4: Evaluation of a clause set using a 2-fault assumption. The bold path marks a series of non-conflicting assignments.

Example (LSAT using the 2-fault assumption). Let $\mathcal{C}$ be a set of clauses in CNF defined as in Eq. 5.3, reflecting a system model and observations. Fig. 5.4 shows LSAT traversing the search space using a 2-fault assumption. Following the left-most path in the tree, LSAT makes two variable assignments, $\alpha(M_2) = 1$ and $\alpha(M_1) = 1$, each of which corresponds to an abnormal component in the system model. (To make the illustration more readable, the Boolean values 1 and 0 have been chosen, instead of *true* and *false*.) Thus, the possible assignment $\alpha(A_1) = 1$ on that same branch violates the 2-fault assumption. In consequence, LSAT performs backtracking and continues with the right-hand-side. As such, the general n-fault assumption constitutes an effective pruning criterion, when traversing the semantic tree of an input formula. In this example, the branch is closed (and the underlying search space disregarded) *before* it is fully expanded.

An efficient implementation of the above algorithm, the custom data structures used by LSAT, as well as performance benchmarks are outlined in greater detail in §6.

5.3 Related work

This section briefly outlines other approaches to diagnostic problem solving, i. e., related work. In particular, contributions to three popular diagnosis approaches are discussed for comparison; that is, *non-monotonic reasoning* for diagnosis, *diagnosis of discrete-event systems*, and the so-called *Fault Detection and Isolation* approach to diagnosis, which has its roots in the more traditional engineering disciplines, such as mechanical and electrical engineering, rather than in computational logic and computing sciences.

Non-monotonic reasoning for diagnosis

Besides the first-order inference-based diagnosis methods, there exist various works which consider the problem in the same logical domain, but employ a totally different concept of reasoning. These approaches use so-called *non-monotonic and abductive reasoning* to solve the diagnostic problem.

Unlike in consistency-based diagnosis as described above, non-monotonic reasoning for diagnosis does not use the classical logic inference to obtain diagnoses. The reason lies in that inferences in classical logic are monotonic with respect to their left argument. In the vast majority of formal systems studied in classical logic, for a set of premises T and formulae p, q it formally holds that

$$T \vdash q \Rightarrow T \cup \{p\} \vdash q.$$

That is, adding information cannot invalidate a previous conclusion ($\vdash$ denotes the classical logic inference relation). However, various practical considerations such as the formalisation of so-called *common-sense reasoning* (cf. Lifschitz [1995]) seem to suggest a fundamentally different kind of reasoning, where certain assertions and conclusions are assumed to hold until specific reasons are found to reject them. This seems to fit well also with the diagnostic problem, where a set of system observations are given and a model is sought, explaining them and the possible faults observed:

$$SD \cup \Delta \vdash OBS.$$

In monotonic logic, this approach is clearly infeasible, since the explanations for an observation, captured by the set Δ in the above formula, are infinitely many. An infinite amount of redundant or even intuitively contradictory information may be added to it without affecting the validity of the formula.

One of the early approaches to non-monotonic reasoning in the form of *default logic* is due to Reiter [1980], who formally introduced the notion of a "default", a special type of axiom which has to be assumed valid under normal circumstances until evidence is found to revoke this assumption. Similar approaches to reasoning with defaults are due to McCarthy [1987] in the form of *circumscription*, and Poole [1988] who provided an efficient implementation of *abductive reasoning* similar to the example given above. Formally, given a set of sentences T (i. e., a theory), and a sentence OBS (i. e., the observation), the abductive task is characterised as the problem of finding a minimal set of sentences Δ (i. e., the abductive explanation for OBS), such that

1. $T \cup \Delta \vdash OBS$, and
2. $T \cup \Delta$ is consistent.

This coarse characterisation of abduction is independent of the language in which T, OBS, and Δ are formulated. Hence, it has been used specifically to tackle the consistency-based diagnosis problem in first-order logic (cf. Poole [1988; 1989], Console et al. [1991], Poole [1994b]).

However, these types of default logic are, in general, not even semi-decidable (Poole [1994a]). Despite that, efficient implementations for default reasoning do exist, which are guided, e. g., by probabilistic information to consider only the likely explanations rather than the entire search space. Moreover, many of these implementations, like the one proposed for abduction by Poole or the one described by Flach [1994], are realised using the Prolog logic programming language (Colmerauer and Roussel [1993]). Prolog itself, whose programs are collections of Horn-clauses (i. e., quantified disjunctions of propositions with at most one positive occurrence of a proposition), is based upon a form of non-monotonic evaluation by using the so-called *closed world assumption* (CWA). The CWA has been first introduced by Reiter [1977] and states that with respect to a given collection of data about some subject, no relevant information is missing. Hence, if some information a is not part of a data set, $\neg a$ is assumed. Prolog's clause evaluation strategy treats negation as failure: it derives $\neg a$ from a data set D just in case a query for a given D fails. Since a may be derivable from a larger set of data, derivability under the CWA is said to be non-monotonic. However, there exist practical limitations to this evaluation scheme, since a must be of a suitably simple syntactic structure. Negation-as-failure for complex clauses is handled by heuristics and specific optimisations in the different implementations of the language (cf. Flach [1994]).

Diagnosis of discrete-event systems

The focus of previously described diagnosis approaches rests mainly on the analysis of static systems. Another branch of diagnosis is more devoted to dynamic analysis of so-called *discrete-event systems*. Basically, all dynamic systems that evolve via abrupt occurrences of (physical) events at possibly unknown irregular time intervals, qualify as discrete-event systems (Cassandras [1993]). Common examples of such events include the arrival of a job or the completion of a task. Analysis of discrete-event systems can be complex, since many such events strongly depend on each other, and their occurrence is often modelled using stochastic means, such as Markov chains and processes (cf. Chong [2000]).

Due to the dynamics of discrete-event systems, the approaches to their diagnosis are, in general, based on more involved and substantially larger system models, such as communicating and even timed automata (cf. Baroni et al. [1999], Bouyer et al. [2005a]). Many such approaches suffer from poor on-line performance or a so-called state-space "explosion", meaning that the model capturing the overall system state-space, e. g., in terms of a transition system, becomes too large for subsequent and automatic analysis.

Two approaches are predominant in the literature of diagnosis of discrete-event systems: the so-called *diagnoser-based systems*, as introduced by Sampath et al. [1994], and *decentralised diagnostic systems* as described, for instance, by Baroni et al. [1999].

A diagnoser is conceptually close to a monitor in runtime verification (see §4). It observes the events of a system under scrutiny and maps observations to possible failures. Although possible in a distributed setup, diagnoser-based methods are in particular applicable to isolated systems, because the employed system models capture both possible interactions as well as single events of components. Therefore, depending on the setup, these approaches suffer from the state-space "explosion". Schumann et al. [2004] circumvent this limiting factor by using OBDDs for an efficient state-space representation within the diagnoser.

For distributed systems, the decentralised approach is predominant. The system model then represents the individual components of a distributed discrete-event system as well as their interactions of events in terms of communicating automata, rather than a single off-line compiled model. This avoids the state-space "explosion" when diagnosing distributed systems, but potentially suffers from poor on-line performance as also noted by Schumann et al. [2004].

The Fault-Detection-and-Isolation approach to diagnosis

For the diagnosis of complex physical control systems where only mathematical models offer suitable means for description, methods of *Fault Detection and Isolation* (FDI), based on *analytical redundancy* (Chow and Willsky [1984]), are used. Typical examples include diagnosis of the control system for the combustion process inside a spark-ignited engine.

The way these systems work is depicted in Fig. 5.5. First, *residuals* are generated and then evaluated. A residual is a signal generated from some *parity equation* based on the actually measured values of the system under scrutiny. The parity equations are part of the mathematical and physical process model, and if the actual values observed conform with the values predicted by that model, the difference must be zero, or close to that. Other values indicate that a failure occurred. The fault detection in these methods then consists of creating various parity equations for diagnosable features of the system or process, and by evaluating the corresponding residuals as the system runs, often supported by statistical models, for instance.

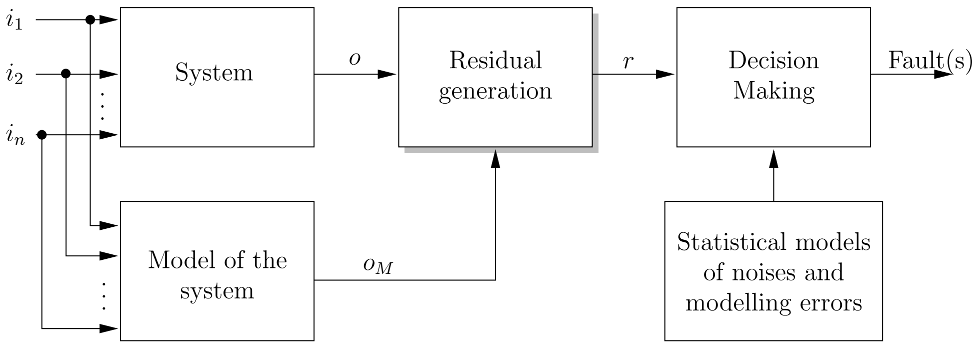


Fig. 5.5: The principle of Fault Detection and Isolation.

Naturally, diagnosis methods based on analytical redundancy target fundamentally different diagnosis problems than the logic-based ones, where the aim is to identify faulty parts of a (usually) distributed or component-based system. However, there are various results from combining methods of model-based diagnosis and analytical redundancy for the diagnosis of distributed control systems, such as found in present-day cars (cf. Struss and Malik [1997], Cordier et al. [2000], Struss and Price [2003], Nyberg and Krysander [2003]).

5.4 Summary

This chapter proposes a combination of methods of runtime verification and model-based diagnosis, in that the results of the monitors are used as input to the diagnostic process. The proposed combination aims at differentiating between specialised methods for the detection of symptoms (i. e., failures) and for the detection of their root causes (i. e., faults). Due to the combination, the computational complexity of model-based diagnosis can be substantially reduced, while the runtime verification approach described in the previous chapter could be shown as being optimal. Although the diagnosis problem remains inherently $\mathcal{NP}$-complete, the introduced mapping of first-order consistency-based diagnosis to a propositional satisfiability problem allows for a computationally efficient implementation. This is realised in terms of LSAT which is able to determine *all* satisfying assignments for a set of input clauses, thus solving the #SAT problem. Moreover, LSAT determines diagnoses based on the minimal cardinality of the faults contained in a diagnosis, and provides an effective pruning criterion for the overall problem search space (n-fault assumption). Details of the implementation are the subject of the next chapter.

Chapter 6

Implementation, tool-support, and comparative results

An algorithm must be seen to be believed.

(Donald E. Knuth, The Art of Computer Programming, Vol. 1*)*

THIS CHAPTER COULD ALSO BE ENTITLED "Putting it all together". It demonstrates the feasibility of the introduced algorithms and methods in terms of their actual implementation. It further presents some comparative results in the form of benchmarks, and shows how to integrate the proposed methods into a common framework for runtime reflection, which has been first outlined in this form by Bauer et al. [2006a].

Conceptually similar to a compiler for a programming language, the framework is separated into a *front end*, accepting the user-supplied specifications to monitor, and a *back end*, providing means for dynamic systems verification and deduction. In this chapter, implementation aspects of both are discussed. As far as the back end is concerned, only the domain-independent implementation of the monitoring and diagnosis layers are sketched, for logging and reconfiguration constitute, in general, domain-specific tasks. For logging, various standard logging mechanisms can be used, whereas for reconfiguration the system's application domain determines which mechanisms and techniques may be most suitable for a reconfiguration. Consider, for instance, the technical differences between a distributed (reactive) business information system used for banking applications, and a network of reactive and real-time sensitive embedded controllers with strict processing-, memory-, and communication-constraints.

Note that how to obtain and actually use the framework, which is developed under an open source license, is not subject of this chapter. Instead this is addressed briefly by Appendix B of this thesis.

6.1 Front end: An optimising compiler for SALT

Runtime reflection is a method that enables systems to reason about their current system state at runtime. As such, the approach is not strictly bound to a set of tools, but rather defined by the methods that constitute the analysis, i. e., monitoring and diagnosis, where temporal logic properties form the foundation for monitoring. However, for reasons already outlined in §3.3, plain temporal logic as used for the generation of monitors is often considered "too low-level" for specifying system properties error-free, concise, and human-readable. The specification language Salt, as proposed in §3.3 is thus treated in this chapter from a technical point of view in terms of being a front end for runtime reflection, and for making the framework accessible to its users.

Salt specifications are translatable into different logics (see §3.3). For instance, if no past and real-time operators are used in a specification, then it is translatable into a standard LTL formula, φ. The runtime reflection tool-set is then able to automatically generate an automaton $\mathcal{A}^{\varphi}$, such that $\mathcal{L}(\varphi) \equiv \mathcal{L}(\mathcal{A}^{\varphi})$. If no temporal operators are involved at all, then the output of a Salt translation is in the propositional domain. However, from the perspective of runtime reflection, this is not a very interesting case, and not what Salt is aimed for.

This section outlines the implementation of a compiler that accepts a Salt specification and returns, depending on the temporal operators in the specification, a temporal logic formula with or without past and with or without real-time operators. The Salt compiler has been realised as a Master's thesis project by Streit [2006], and was supervised by the author of this text, whereas first results of this work were published by Bauer et al. [2006c] (see also Bauer et al. [2006d]).

6.1.1 Internals of the SALT compiler

Like the runtime reflection framework itself, the Salt compiler consists of a front end and a back end. The front end is implemented in Java, while its back end, which optimises specifications for size, is realised in the functional programming language Haskell (cf. Wadler [1992], Peyton Jones [2005]). Basically, the back end is an executable realisation of the Salt semantics outlined in Appendix A.

As it is the case with other standard programming languages, compilation of Salt is undertaken in several *compilation stages*. First, user-defined macros, counting quantifiers and iteration operators are expanded to expressions using only a core set of Salt operators. Then, the Salt operators are replaced by expressions in the subset Salt--, which contains the full expressiveness of LTL/TLTL as well as exception handling and stop operators. The translation from Salt-- into LTL/TLTL is treated as a separate step since it requires "weaving" the abort conditions into all subexpressions. The result is an LTL/TLTL formula in form of an abstract syntax tree that is transformed in a straightforward manner into concrete syntax via a so-called *printing function*. The tool currently defines printing functions for the SMV and SPIN syntax, but users can provide additional printing functions to support a tool of their own choice other than runtime reflection, or the mentioned model checkers.

The use of optimised, context-dependent translation patterns as well as a final optimisation step performing local changes also helps reducing the size of the generated formulae.

6.1.2 Experimental results

Since the time that is required for temporal logic related analyses, such as LTL model checking, often depends exponentially on the size of the formula, and consequently, on the size of the corresponding Büchi automaton (cf. Schnoebelen [2002]), efficiency was an important issue also in the development of the SALT language and its compiler. One might suspect that formulae generated from abstract SALT specifications are bigger and less efficient to check than handwritten ones, but undertaken experiments show that this is not necessarily and usually the case.

In order to quantify the efficiency of the SALT compiler, existing LTL formulae were compared to the formulae generated by the compiler from a corresponding SALT specification. This was done for two data sets: the specification pattern system (Dwyer et al. [1999], 50 specifications) and a collection of real-world example specifications, mostly from the survey data provided by Dwyer et al. [1999] (26 specifications). The increase or decrease of the size was measured using the following parameters:

BA [Fri]: Number of states of the Büchi automaton (BA) generated by using the algorithm proposed by Fritz [2003].

BA [Odd]: Number of states of the Büchi automaton generated by using the algorithm proposed by Gastin and Oddoux [2001].

U: Number of **U**, **R**, **G** and **F** in the formula.

X: Number of **X** in the formula.

Boolean: Number of Boolean leafs, i. e., variable references and constants. This forms a suitable parameter for estimating the length of the formula.

The data for this experiment was obtained by first encoding the relevant temporal formulas manually in terms of SALT specifications, and then ensuring semantic equivalence between the translated formulae and the original formulae via the SMV model checker.

The results can be seen in Fig. 6.1. Formulae generated by the SALT compiler contain a greater number of Boolean leafs, but use *fewer temporal operators* and, therefore, also yield a smaller Büchi automaton. The error markers in the figure indicate the standard deviation.

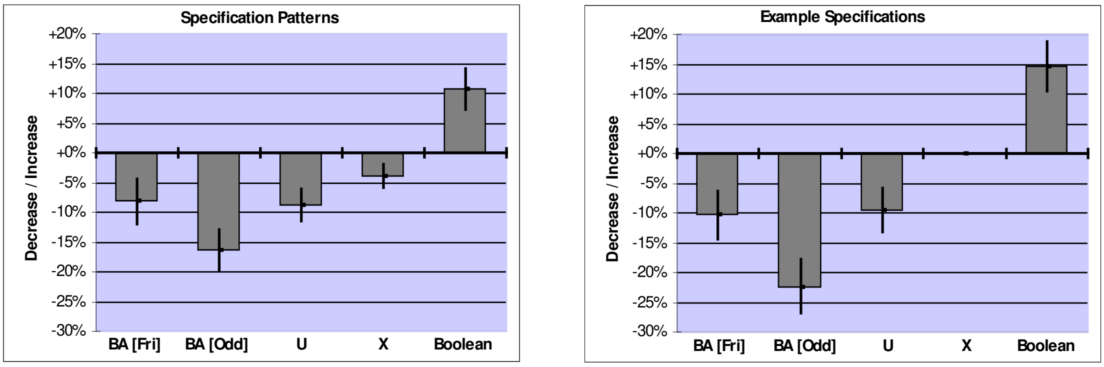


Fig. 6.1: Size of generated formulae.

Discussion

Using SALT for writing specifications does not generally degrade the verification efficiency, even for exhaustive procedures such as model checking. On the contrary, one can observe that it often leads to more succinct formulae (and thus, smaller Büchi automata).

The reason for this result is that SALT performs a number of optimisations. For instance, when translating a formula of the form $\varphi \mathbf{U}_w \psi$, the compiler can choose between the following two equivalent expressions

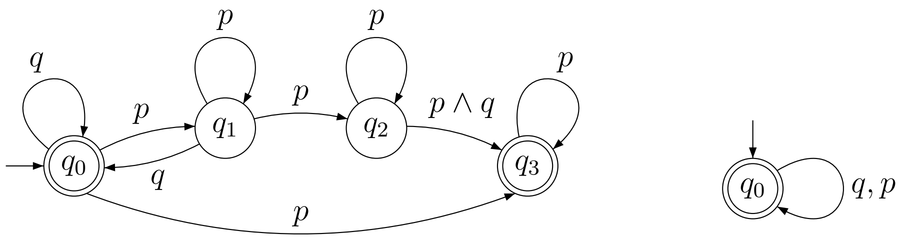
$$\neg(\neg\psi \mathbf{U}(\neg\varphi \wedge \neg\psi)) \quad \text{and} \quad (\varphi \mathbf{U} \psi) \vee \mathbf{G}\varphi.$$

While the first expression duplicates ψ in the resulting formula, the second expression duplicates φ, and introduces a new temporal operator. In most cases, the first expression, which is less intuitive for human readers, yields better technical results.

Another equivalence utilised by the compiler is: $\mathbf{G}(\varphi \mathbf{U}_w \psi) \Leftrightarrow \mathbf{G}(\varphi \vee \psi)$. With $\varphi \mathbf{U}_w \psi$ being equivalent to $(\varphi \mathbf{U} \psi) \vee \mathbf{G}\varphi$, the left hand side reads as $\mathbf{G}((\varphi \mathbf{U} \psi) \vee \mathbf{G}\varphi)$. When φ and ψ are propositions, this expression results in a Büchi automaton with four states using the algorithm proposed by Fritz [2003]. $\mathbf{G}(\varphi \vee \psi)$, however, is translated into a Büchi automaton with only a single state (see Fig. 6.2).

However, the benefit obtained from using the SALT approach is not intrinsic. The rewriting of LTL formulae could be done without having SALT as a high-level language. What is more, given an *LTL-to-Büchi translator* that produces a minimal Büchi automaton for the language defined by a given formula, no optimisations on the formula level would be required, and such a translation function exists—at least theoretically.[1]

[1] As the class of Büchi automata is enumerable and language equivalence of two automata decidable, it is possible to enumerate the class of automata ordered by size and take the first one that



(a) $\mathbf{G}((p\mathbf{U}q) \vee \mathbf{G}p)$ (b) $\mathbf{G}(p \vee q)$

Fig. 6.2: Two equivalent Büchi automata generated by the algorithm as proposed by Fritz [2003].

6.2 Back end

The back end of the runtime reflection framework consists of the layers depicted in Fig. 1.1 in the introduction of this thesis. The following technical description of it focuses on implementation aspects of those layers which are laid out in a domain-independent manner, used for diagnosis and monitoring. Recall that, for instance, the logging layer is often predetermined by an already-in-place logging facility in the system under scrutiny (e. g., LOG4J in many real-world Java applications, cf. Gunter et al. [2002], or for C++, the counterpart LOG4CXX, see also §4.4.2). Hence, it is assumed that both layers, logging and reconfiguration, have to be instantiated by some custom mechanisms suitable for the system to be analysed. Monitoring and diagnosis, however, are laid out in a domain-independent manner, and can be used in combination, or also separately, depending on the computational resources available in a system (e. g., CPU and memory). In the remainder, implementation schemes for both are developed, and further optional optimisations suggested. Moreover, it is shown how to integrate the individual layers to a common framework as well as how to use them separately as stand-alone tools.

6.2.1 Monitoring: implementation and case-study

This section proposes an efficient implementation scheme for the monitoring approach developed in §4 in the untimed case, and which was first described in this form by Arafat et al. [2005]. The scheme consists of a code generator, resulting in monitors that require minimal space with respect to the language being monitored and which proceed by using an on-the-fly power-set construction. Additionally, this section demonstrates, using the case-study briefly discussed in §4.4.2, how monitoring can be integrated into a real-world (C++) application.

is equivalent to the one to be minimised. However, such an approach is not feasible in practice.

Code generation scheme

Recall, the approach for producing a monitor for a given LTL formula φ as described involves the following steps. First, for a given LTL formula φ, a corresponding nondeterministic Büchi automaton $\mathcal{A}^\varphi$ must be constructed. Second, the nondeterministic Büchi automaton $\mathcal{A}^\varphi$ must be transformed into a nondeterministic finite automaton $\hat{\mathcal{A}}^\varphi$, which will then be determinised to yield the deterministic finite automaton $\tilde{\mathcal{A}}^\varphi$.

Since the automata are defined with respect to an alphabet $\Sigma = 2^{AP}$, each symbol $a \in \Sigma$ corresponds to a finite set of atomic propositions. A set $a \subseteq AP$ represents the assignment which evaluates a proposition $p \in AP$ to *true* if and only if $p \in a$ holds. Thus, the transitions of the automata can be denoted as a tuple (s, Φ, s'), where s is the original state, Φ is a formula over the set of propositions AP, and s' is the new state. Such a transition (s, Φ, s') is *enabled* for a given alphabet symbol $a \subseteq AP$, if Φ is satisfied by a.

Following Lemma 4.4.3, the corresponding nondeterministic Büchi automaton $\mathcal{A}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, F^\varphi)$ is then transformed into a nondeterministic finite automaton $\hat{\mathcal{A}}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, \hat{F}^\varphi)$, by checking for every state $q \in Q^\varphi$ whether the language accepted by $\mathcal{A}^\varphi(q)$ is empty or not (recall further, $\mathcal{A}^\varphi(q)$ uses q as initial state but is otherwise identical to $\mathcal{A}^\varphi$). $\mathcal{A}^\varphi(q)$ accepts an ω-word, if and only if starting at q, a final state $q' \in F^\varphi$ can be reached which is a member of a non-trivial strongly connected component, i. e., there must be a cycle in the state-transition graph which leads from q' back again to q'. This process is repeated for the negated formula $\neg\varphi$ in order to obtain the corresponding deterministic finite automaton, $\tilde{\mathcal{A}}^{\neg\varphi}$, and finally to obtain the finite state machine, $\bar{\mathcal{A}}$, with a cross-product construction (see Definition 4.4.3).

Typically, an explicit generation of the finite state machine $\bar{\mathcal{A}}$ causes a double exponential "blowup", firstly for building the nondeterministic Büchi automaton $\hat{\mathcal{A}}^\varphi$, and secondly for computing the corresponding deterministic finite automaton $\tilde{\mathcal{A}}^\varphi$. For this reason, the explicit construction of two deterministic finite automata, $\tilde{\mathcal{A}}^\varphi$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ is not undertaken, in favour of an *implicit representation* of the finite state machine $\bar{\mathcal{A}}$ by means of two nondeterministic finite automata $\hat{\mathcal{A}}^\varphi$ and $\hat{\mathcal{A}}^{\neg\varphi}$. In other words, for each of the two nondeterministic finite automata (NFA), a C++-class is generated which implements the `NFA` interface and thus, offers the following three functions:

- $getSuccessors(s, a)$ takes a state s and a subset $a \subseteq AP$ of atomic propositions and returns the set of successors reachable from s by a. That is, for every transition (s, Φ, s') in the transition table, it is checked whether a satisfies Φ and, if so, s' added to the result set.
- $isFinal(s)$ returns *true* (or, *false*) if s is a final state (not a final state, correspondingly).
- $initialStates()$ returns the set of initial states of the nondeterministic finite automaton.

To make a nondeterministic finite automaton deterministic dynamically at runtime, and without explicitly storing its comprehensive look-up table, a class `DFA` is defined, similar to the interface above, which “wraps” an `NFA` object and provides the following functions:

- $getSuccessor(s, a)$ takes a state s and a subset $a \subseteq AP$ of atomic propositions and returns a single successor reachable from s by a.
- $isFinal(s)$ returns *true* (or, *false*) if s is a final state (not a final state, correspondingly).
- $initialState()$ returns the single initial state of the deterministic finite automaton.

```
StateSet DFA :: initialState ()
{
    return (nfa.initialStates ());
}

StateSet DFA :: getSuccessor (StateSet S, PropositionSet a)
{
    StateSet result;
    for_all s in S
        result.add (nfa.getSuccessors (s, a));
    return result;
}

bool DFA :: isFinal (StateSet S)
{
    for_all s in S
        if (nfa.isFinal (s))
            return true;
    return false;
}
```

Fig. 6.3: Implementation of the `DFA` functions.

A single state of the deterministic finite automaton corresponds to a set of states of the nondeterministic finite automaton. To implement the functions described above, a `DFA` object uses a reference to the corresponding `NFA` object in order to compute the state transitions of the `DFA` object in an *on-the-fly manner*, as shown in Fig. 6.3 which uses a C++-inspired pseudo code. This code is independent of the formula φ and the underlying `NFA` object, and is, therefore, implemented once, manually rather than automatically generated. Finally, the finite state machine is implemented in a similar on-the-fly fashion: the constructor of the `FSM` class takes two references which point to the `DFA` objects which implement $\tilde{\mathcal{A}}^{\varphi}$ and $\tilde{\mathcal{A}}^{\neg\varphi}$ and stores them in the fields `dfa_pos` and `dfa_neg`. Furthermore, the finite state machine maintains the current state of the two deterministic finite automata in the fields `state_pos` and `state_neg`, respectively. The `FSM` class then provides a function, `processInput`, which takes a subset, a, of propositions from AP and returns the current evaluation of the system trace with respect to φ as can be seen in Fig. 6.4.

```
BoolThree FSM :: processInput (PropositionSet a)
{
    state_pos = dfa_pos.getSuccessor (state_pos, a);
    state_neg = dfa_neg.getSuccessor (state_neg, a);
    if (!dfa_pos.isFinal (state_pos)) return false;
    if (!dfa_neg.isFinal (state_neg)) return true;
    return ?;
}
```

Fig. 6.4: Implementation of the main processing function in the monitor.

Case study: C++ static initialisation order fiasco

This section details on a case study, which has already been described briefly in §4.4.2, and which has been undertaken with the monitoring scheme developed above. Recall that in C++ the initialisation order of static objects is nondeterministic. Hence, for large C++ applications, consisting of many independently developed code modules (and static objects), it is a major concern to ensure an appropriate startup behaviour of the overall system. One way to check for a "legal" startup behaviour, i. e., where no threads are spawned before program initialisation has completed, is to use monitors (see also §4.4.2).

For the purpose of implementing threads in a portable manner, most (UNIX-based) systems rely on the POSIX standard for threads (in short, pthreads; see IEEE [1995]). The application under scrutiny then has to be instrumented to employ a logging facility for emitting relevant system events, such as the creation or destruction of a thread. In theory, there are no limitations on the employed logging framework. However, in this case study a custom solution was used, which is described in greater detail, e. g., by Arafat [2005], and which is also available under an open source license as part of the runtime reflection framework (see Appendix B). Other logging facilities, such as the Apache Software Foundation's library, log4cxx are also possible, but naturally rely on a different API for code instrumentation.

After instrumentation of the code, the logging facility allows users to attach so-called *loggers* to the stream of system events. Each time a relevant system event described by an annotation occurs, the logging function of all registered loggers is invoked to write the exact contents of the event onto hard disk, or to send it to a remote *server* for further analysis. Note that in this framework, a monitor is but a special kind of logger which can be automatically generated from an LTL formula, and whose "analysis server" is the diagnosis layer as described in the next section.

The actual instrumentation in this study occurred by providing so-called *code wrappers* around relevant system calls, such as `pthread_create` which in accordance to the POSIX standard may have the following signature:

```
int pthread_create (pthread_t *threadp,
                    pthread_attr_t const *attr,
                    void* (*start_routine) (void*),
                    void* arg) __THROW
```

The macro provided by the logging layer, `DIAGNOSTICS_AUDIT_INSTRUMENT_C_CALL`, "wraps" this system call such that any call to `pthread_create` (provided by the system library `libpthread.so.0`) emits an input event for the registered loggers; hence, the name *code wrappers*. The code responsible for this is given in Fig. 6.5. Naturally, if other logging facilities are used but the one provided by the runtime reflection framework, the instrumentation will use a different set of calls and parameters.

```
#include <pthread.h>
#include <diagnostics/instrumentation.hpp>

extern "C" DIAGNOSTICS_AUDIT_INSTRUMENT_C_CALL
("libpthread.so.0",                       // Library providing
                                          // pthread_create.
 int,                                     // Result type.
 pthread_create,                          // Method name.
 (pthread_t *threadp,                     // Argument types.
  pthread_attr_t const *attr,
  void* (*start_routine) (void*),
  void* arg),
 __THROW,
 (threadp, attr, start_routine, arg), // Arguments to THROW.
 "");                                     // Additional comments for
                                          // this system event.
```

Fig. 6.5: Wrapper around `pthread_create`.

The interface for the automatically generated monitor `SIOF_Monitor` (short for "static initialisation order fiasco monitor") is given in Fig. 6.6. Its implementation is depicted in Fig. 6.7. Basically, the monitor receives a record entry resembling a verbose variant of a system event, including possible comments, and then evaluates its content in terms of a bit vector encoding the truth values of propositions for an observed event. The constructor first integrates the generated classes for the two nondeterministic finite automata, namely `Pos_static_fiasco` and `Neg_static_fiasco`. Further, the constructor sets the `m_bit_vector` to its initial state. The `p_translate` function interprets each logged `Record_t` in terms of a `Bit_Vector_t`: If a relevant log message occurs, it updates the `bit_vector` accordingly and returns *true*. Otherwise, it only returns *false*.

Finally, Fig. 6.8 shows the `main` function, i. e., the starting point of the application, whose sole purpose it is to attach a monitor to the stream of system events. The call to `set_initial_loggers` provides an initial set of `Logger` objects, even before entering

```
#ifndef EXAMPLE_SIOF_MONITOR_HPP
#define EXAMPLE_SIOF_MONITOR_HPP

#include <ltl2fsm/monitor_code/Monitor_Wrapper.hpp>

class SIOF_Monitor : public :: ltl2fsm :: Monitor_Wrapper
{
public:
    SIOF_Monitor ();
    virtual ~SIOF_Monitor ();
protected:
    virtual bool p_translate (Record_t const &record,
                              Bit_Vector_t &bit_vector);
};

#endif
```

Fig. 6.6: The interface of the automatically generated monitor.

the `main` function. When the first system event occurs, the logging facility is activated and calls `set_initial_loggers` during initialisation. The `main` function starts with a `DIAGNOSTICS_PROD_PROCEDURE_GUARD` annotation that generates a system event on entry and exit of this function. Moreover, after entering `main`, a thread is created which starts with `start_func` and awaits its termination. At the end of `main`, the state of `Siof_Monitor` is returned. In this case, since the first thread is spawned after `main` has been reached, the `Siof_Monitor` is in the state *true*.

Integration and work-flow

The resulting work-flow of the above described procedure, including the compilation of Salt specifications is summarised in Fig. 6.9: From a Salt specification a temporal logic formula φ is generated (in this picture, $\varphi \in \text{LTL}$), and then a Büchi automaton is generated for φ and $\neg\varphi$, respectively. The FSM-Generator then creates C++ code for a finite state machine which performs the power-set construction on-the-fly, and analyses the stream of events according to the 3-valued semantics of φ.

6.2.2 Diagnosis: implementation and benchmarks

Like the monitoring layer, diagnosis in the runtime reflection framework is implemented using the C++ programming language to facilitate ease of integration with the other constituents, and for gaining optimal runtime performance by using uninterpreted code. However, this choice is not inherent in the theoretical approach to diagnosis. For instance, the Salt compiler is realised in a combination of the languages Java and Haskell.

```
#include "siof_monitor.hpp"

#include <generated/Neg_static_fiasco.hpp>
#include <generated/Pos_static_fiasco.hpp>

#include <ltl2fsm/monitor_code/Fsm.hpp>
#include <ltl2fsm/monitor_code/Dfa.hpp>
#include <diagnostics/frame/record.hpp>

using namespace ltl2fsm;

SIOF_Monitor :: SIOF_Monitor() : Monitor_Wrapper
(new Fsm (new Dfa (new Pos_static_fiasco),
          new Dfa (new Neg_static_fiasco)), 2)
{
    // Action: pthread_create is not called initially.
    m_bit_vector[0]=false;
    // Action: main starts not immediately.
    m_bit_vector[1]=false;
}

SIOF_Monitor :: ~SIOF_Monitor ()
{
}

#define WHAT_MAIN "PROCEDURE=\"int main ()\""
#define WHAT_PC   "PROCEDURE=\"int pthread_create ("

bool SIOF_Monitor :: p_translate(Record_t const &record,
                                 Bit_Vector_t &bit_vector)
{
    using namespace diagnostics;
    if (record.type () == TYPE_PROCEDURE_ENTER
        && record.str_what ().find (WHAT_MAIN) == 0) {
         bit_vector[1] = true;
         return true;
    }
    if (record.type () == TYPE_PROCEDURE_ENTER
        && record.str_what ().find (WHAT_PC) == 0) {
         bit_vector[0] = true;
         return true;
    }
    return false;
}
```

Fig. 6.7: The code of the automatically generated monitor.

```
#include <diagnostics/annotations.hpp>
#include <diagnostics/configuration.hpp>

#include "siof_monitor.hpp"

#include <pthread.h>

static ltl2fsm :: Monitor_Wrapper* siof_monitor;

using namespace std;

DIAGNOSTICS_NAMESPACE_BEGIN;
void set_initial_loggers (vector<Logger*> &loggers)
{
    loggers.push_back (siof_monitor = new SIOF_Monitor);
}
DIAGNOSTICS_NAMESPACE_END;

void* start_func (void*)
{
    DIAGNOSTICS_PROD_PROCEDURE_GUARD ("");
    return NULL;
}

int main ()
{
    DIAGNOSTICS_PROD_PROCEDURE_GUARD("");

    pthread_t tid;
    void **return_value;
    pthread_create (&tid, NULL, &start_func, NULL);
    pthread_join (tid,return_value);

    cout << siof_monitor->status () << endl;
    return 0;
}
```

Fig. 6.8: Attaching the monitor to the stream of system events.

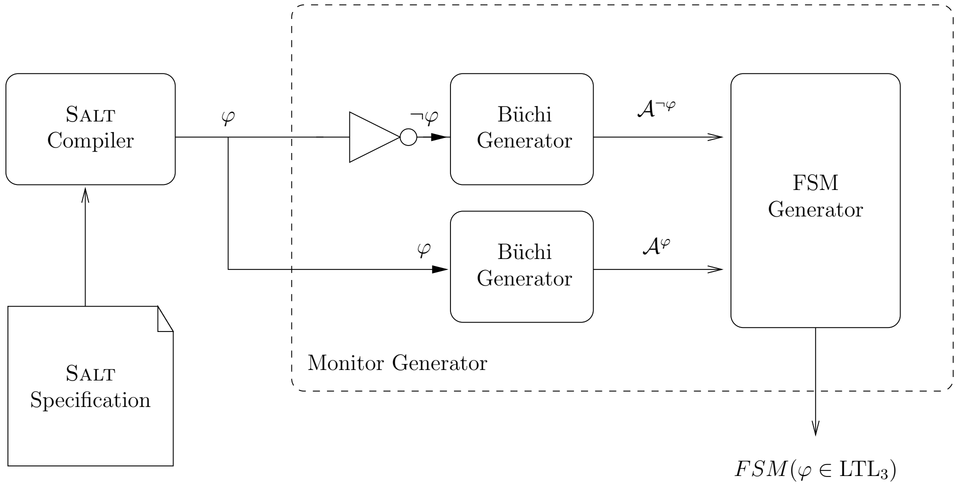


Fig. 6.9: Work-flow from specification to monitor generation.

The core of the diagnosis layer consists of a #SAT-solver outlined in §5 of this thesis, and referred to as LSAT. The internals of LSAT were first described by Bauer [2005].

This section details on essential implementation details of LSAT, such as data structures used, optional optimisations, and comparative benchmarks. A brief discussion of an integration scenario into a model-based design- and work-flow finalises this section, which is also described in greater detail by Bauer et al. [2007].

Data structures

While LSAT's solving algorithm has been previously laid out in §5, this section focusses on its employed data structures and possible improvements that can increase the efficiency of LSAT further. Basically, LSAT is realised by two main functions, `branch` and `refute` (for details, see §5). Both functions share a set of global variables in order to keep the space complexity for storing runtime information linear during execution. The alternative to this would be to keep all variables local, but on the expense of storing changes to them that need to be propagated and kept on the runtime stack when executing.

Fig. 6.10 shows the main two data structures used by LSAT in a schematic manner. Every variable $v_i \in V$ accessed by either function corresponds to an object, `Variable`, and every clause $C_i \in \mathcal{C}$ to an object, `Clause`, where V denotes, again, the set of variables and $\mathcal{C}$ the set of clauses over V used by LSAT's input formula. `Variable` has the following elements:

- `index`: An index to identify a variable unambiguously.
- `value`: A truth-value assignment, if made by the algorithm.
- `abnormal`: A flag, denoting whether or not this variable corresponds to a component of a diagnosis problem $(SD, COMP, OBS, SIG)$, such that $v \equiv c \in$

Variable
index:int
value:int
abnormal:bool
pos-clauses:List<Clause>
neg-clauses:List<Clause>

Clause
pos-literals:List<Variable>
neg-literals:List<Variable>
inact:Variable
literals:int

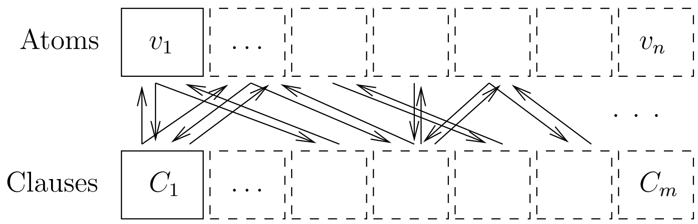


Fig. 6.10: LSAT's internal representation visualised with v_i:`Variable` and c_i:`Clause`.

COMP.

- `pos-clauses`: A (possibly empty) list of clauses where v occurs positively.
- `neg-clauses`: A (possibly empty) list of clauses where v occurs negatively.

`Clause` consists of the following entries:

- `pos-literals`: A (possibly empty) list of literals which occur positively in a clause C.
- `neg-literals`: A (possibly empty) list of literals which occur negatively in a clause C.
- `inact`: A pointer to a variable, which is responsible for disregarding the clause C from future variable assignments in the course of the algorithm.
- `literals`: A list of literals that appear in this clause, positively or negatively.

By keeping a set of lists in both data structures, a mutually linked list is created which is also depicted schematically in the lower part of Fig. 6.10: variables have a reference to the clauses they appear in, whereas clauses have a reference to the variables occurring in them. In C++, this can be realised efficiently by using *pointers*, such that access to either data structure occurs via dereference.[2]

[2] Naturally, care has to be taken when storing pointer locations in container objects of the C++ Standard Template Library (STL) that use semi-automatic memory management.

```
void branch (void)
{
  Variable* atom;

  if ((atom = find_unmarked ())) {
    refute (atom, +1); refute (atom, -1);
  }
  else {
    // Print set of satisfying assignments.
    cout << solution << endl;

    // If #SAT must be solved, do not interrupt.
    if (! required_all_solutions ())
      exit (0);
  }
}
```

Fig. 6.11: Function `branch`.

Unit propagation as optimisation

The actual implementation of LSAT as outlined in §5 is given in Fig. 6.11 and Fig. 6.12 in a C++-like notation.

Note that the actual program LSAT (see Appendix B) contains unit propagation as an optimisation as originally suggested for DPLL (see §5.2.2). A unit clause exists, if and only if there exists a clause with only a single (unassigned) literal. Detection of unit clauses speeds up execution since the variable assignment is obvious from the clause. If, before `branch` is called by `refute`, all unit clauses are iteratively deactivated, the size of the remaining semantic tree to be expanded can be significantly reduced. For brevity, this has not been outlined earlier as unit propagation integrates into the algorithm transparently and does not affect important properties, such as asymptotic memory requirements. To see how it works, the changed code fragment from `refute` is given in Fig. 6.13.

Unit propagation works iteratively until no more unit clauses can be found using the currently determined assignment. Upon backtracking, however, all these unit clauses are reactivated, because upon using a new assignment, they may not account as unit clauses any longer. Unit propagation works by identifying those previously unassigned variables that (now) occur as singletons, and by deactivating the corresponding clauses until no more singletons can be found. Reactivation occurs symmetrically.

Example

Like the monitoring layer, LSAT can be used as a stand-alone tool to solve the general SAT or #SAT problem. If used stand-alone, LSAT accepts an extended DIMACS

```
void refute (Variable* lit, int sign)
{
  switch (sign) {
  case 1: {
    tmp_clauses := neg_clauses (lit);
    true_clauses := pos_clauses (lit);
    neg_clause_counter (tmp_clauses, -1);
    break; }
  default: {
    tmp_clauses := pos_clauses (lit);
    true_clauses := neg_clauses (lit);
    pos_clause_counter (tmp_clauses, -1);
    break; }
  }

  solution.push_back (sign * lit->idx ());

  if (!empty_clause (tmp_clauses)
      && fault_counter () <= max_faults ()) {
    vector<Variable*> units;

    // Mark and deactivate.
    lit->set_val (1);
    activate (true_clauses, lit);

    // Branch and select new.
    branch ();

    // Unmark and reactivate.
    lit->set_val (0);
    activate (true_clauses, NULL);
  }

  // Backtracking.
  (solution.back () > 0 ?
   neg_clause_counter (tmp_clauses, +1) :
   pos_clause_counter (tmp_clauses, +1));
  solution.pop_back ();
}
```

Fig. 6.12: Function `refute`.

```
void refute (Variable* lit, int sign)
{
  // ...

  Variable* tmp;

  // Iterative unit propagation.
  do
    {
      if ((tmp = deactivate_unit_clauses (del_clauses,
                                          decr_clauses,
                                          the_sign)))
        {
          all_del_clauses.push_back (del_clauses);
          all_decr_clauses.push_back (decr_clauses);
          all_signs.push_back (the_sign);
        }
      del_clauses.clear ();
      decr_clauses.clear ();
      units.push_back (tmp);
    } while (tmp);

  // Branch and select new.
  branch ();

  // Iterative unit clause reactivation.
  for (unsigned i = 0; i < units.size (); i++)
    {
      reactivate_unit_clauses (all_del_clauses[i],
                               all_decr_clauses[i],
                               all_signs[i],
                               units[i]);
      del_clauses.clear ();
      decr_clauses.clear ();
      all_signs.clear ();
    }

  // ...
}
```

Fig. 6.13: Iterative unit propagation in `refute`.

input format which is used to describe a set of clauses in CNF.[3]

In DIMACS format, all variables are encoded by discrete numbers, such that a set of clauses in CNF, given by $\mathcal{C}$, must be mapped to unique integers. One such possible mapping for the example depicted in Fig. 5.2 on p. 107 of system components and observations to unique integers is given as follows: $i_1 \mapsto 1, i_2 \mapsto 2, i_3 \mapsto 3, i_4 \mapsto 4, m_1 \mapsto 5, m_2 \mapsto 6, o_1 \mapsto 7, o_2 \mapsto 8, M_1 \mapsto 9, M_2 \mapsto 10, A_1 \mapsto 11, A_2 \mapsto 12$. Using LSAT's extended DIMACS format, the example, where m_1 and m_2 cannot be observed, and o_1 and o_2 are not as expected, could then be encoded and automatically solved as follows.

`p cnf 12 22`	Standard DIMACS header.
`9 -1 -2 5`	*SD:* causal dependencies of S.
`10 -3 -4 6`	$(10 \vee -3 \vee -4 \vee 6)$
`11 -5 -6 7`	$\bigwedge$ $(11 \vee -5 \vee -6 \vee 7)$
`12 -6 -5 8`	$\bigwedge$ $\ldots$
`-5 1`	
`-5 2`	
`-5 -9`	
`-6 3`	
`-6 4`	
`-6 -10`	
`-7 5`	
`-7 6`	
`-7 -11`	
`-8 5`	
`-8 6`	
`-8 -12`	
`a 9 10 11 12`	*COMP:* the directive `a` defines the components in S.
`1`	*OBS:* $ok(1)$
`2`	$\ldots$
`3`	
`4`	
`-7`	$\neg\, ok(7)$.
`-8`	$\neg\, ok(8)$.

(Given sufficient runtime) LSAT finds a satisfying assignment for the above input formula, if at least one such assignment exists, where the set of input clauses resembles a model-based diagnosis problem $\mathcal{M} = (SD, COMP, OBS, SIG)$. Moreover, in this example, a 2-fault-assumption has been used, since m_1 and m_2 are not observable, and increase the set of potential diagnoses. Unobservable model artifacts which need to be reflected in the reasoning, are called *unknowns*, and are denoted by a set $UNOBS$.

[3]DIMACS is the *Center for Discrete Mathematics and Theoretical Computer Science*, Rutgers, New Jersey, and provides a de-facto standard for SAT-solver input formats (cf. Cook and Mitchell [1997]).

It follows that the presence of unknowns raises the number of possible results for diagnosis in the worst case to $2^{(|COMP|+|UNOBS|)}$. Hence, it is sensible in many cases to restrict the number of possible diagnoses by using an appropriate n-fault assumption. In this case, let $n = 2$. Assumptions concerning symptoms in the obtained result are underlined, whereas assumptions regarding abnormal components are denoted using frames:

1 $\boxed{12}\ \underline{-8}\ 1\ 2\ 3\ 4\ \underline{-7}\ \boxed{11}\ -10\ 6\ -9\ 5$
2 $\boxed{12}\ \underline{-8}\ 1\ 2\ 3\ 4\ \underline{-7}\ -11\ \boxed{10}\ \underline{-6}\ -9\ 5$
3 $\boxed{12}\ \underline{-8}\ 1\ 2\ 3\ 4\ \underline{-7}\ -11\ -10\ 6\ \underline{-5}\ \boxed{9}$
4 $-12\ 1\ 2\ 3\ 4\ \underline{-7}\ \underline{-8}\ \boxed{11}\ \boxed{10}\ \underline{-6}\ -9\ 5$
5 $-12\ 1\ 2\ 3\ 4\ \underline{-7}\ \underline{-8}\ \boxed{11}\ -10\ 6\ \underline{-5}\ \boxed{9}$
6 $-12\ 1\ 2\ 3\ 4\ \underline{-7}\ \underline{-8}\ -11\ \boxed{10}\ \underline{-6}\ \boxed{9}\ \underline{-5}$
7 $-12\ 1\ 2\ 3\ 4\ \underline{-7}\ \underline{-8}\ -11\ \boxed{10}\ \underline{-6}\ -9\ 5$
8 $-12\ 1\ 2\ 3\ 4\ \underline{-7}\ \underline{-8}\ -11\ -10\ 6\ \underline{-5}\ \boxed{9}$

Each of the eight results (out of a total of 13) encodes exactly one valid truth assignment for $\mathcal{M}$, such that the observations, $\neg ok(o_1)$ and $\neg ok(o_2)$, can be explained under the assumption that not more than two components in the system are responsible for the deviations. The top-most result, for instance, gives the explanation $AB(A_2) \wedge AB(A_1)$, with $A_2 \mapsto 12$ and $A_1 \mapsto 11$ under the assumption that both m_1 and m_2 are as expected.

Benchmarks

With the implementation of the n-fault assumption, LSAT targets foremost model-based diagnosis in a propositional domain, where monitors provide the assignment of literals that encode observations of system events. However, LSAT is not restricted to this setup, and in order to demonstrate applicability of the proposed solution, a number of standard tests have been performed that were taken from the domain of hardware verification. The so-called ISCAS (short for *IEEE International Symposium on Circuits and Systems*) set of benchmark circuits is widely used by the hardware verification community for validating digital design and verification tools. The respective combinatorial tests resemble the logic underlying integrated circuits, and range from several hundred to ca. 20,000 "components" (i. e., logical gates), and ca. 60,000 clauses (i. e., connections between gates).

For the purpose of validation, the respective designs were altered at random to induce combinatorial faults, which could only be resolved by revoking assumptions regarding the "health state" of one of many logical gates.

Table 6.1 summarises results of applying a selection of ISCAS test cases to LSAT on a Pentium 4 architecture with 512 MB of RAM using either the 5-fault assumption, or no restriction at all. If a test could not be finished within 60 seconds, it was considered to be a timeout. The results obtained substantiate the appropriateness of the presented concept. Four tests could not be finished using the ∞-fault assumption, while

Table 6.1: Modified ISCAS‘89 benchmarks under the n-fault assumption.

				∞-fault		5-fault	
Name:	#*COMP*:	#Var.:	#Cl.:	#Steps:	CPU:	#Steps:	CPU:
s208.1	66	122	389	84	0.17 sec	60	0.25 sec
s298	75	136	482	27	0.11 sec	58	0.32 sec
s444	119	205	714	20	0.18 sec	105	0.91 sec
s526n	140	218	833	–	timeout	295	0.23 sec
s820	256	312	1,335	–	timeout	562	0.59 sec
s1238	428	540	2,057	38	0.97	262	0.21 sec
s13207	2,573	8,651	27,067	–	timeout	17	0.57 sec
s15850	3,448	10,383	33,189	–	timeout	41	0.17 sec
s35932	12,204	17,828	60,399	2,339	11.16 sec	29	0.21 sec

Lsat solved these tasks when constrained to five faults. The variance between CPU time and the performed number of algorithmic steps can be explained by the heuristic approach and variantly efficient object-accessor functions in the implementation of the Lsat tool itself.

Integration and work-flow

Lsat as described above is both an efficient diagnosis engine, as well as an efficient solver for the #SAT, respectively SAT, problem. If used for the purpose of model-based diagnosis, its set of input clauses $\mathcal{C}$ consists of a monitoring-based diagnosis problem $(SD, COMP, OBS, SIG)$, where $SD, COMP$ can be obtained, e. g., from a CASE-tool used for system modelling, and where the monitors then provide the set of observations OBS (see Fig. 6.14).

There are no constraints on the type of CASE-tool used for extraction of the system model as long as the causal relations between the input and output signals of subcomponents of a system is provided. Bauer et al. [2007] describe how to use MATLAB/Simulink (Mathworks [2000]) for this purpose, a systems modelling tool which is predominant, e. g., for the design and implementation of embedded control systems. However, other CASE-tools, such as AutoFocus (see §4.1 and Appendix B) are also possible to integrate. The paper further describes how Lsat (in combination with other constraint solvers) can be used to verify extracted system models statically with respect to a set of predetermined properties. The technical as well as theoretical details of this step are beyond the scope of this thesis. However, the cited paper does provide the necessary details, and a further case study, taken from the automotive domain.

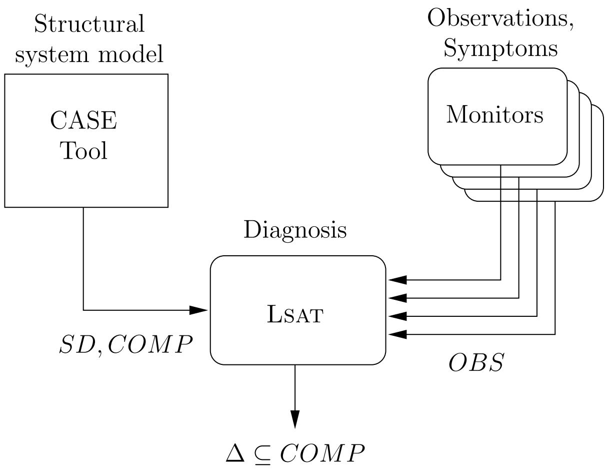


Fig. 6.14: Integration of system models, LSAT, and the monitors.

6.3 Summary

This chapter represents the most "hands-on" part of this thesis. It proposes concrete and efficient implementation schemes for the methods developed in the previous chapters of this thesis. In particular, an optimising compiler for SALT is discussed, an efficient monitoring and code generation scheme that relies upon on-the-fly determinsation of nondeterministic finite automata, and a memory-efficient realisation of the diagnosis layer LSAT is given. Various implementation aspects are substantiated by concrete C++ code examples, comparative benchmarks, and a real-world case-study, showing the steps involved to integrate monitors into an actual C++ system.

Chapter 7

Conclusions

If I have seen further it is by standing on ye shoulders of Giants.

(Sir Isaac Newton, Letter to Robert Hooke, 1676*)*

THIS CHAPTER briefly summarises some important results of this thesis, and gives an outlook on possible future research directions.

7.1 Summary

Runtime reflection as developed in this thesis can be broadly characterised as a *method* that enables systems to reflect upon their overall system status at runtime. The *formal foundations* as well as efficient *means for their implementation* are the heart of this text. Various conclusions can be drawn from it.

The presented approach to runtime reflection shows that a *methodological differentiation* between the detection and the diagnosis of failures is sensible. Failure detection by means of runtime verification is an efficient way of detecting symptoms of a system failure, but not good at differentiating the symptoms of failure from their actual causes. Therefore, this thesis argues in favour of a methodological differentiation between these tasks, and puts forth an approach using a strong separation of concerns: (1.) It develops a specific technique which is efficient in the detection of symptoms of system failures, and (2.) a specific technique that aims primarily at identifying meaningful explanations for them in terms of isolating those (faulty) system components which may be responsible for the symptoms.

Runtime reflection as developed in this thesis is not specific to a certain domain, such as mechanical engineering or control systems. Therefore, it can be argued that the abstract requirement to detect symptoms of a system failure is a completely different one to that of identifying causes, and should, as described in this text, be realised by different techniques.

The approach to failure detection is technically based upon the construction of monitor components from temporal logic specifications. The specifications encode desired system behaviour and are interpreted over infinite sequences of (observable system) events. However, since monitors have to interpret such specifications over finite sequences, a *3-valued semantics* is used to give the temporal logics LTL, respectively TLTL, a determined and unambiguous meaning with respect to the finite view on the system.

For both logics a monitor construction is developed which is optimal with respect to the space-complexity of the generated monitors as well as able to detect *all minimal bad prefixes* of non-conforming system behaviour. In the untimed case, where the monitor is generated from an LTL formula, the construction is based upon Büchi automata, whereas in the timed case, for TLTL, event-clock and their corresponding region automata are used. The timed case, in a sense, constitutes a change of paradigm, in that no longer quasi-synchronous sequences of events are observed and processed, but events whose occurrence is associated with a time-stamp. Therefore, quantitative assertions about events can be formulated and checked as compared to merely qualitative ones in the untimed case. The downside, however, lies in that the models of computation, i. e., that of an event-clock and region automaton, are much more involved as compared to the untimed case.

The presented approach to systems diagnosis transfers the principles of first-order model-based diagnosis to a propositional setup. In first-order diagnosis, explanations for an observed failure are generated by checking and restoring the consistency of first-order logic sentences which resemble a distributed or component-based system's components, their causality and behaviour as well as a set of corresponding observations. In the runtime reflection framework, the behaviour is monitored continuously, and diagnosis performed *only* if one or many monitors signal an aberration. By using *optimal size* monitors, which also detect minimal bad prefixes of behaviour, diagnosis can be mapped efficiently to a propositional satisfiability and counting problem, i. e., the so-called #SAT problem, thus also avoiding the issue of undecidability of first-order logic. Moreover, diagnosis must only be used in this setup when needed, instead of continuously.

Since in diagnosis of component-based systems, not all theoretically possible diagnoses are of practical interest (i. e., principle of parsimony, see §5), only those are determined where the answer set contains less or equal than $n \in \mathbb{N}$ faulty components, with n being referred to as the n-fault assumption. This is achieved in terms of a deterministic algorithmic solution for the #SAT problem, and by using the n-fault assumption as an optimisation heuristic. Accompanying experimental results of a tool prototype realising this idea, show that this optimisation yields considerable performance improvements over a non-optimised solution.

As an interface or a front end to the technical framework of runtime reflection, this thesis proposes Salt, the structured assertion language for temporal logic. Besides offering a number of high-level constructs for the expression of temporal properties, Salt has its formal foundations in LTL, respectively TLTL, when timed systems are to be specified. As such, Salt specifications can be translated either into monitors which, in turn, are built from LTL or TLTL formulae, or be used in combination with existing formal verification frameworks, such as model checking tools. First experimental results from a SALT compiler which realises the presented translation schemes show that the higher level of abstraction achieved by many of SALT's syntactic constructs, does not automatically result in bigger plain temporal logic formulae. In fact, the Büchi automata resulting from translated SALT formulae were in some cases smaller than their equivalent ones, if translated from a handwritten LTL formula. Although this result is not inherent to the SALT approach, it allows for some interesting conclusions: for example that there is still potential for improvement in the translation of plain LTL formulae to Büchi automata.

Finally, this thesis presents the algorithms and concepts that make up a technical infrastructure for runtime reflection. More so, in Appendix B, some pointers are given, where to obtain tool prototypes that have been released under an open source license and are based on these algorithms and concepts.

As such, this thesis presents both the theory and practice of runtime reflection.

7.2 Future research directions

Runtime reflection as presented in this thesis raises several fundamental questions as well as practical questions that concern, in particular, the applicability of the proposed framework. In the following, selected issues of both are highlighted, and potential for further research discussed.

7.2.1 Fundamental research questions

During the course of this thesis, various "design decisions" were made regarding the use of data structures, algorithms, and techniques to achieve the goals of runtime reflection. Although most of these design decisions are sketched in the appropriate text passages, others deserve a special mention at this point; firstly, because their detailed examination raises too many questions which cannot be answered without additional fundamental research, and secondly because this research is not directly in the scope of this thesis.

For instance, the decidability results for TLTL as given by Raskin and Schobbens [1999] (see also D'Souza [2003]) are an important reason for the choice of real-time temporal logic in this thesis. TLTL uses a so-called *point-wise semantics* with concrete time-stamps, whereas other real-time logics, such as MTL (cf. Ouaknine and Worrell [2005]) use *continuous semantics* based on continuous intervals of time. A lot of research especially in the 1990s was devoted to comparing the expressiveness of different real-time temporal logics, and some of these results have not shaped until more than 15 years after the introduction of a new real-time logic (cf. Alur and Henzinger [1991], Alur et al. [1996], Bouyer et al. [2005b]). Therefore, and not only in the context of this thesis, it would be interesting to determine how TLTL compares to (subsets) of other real-time logics, and whether or not, the construction of monitoring components would transfer over to these logics in a similar manner. In other words, which are the properties that are practically relevant, and go either beyond the properties expressible in TLTL, or can be captured more appropriately using a different real-time logic, such as a continuous one? Is it possible to construct a deterministic runtime monitor for such properties, which also detects minimal bad prefixes, and if so, for which classes of properties? Recall, the presented monitoring scheme is not restricted to safety properties alone.

Moreover, the monitor construction for TLTL is based upon the region graph construction of a timed automaton. However, the number of regions in a timed automaton is exponential both in the number of occurring clocks as well as in the maximal clock constants used in an automaton (see §4). In other words, the region graph construction yields practically large state spaces, which is the reason why, in timed model checking tools, a coarser representation of clock constraints is preferred. Such a representation exists in the form of clock zones as used by, e. g., Uppaal (Behrmann et al. [2002]). The number of zones, however, can still be equal to the number of regions in the worst case, and for some "rarely occurring" clock constraints even unbounded without subsequent normalisation (see §4). Additionally, it is not always possible to find a *stable* zone equivalence (cf. Bengtsson and Yi [2004]). Hence, it is not clear whether or not the presented construction of TLTL monitors could actually benefit from such a representation, and where the limitations are in terms of the clock constraints which can be expressed more efficiently by using zones. Recall, a monitor is but a finite state machine, hence, minimisation is possible and it remains to show that a minimised monitor, based on regions, corresponds to a (minimised) monitor based on a (normalised) zone graph construction, which exhibits stability.

7.2.2 Domain-specific aspects and applicability

Runtime reflection also raises a number of practical questions. For instance, it is worth investigating the *trade offs* that have to be made, when considering the use of this framework in resource or economically bound environments, such as embedded control, avionics or automotive systems. All mechanisms of fault tolerance, which, for the sake of argument, runtime reflection loosely classifies as, require redundancy, either in the hardware or the software of a system. Runtime reflection as presented in this thesis suggests the use of redundancy foremost in software in terms of the monitoring components and a centralised diagnoser component. However, even software redundancy comes at a price, e. g., in terms of additional CPU cycles or memory. More so, a heterogeneous scenario could also be thought of, where software *and* hardware monitors (such as "watchdogs") are used to diagnose root causes of an observed failure. Hence, for practical use, an interesting question lies in the *diagnosability* of systems. In other words, where should monitors be placed to detect costly, critical, or otherwise relevant patterns of failure, and how much does this redundancy cost? This thesis does not aim to give an answer to such questions. However, an empirical investigation based on scenarios where fault tolerance mechanisms are crucial and even mission critical, e. g., as in electronic avionics or automotive systems, could point to further important methodological aspects in the use and deployment of runtime reflection.

SALT, although only proposed as a front end, also deserves a deeper empirical analysis. One important point of interest lies in the correlation between the conciseness of a temporal logic specification and its correctness with respect to the real-world property it aims to capture. SALT offers various high-level constructs which help to make complicated temporal logic specifications more concise and, as this thesis argues, readable. However, it is not claimed that SALT *totally* lifts the burden of having to thoroughly understand the semantics of (linear time) temporal logic in order to transform (possibly informal) temporal requirements into a formal specification, even if the used formalism is closer to common, imperative programming languages. The productivity increase, or absolute reduction of errors in temporal logic specifications could be quantified precisely in order to identify potential for improvement, suggest additional operators or different levels of abstraction from the underlying plain temporal logic. Moreover, depending on the application domains where such studies would be carried out in, this could also allow for further insights regarding the suitability of SALT and temporal logic in general to capture domain specific artifacts in different stages of, say, a software and systems development life-cycle.

A step in this direction was made previously by Dwyer et al. [1999]. However, the focus of their studies was on identifying recurring *patterns* in existing real-world temporal logic specifications, such as scopes (see also §3.3.3), rather than identifying practical limitations of the employed formalisms. Broader studies with an emphasis on the correlation between the ability of a temporal logic to capture specific artifacts and the application domain it is used in have yet to be carried out.

Bibliography

ADAC (2005). ADAC-Pannenstatistik 2004. ADAC-Presseservice.

ADAC (2006). ADAC-Unfallstatistik 2005. ADAC-Presseservice.

Aho, A. V., Sethi, R. and Ullman, J. D. (1988). *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Alpern, B. and Schneider, F. B. (1984). Defining liveness. Tech. rep., Ithaca, NY, USA.

Alpern, B. and Schneider, F. B. (1987). Recognizing safety and liveness. *Distributed Computing*, **2** 117–126.

Alur, R. and Dill, D. L. (1990). Automata for modeling real-time systems. In *ICALP* (M. Paterson, ed.), vol. 443 of *Lecture Notes in Computer Science*. Springer-Verlag.

Alur, R. and Dill, D. L. (1996). Automata-theoretic verification of real-time systems. In *Formal Methods for RealTime Computing* (C. Heitmeyer and D. Mandrioli, eds.). Wiley, 55–82.

Alur, R., Feder, T. and Henzinger, T. A. (1996). The benefits of relaxing punctuality. *Journal of the ACM*, **43** 116–146.

Alur, R., Fix, L. and Henzinger, T. A. (1999). Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, **211** 253–273.

Alur, R. and Henzinger, T. (1991). Logics and Models of Real-Time: A Survey. In *Real Time: Theory in Practice*, vol. 600. Springer-Verlag.

Alur, R. and Henzinger, T. A. (1989). A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*.

Amálio, N. and Polack, F. (2003). Comparison of formalisation approaches of UML class constructs in Z and Object-Z. In *ZB* (D. Bert, J. P. Bowen, S. King and M. A. Waldén, eds.), vol. 2651 of *Lecture Notes in Computer Science*. Springer-Verlag.

Arafat, O. (2005). *A Tool for automated monitor code generation from LTL formulae*. Master's thesis, Institut für Informatik, Technische Universität München, Germany.

Arafat, O., Bauer, A., Leucker, M. and Schallhart, C. (2005). Runtime verification revisited. Tech. Rep. TUM-I0518, Institut für Informatik, Technische Universität München.

Armoni, R., Bustan, D., Kupferman, O. and Vardi, M. Y. (2003). Resets vs. aborts in linear temporal logic. vol. 2619 of *Lecture Notes in Computer Science*. Springer-Verlag.

Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M. Y. and Zbar, Y. (2002). The ForSpec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for Construction and Analysis of Systems* (J.-P. Katoen and P. Stevens, eds.), vol. 2280 of *Lecture Notes in Computer Science*. Springer-Verlag.

Avižienis, A. (1967). Design of fault-tolerant computers. In *Proceedings of the AFIPS Fall Joint Computer Conference* (D. C. Thompson, ed.), vol. 31. Washington.

Avižienis, A., Laprie, J.-C. and Randell, B. (2001). Fundamental concepts of dependability. Tech. Rep. 739, University of Newcastle upon Tyne, School of Computing Science.

Ball, T. and Rajamani, S. K. (2002). The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA.

Baroni, P., Lamperti, G., Pogliano, P. and Zanella, M. (1999). Diagnosis of large active systems. *Artificial Intelligence*, **110** 135–183.

Barringer, H., Goldberg, A., Havelund, K. and Sen, K. (2004). Rule-based runtime verification. In *VMCAI* (B. Steffen and G. Levi, eds.), vol. 2937 of *Lecture Notes in Computer Science*. Springer.

Bauer, A. (2004). Creating a portable programming language using open source software. In *Proceedings of the USENIX Annual Technical Conference*. FREENIX technical sessions, USENIX Association, Boston, MA.

Bauer, A. (2005). Simplifying diagnosis using LSAT: a propositional approach to reasoning from first principles. In *Proceedings of the 2005 International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, vol. 3524 of *Lecture Notes in Computer Science*. Springer-Verlag.

Bauer, A., Leucker, M. and Schallhart, C. (2006a). Model-based runtime analysis of distributed reactive systems. In *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society.

Bauer, A., Leucker, M. and Schallhart, C. (2006b). Monitoring of real-time properties. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, vol. 4337 of *Lecture Notes in Computer Science*. Springer-Verlag.

Bauer, A., Leucker, M. and Streit, J. (2006c). SALT—Structured Assertion Language for Temporal logic. Tech. Rep. TUM-I0604, Institut für Informatik, Technische Universität München.

Bauer, A., Leucker, M. and Streit, J. (2006d). SALT—Structured Assertion Language for Temporal logic. In *Proceedings of the Eighth International Conference on Formal Engineering Methods (ICFEM)*, vol. 4260 of *Lecture Notes in Computer Science*. Springer-Verlag.

Bauer, A., Pister, M. and Tautschnig, M. (2007). Tool-support for the analysis of hybrid systems and models. In *Proceedings of the 2007 Conference on Design, Automation and Test in Europe (DATE)*. IEEE Computer Society. Forthcoming.

Bauer, A., Romberg, J. and Schätz, B. (2005). Integrierte Entwicklung von Automotive-Software mit AutoFocus. *Informatik – Forschung und Entwicklung*, **19** 194–205.

Baumgartner, P., Fröhlich, P., Furbach, U. and Nejdl, W. (1996). Tableaux for Diagnosis Applications. Tech. Rep. 23–96, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz.

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BECK, K. and FOWLER, M. (2000). *Planning Extreme Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BEER, I., BEN-DAVID, S., EISNER, C., FISMAN, D., GRINGAUZE, A. and RODEH, Y. (2001). The temporal logic sugar. In Berry et al. [2001], 363–367.

BEHRMANN, G., BENGTSSON, J., DAVID, A., LARSEN, K. G., PETTERSSON, P. and YI, W. (2002). Uppaal implementation secrets. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, vol. 2469 of *Lecture Notes in Computer Science*. Springer-Verlag.

BENGTSSON, J. and YI, W. (2004). Timed automata: Semantics, algorithms and tools. In *In Lecture Notes on Concurrency and Petri Nets* (W. Reisig and G. Rozenberg, eds.), vol. 3098 of *Lecture Notes in Computer Science*. Springer-Verlag.

BENVENISTE, A. (2002). Non-massive, non-high performance, distributed computing: Selected issues. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*. Springer-Verlag, London, UK.

BENVENISTE, A., CASPI, P., EDWARDS, S., HALBWACHS, N., LE GUERNIC, P. and DE SIMONE, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*, **91**.

BERGE, C. (1989). *Hypergraphs: Combinatorics of Finite Sets.* North Holland, Amsterdam.

BERRY, G. (1999). The constructive semantics of pure Esterel. Draft book (version 3.0).

BERRY, G. (2000). *Proof, Language and Interaction: Essays in Honour of Robin Milner*, chap. The Foundations of Esterel. MIT Press.

BERRY, G., COMON, H. and FINKEL, A. (eds.) (2001). *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, vol. 2102 of *Lecture Notes in Computer Science*. Springer.

BEYS, P. and JANSEN, M. (1999). Automatic reuse of knowledge: A theory. In *Proceedings of the Twelfth Workshop on Knowledge Acquisition, Modeling and Management*. Alberta, Canada.

BIERE, A., CIMATTI, A., CLARKE, E., STRICHMAN, O. and ZHU, Y. (2003). *Advances in Computers*, chap. Bounded model checking. Academic Press.

BLOEM, R., GABOW, H. N. and SOMENZI, F. (2000). An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Springer-Verlag, London, UK.

BOEHM, B. W. (1981). *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, New Jersey.

BOEHM, B. W. and PAPACCIO, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, **14** 1462–1477.

BOOCH, G., RUMBAUGH, J. and JACOBSON, I. (1998). *The Unified Modeling Language User Guide*. Addison-Wesley.

BOUYER, P., CHEVALIER, F. and D'SOUZA, D. (2005a). Fault diagnosis using timed automata. In *FoSSaCS* (V. Sassone, ed.), vol. 3441 of *Lecture Notes in Computer Science*. Springer.

BOUYER, P., CHEVALIER, F. and MARKEY, N. (2005b). On the expressiveness of TPTL and MTL. In *Proceedings of the 25th Conference on Fundations of Software Technology and Theoretical Computer Science (FSTTCS'05)* (R. Ramanujam and S. Sen, eds.), vol. 3821 of *Lecture Notes in Computer Science*. Springer, Hyderabad, India.
URL `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/BCM-fsttcs05.pdf`

BRADFIELD, J. and STEVENS, P. (1998). Observational mu calculus. In *Proceedings of the Workshop on Fixed Points in Computer Science, FICS'98*. An extended version is available as BRICS-RS-99-5.

BRADFIELD, J. C., KÜSTER FILIPE, J. and STEVENS, P. (2002). Enriching OCL using observational mu-calculus. In *FASE* (R.-D. Kutsche and H. Weber, eds.), vol. 2306 of *Lecture Notes in Computer Science*. Springer.

BREITLING, M. (2001). *Formale Fehlermodellierung für verteilte reaktive Systeme*. Ph.D. thesis, Institut für Informatik der Technischen Universität München.

BROY, M. (1997). Requirements engineering for embedded systems. In *Proceedings of the Workshop on Formaler Entwurf sicherheitskritischer eingebetteter Systeme (FEmSys)*.

BROY, M. (1999). Software technology—formal methods and scientific foundations. *Information & Software Technology*, **41** 947–950.

Broy, M., Facchi, C., Grosu, R., Hettler, R., Hussmann, H., Nazareth, D., Slotosch, O., Regensburger, F. and Stølen, K. (1993). The requirement and design specification language Spectrum, an informal introduction (V 1.0), part 1 & 2. Tech. Rep. TUM-I9312, Technische Universität München.

Broy, M., Huber, F. and Schätz, B. (1999). AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik – Forschung und Entwicklung (IFE)* 121–134.

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M. and Pretschner, A. (eds.) (2005). *Model-based Testing of Reactive Systems*, vol. 3472 of *Lecture Notes in Computer Science*. Springer-Verlag.

Broy, M. and Rausch, A. (2005). Das neue V-Modell XT. *Informatik Spektrum*, **28** 220–229.

Broy, M. and Stølen, K. (2001). *Specification and development of interactive systems: Focus on streams, interfaces, and refinement*. Springer-Verlag, New York.

Büchi, J. R. (1962). On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method, and Philosophy of Science*. Stanford University Press, Stanford, CA, USA.

Büchi, M. and Weck, W. (1999). The greybox approach: When blackbox specifications hide too much. Tech. Rep. 297, Turku Center for Computer Science. Http://www.abo.fi/˜mbuechi/publications/TR297.html.

Burns, A. and Wellings, A. J. (2001). *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Bylander, T., Allemang, D., Tanner, M. C. and Josephson, J. R. (1991). The computational complexity of abduction. *Artificial Intelligence*, **49** 25–60.

Carter, W. C. (1979). Fault detection and recovery algorithms for fault-tolerant systems. In *Proceedings of the EURO IFIP'79 conference*.

Cassandras, C. (1993). *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, Homewood, IL.

Chong, E. K. P. (2000). Discrete-event systems and their optimization. In *Perspectives in Control Engineering: Technologies, Applications, and New Directions* (T. Samad, ed.). IEEE Press.

Chow, E. Y. and Willsky, A. S. (1984). Analytical redundancy and the design of robust failure detection systems. *IEEE Transactions on Automatic Control*, **7** 603–614.

Cimatti, A., Roveri, M. and Sheridan, D. (2004). Bounded verification of past LTL. In *Formal Methods in Computer-Aided Design; 5th International Conference, FMCAD 2004* (A. J. Hu and A. K. Martin, eds.). Lecture Notes in Computer Science, Springer Verlag, Austin, TX, USA.

Clarke, E., Kroening, D., Ouaknine, J. and Strichman, O. (2004). The completeness threshold for bounded model checking.

Clarke, E. M. and Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*. Springer-Verlag, London, UK.

Clarke, E. M., Grumberg, O. and Peled, D. A. (1999). *Model Checking*. The MIT Press, Cambridge, Massachusetts.

Clarke, E. M. and Schlingloff, H. (2001). Model checking. In *Handbook of Automated Reasoning* (A. Robinson and A. Voronkov, eds.), vol. II, chap. 24. Elsevier Science, 1635–1790.

Clarke, E. M., Wing, J. M., Alur, R., Cleaveland, R., Dill, D., Emerson, A., Garland, S., German, S., Guttag, J., Hall, A., Henzinger, T., Holzmann, G., Jones, C., Kurshan, R., Leveson, N., McMillan, K., Moore, J., Peled, D., Pnueli, A., Rushby, J., Shankar, N., Sifakis, J., Sistla, P., Steffen, B., Wolper, P., Woodcock, J. and Zave, P. (1996). Formal methods: state of the art and future directions. *ACM Computing Surveys*, **28** 626–643.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. and Talcott, C. (2003). The Maude 2.0 system. In *Rewriting Techniques and Applications (RTA 2003)* (R. Nieuwenhuis, ed.). No. 2706 in Lecture Notes in Computer Science, Springer-Verlag.

Clocksin, W. F. and Mellish, C. S. (1987). *Programming in Prolog*. 3rd ed. Springer-Verlag.

Colmerauer, A. and Roussel, P. (1993). The birth of prolog. In *HOPL Preprints*.

Console, L., Portinale, L. and Dupré, D. T. (1991). Focussing abductive diagnosis. *AI Commun.*, **4** 88–97.

COOK, S. A. (1971). The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*. ACM Press, New York, NY, USA.

COOK, S. A. and MITCHELL, D. G. (1997). Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications* (Du, Gu and Pardalos, eds.), vol. 35 of *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1–17.

CORBETT, J., DWYER, M., HATCLIFF, J. and ROBBY (2001). Expressing checkable properties of dynamic systems: The Bandera specification language. Tech. Rep. 04, Kansas State University, Department of Computing and Informatio Sciences.

CORDIER, M. O., DAGUE, P., DUMAS, M., LEVY, F., MONTMAIN, J., STAROSWIECKI, M. and TRAVE-MASSUYES, L. (2000). AI and automatic control approaches of model-based diagnosis: Links and underlying hypotheses. In *Proceedings of the IFAC Symposium SAFEPROCESS 2000*. Budapest, Hungary.

DAHL, O.-J., DIJKSTRA, E. W. and HOARE, C. A. R. (1972). *Structured Programming*. Academic Press, London, UK.

D'AMORIM, M. and ROSU, G. (2005). Efficient monitoring of omega-languages. In *CAV* (K. Etessami and S. K. Rajamani, eds.), vol. 3576 of *Lecture Notes in Computer Science*. Springer.

DAVIS, M., LOGEMANN, G. and LOVELAND, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, **5** 394–397.

DAVIS, M. and PUTNAM, H. (1960). A computing procedure for quantification theory. *Journal of the ACM*, **7** 201–215.

DE KLEER, J. (1986). Problem solving with the ATMS. *Artificial Intelligence*, **28** 197–224.

DE KLEER, J. and KURIEN, J. (2003). Fundamentals of model-based diagnosis. In *International Federation of Automatic Control (IFAC): Safeprocess 2003* (M. Staroswiecki, ed.). Washington.

DE KLEER, J. and WILLIAMS, B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, **32** 97–130.

DE KLEER, J. and WILLIAMS, B. C. (1992). Diagnosis with behavioral modes 124–130.

Delgado Kloos, C. and Damm, W. (eds.) (1997). *Practical Formal Methods for Hardware Design. Esprit Project 6128: Format, Vol 1*. Research Reports Esprit.

Deur, J., Pavković, D., Jansz, M. and Perić, N. (2003). Automatic tuning of electronic throttle control strategy. In *Proceedings of 2003 Mediterranean Conference on Control and Automation (MED 2003)*. IEEE Computer Society, Rhodes, Greece.

Dewhurst, S. (2002). *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Dill, D. L. (1989). Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems* (J. Sifakis, ed.), vol. 407 of *Lecture Notes in Computer Science*. Springer-Verlag.

Dill, D. L. and Rushby, J. (1996). Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, **29** 23–24.

Dröschel, W. and Wiemers, M. (2000). *Das V-Modell 97*. Oldenbourg.

Drusinsky, D. (2000). The temporal rover and the ATG rover. In *SPIN*.
URL `citeseer.ist.psu.edu/drusinsky00temporal.html`

D'Souza, D. (2003). A logical characterisation of event clock automata. *International Journal of Foundations of Computer Science (IJFCS)*, **14** 625–639.

Dustin, E., Rashka, J. and Paul, J. (1999). *Automated software testing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Dwyer, M. B., Avrunin, G. S. and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA.

Ebeling, C. E. (1997). *An Introduction to Reliability and Maintainability Engineering*. McGraw-Hill Companies, Inc.

Edwards, S., Lavagno, L., Lee, E. A. and Sangiovanni-Vincentelli, A. (1997). Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, **85** 366–390.

Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A. and Campenhout, D. V. (2003). Reasoning with temporal logic on truncated paths. In

CAV (W. A. H. Jr. and F. Somenzi, eds.), vol. 2725 of *Lecture Notes in Computer Science*. Springer.

Eiter, T. and Gottlob, G. (1995). Identifying the minimal transversals of a hypergraph and related problems. *SIAM J. Comput.*, **24** 1278–1304.

Emerson, E. A. and Clarke, E. M. (1980). Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*. Springer-Verlag, London, UK.

Emerson, E. A. and Halpern, J. Y. (1982). Decision procedures and expressiveness in the temporal logic of branching time. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*. ACM Press, New York, NY, USA.

Emerson, E. A. and Lei, C.-L. (1985). Modalities for model checking: Branching time strikes back. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. ACM SIGACT-SIGPLAN, ACM Press, New Orleans, Louisiana. Extended abstract.

Farwer, B. (2001). Omega-automata. In *Automata, Logics, and Infinite Games* (E. Grädel, W. Thomas and T. Wilke, eds.), vol. 2500 of *Lecture Notes in Computer Science*. Springer.

Finkel, O. (2003). Borel hierarchy and omega context free languages. *Theor. Comput. Sci.*, **290** 1385–1405.

Finkel, O. and Simonnet, P. (2003). Topology and ambiguity in omega context free languages. *Bulletin of the Belgian Mathematical Society*, **10** 707–722.

Fisher, M. (1991). A resolution method for temporal logic. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*. Sydney, Australia.

Fitting, M. (1996). *First-Order Logic and Automated Theorem Proving*. 2nd ed. Springer-Verlag.

Flach, P. (1994). *Simply Logical: intelligent reasoning by example*. John Wiley & Sons Ltd., Sussex, UK.

Foster, H., Marschner, E. and Wolfsthal, Y. (2005). IEEE 1850 PSL: The next generation. In *DVCon*.

France, R., Evans, A., Lano, K. and Rumpe, B. (1998). The UML as a formal modeling notation. *Comput. Stand. Interfaces*, **19** 325–334.

Free Software Foundation (1991). GNU General Public License. `http://www.fsf.org/licenses/gpl.html`, Free Software Foundation, Inc., Cambridge, Massachusetts.

Fritz, C. (2003). Constructing Büchi automata from linear temporal logic using simulation relations for alternating büchi automata. In *CIAA* (O. H. Ibarra and Z. Dang, eds.), vol. 2759 of *Lecture Notes in Computer Science*. Springer.

Gabbay, D. M. (1989). The declarative past and imperative future: Executable temporal logic for interactive systems. In *Temporal Logic in Specification*. Springer-Verlag.

Gabbay, D. M., Hogger, C. J., Robinson, J. A. and Siekmann, J. H. (eds.) (1994). *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume2, Deduction Methodologies*. Oxford University Press.

Gallo, G. and Scutella, M. G. (1988). Polynomially solvable satisfiability problems. *Inf. Process. Lett.*, **29** 221–227.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gärtner, F. C. (2001). *Formale Grundlagen der Fehlertoleranz in verteilten Systemen*. Ph.D. thesis.

Gastin, P. and Oddoux, D. (2001). Fast ltl to büchi automata translation. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*. Springer-Verlag, London, UK.

Geilen, M. (2001). On the construction of monitors for temporal logic properties. *Electr. Notes Theor. Comput. Sci.*, **55**.

Gerdsmeier, T., Ladkin, P. B. and Loer, K. (1997). Formalising failure analysis. Tech. Rep. RVS-Occ-97-06, Bielefeld University, Faculty of Technology.

Giannakopoulou, D. and Havelund, K. (2001). Runtime analysis of linear temporal logic specifications. Tech. Rep. 01.21, RIACS/USRA.

Gilb, T. (1988). The pre-natal death of the cis project: A software disaster story. *Journal of Systems and Software*, **8** 161–163.

Greenwell, W. and Knight, J. C. (2003). What should aviation safety incidents teach us? Tech. Rep. CS-2003-12, University of Virginia, Department of Computer Science.

Gries, D. (1982). A note on the standard strategy for developing loop invariants and loops. Tech. rep., Ithaca, NY, USA.

Gunter, D., Tierney, B., Jackson, K. R., Lee, J. and Stoufer, M. (2002). Dynamic monitoring of high-performance distributed applications. In *HPDC*. IEEE Computer Society.

Håkansson, J., Jonsson, B. and Lundqvist, O. (2003). Generating online test oracles from temporal logic specifications. *Journal on Software Tools for Technology Transfer*, **4** 456–471.

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

Halbwachs, N., Lagnier, F. and Raymond, P. (1994). Synchronous observers and the verification of reactive systems. In *AMAST '93: Proceedings of the Third International Conference on Methodology and Software Technology*. Springer-Verlag, London, UK.

Harel, D. and Pnueli, A. (1985). On the development of reactive systems. In *Logics and models of concurrent systems*. Springer, New York, NY, USA, 477–498.

Harrison, J. (TBA). *Introduction to logic and automated theorem proving*. Cambridge University Press, Cambridge, UK.

Havelund, K. (1999). Java pathfinder, a translator from java to promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. Springer-Verlag, London, UK.

Havelund, K. (2000). Using runtime analysis to guide model checking of java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. Springer-Verlag, London, UK.

Havelund, K. and Goldberg, A. (2005). Verify your runs. In *Proceedings of the Grand Verification Challenge Workshop 'Verified Software: Theories, Tools, Experiments'*. Zürich, Switzerland.

HAVELUND, K. and ROSU, G. (2001a). Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, **55**.

HAVELUND, K. and ROSU, G. (2001b). Monitoring programs using rewriting. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*. IEEE Computer Society, Washington, DC, USA.

HAVELUND, K. and ROSU, G. (2002). Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems*.

HAVELUND, K. and ROSU, G. (2004). Efficient monitoring of safety properties. *Journal on Software Tools for Technology Transfer*.

HEINECKE, H. (2005). Automotive system design - challenges and potential. In *DATE*. IEEE Computer Society.

HEINECKE, H., SCHNELLE, K.-P., BORTOLAZZI, J., LUNDH, L., LEFLOUR, J., MATE, J.-L., NISHIKAWA, K. and SCHARNHORST, T. (2004). AUTomotive Open System ARchitecture—An industry-wide initiative to manage the complexity of emerging automotive E/E-architectures. In *Proceedings of the SAE 2004 World Congress*. Society of Automotive Engineers, Detroit, MI.

HOARE, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, **12** 576–580.

HOLZMANN, G. J. (1991). *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

HOPCROFT, J. E. and ULLMAN, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. 1st ed. Addison-Wesley.

HUBER, F., SCHÄTZ, B., SCHMIDT, A. and SPIES, K. (1996). AUTOFOCUS: A Tool for Distributed Systems Specification. In *Proceedings FTRTFT'96 — Formal Techniques in Real-Time and Fault-Tolerant Systems*.

IEEE (1995). *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*.
URL `http://www.ansi.org/`

INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING (1994). Dependability: Basic concepts and terminology. Tech. rep.

Janssen, G. L. J. M. (1989). Hardware verification using temporal logic: A practical view. In *Proceedings of the International Workshop on Applied Formal Methods for correct VLSI Design*. Leuwen.

Jones, C. (1998). *Estimating Software Costs*. McGraw-Hill, New York.

Kamp, J. A. W. (1968). *Tense Logic and the Theory of Linear Order*. Ph.D. thesis, University of California, Los Angeles.

Khachiyan, L., Boros, E., Elbassioni, K. M. and Gurvich, V. (2005). A new algorithm for the hypergraph transversal problem. In *Proceedings of the Eleventh International Computing and Combinatorics Conference* (L. Wang, ed.), vol. 3595 of *Lecture Notes in Computer Science*. Springer.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In *Automata Studies* (C. E. Shannon and J. McCarthy, eds.). Princeton University Press, Princeton, New Jersey, 3–41.

Knight, J. C. (2002). Software challenges in aviation systems. In *Proceedings of the 21st International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, vol. 2434 of *Lecture Notes in Computer Science*. Springer-Verlag, London, UK.

Knuth, D. E. (1998). *The Art of Computer Programming: Sorting and Searching*, vol. 3. 2nd ed. Addison-Wesley.

Köhl, S., Stroop, J., Riedesser, P. and Peller, M. (2005). Testing FlexRay ECUs with a hardware-in-the-loop simulator. *VDI – Mess- und Versuchstechnik in der Fahrzeugentwicklung*.

Kowalski, R. and Hayes, P. J. (1969). Semantic trees in automatic theorem proving. In *Machine Intelligence*, vol. 4. Edinburgh University Press, 87–101.

Kozen, D. (1990). On kleene algebras and closed semirings. In *MFCS '90: Proceedings of the Mathematical Foundations of Computer Science 1990*. Springer-Verlag, London, UK.

Kristoffersen, K. J., Pedersen, C. and Andersen, H. R. (2003). Runtime verification of timed LTL using disjunctive normalized equation systems. *Electronic Notes in Theoretical Computer Science*, **89**.

Kupferman, O. and Vardi, M. Y. (2001). Model checking of safety properties. *Form. Methods Syst. Des.*, **19** 291–314.

Ladkin, P. (1997). The success and failure of complex artifacts. Tech. Rep. RVS-Bk-01, Bielefeld University, Faculty of Technology.

Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, **3** 125–143.

Lamport, L. (1983). What good is temporal logic? In *Proceedings of the IFIP 9th World Computer Congress* (R. E. A. Mason, ed.). North-Holland/IFIP.

Laprie, J.-C. (ed.) (1992). *Dependability: Basic concepts and Terminology*, vol. 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag.

Latvala, T. (2002). On model checking safety properties. Research Report A76, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland.

Laurent, O., Michel, P. and Wiels, V. (2001). Using formal verification techniques to reduce simulation and test effort. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*. Springer-Verlag, London, UK.

Lebow, C. C., Sarsfield, L. P., Stanley, W., Ettedgui, E. and Henning, G. (2000). *Safety in the skies: Personnel and Parties in NTSB Aviation Accident Investigations*. No. MR-1122-ICJ in Monographs/Reports, RAND Institute for Civil Justice, Santa Monica, CA. Accessed through: `http://www.rand.org/pubs/monograph_reports/MR1122/`.

Lee, P. A. and Anderson, T. (1990). *Fault Tolerance: Principles and Practice*. 2nd ed. Dependable computing and fault-tolerant systems, Springer-Verlag, Berlin.

Lehman, M. M. (2005). The role and impact of assumptions in software development, maintenance and evolution. In *Proceedings of the IEEE International Workshop on Software Evolvability*. IEEE Computer Society.

Lehman, M. M. and Parr, F. N. (1976). Program evolution and its impact on software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM.

Leveson, N. G. (2004). The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets*, **41** 564–575.

Lichtenstein, O. and Pnueli, A. (1985). Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

LIFSCHITZ, V. (1995). The logic of common sense. *ACM Comput. Surv.*, **27** 343–345.

MACIASZEK, L. A. (2001). *Requirements analysis and system design: developing information systems with UML*. Addison-Wesley Longman Ltd., Essex, UK.

MALER, O., NICKOVIC, D. and PNUELI, A. (2005). Real time temporal logic: Past, present, future. In *FORMATS* (P. Pettersson and W. Yi, eds.), vol. 3829 of *Lecture Notes in Computer Science*. Springer-Verlag.

MANN, C. C. (2002). Why software is so bad. MIT Technology Review.

MARKEY, N. (2003). Temporal logic with past is exponentially more succinct, concurrency column. *Bulletin of the EATCS*, **79** 122–128.

MATHWORKS (2000). *Using Simulink*. The MathWorks Inc.

MCCARTHY, J. (1987). Applications of circumscription to formalizing common-sense knowledge 153–166.

MCCURDY, H. E. (2001). *Faster, better, cheaper*. The Johns Hopkins University Press.

MCMILLAN, K. L. (1992). *Symbolic model checking: an approach to the state explosion problem*. Ph.D. thesis, Pittsburgh, PA, USA.

MCNAUGHTON, R. (1966). Testing and generating infinite sequences by a finite automaton. *Information and Control*, **9** 521–530.

MICHEL, M. (1988). Complementation is more difficult with automata on infinite words. CNET, Paris.

MIKAELIAN, T., WILLIAMS, B. C. and SACHENBACHER, M. (2005). Diagnosing complex systems with software-extended behavior using constraint optimization. In *Proceedings of the 16th International Workshop on Principles of Diagnosis (DX-05)*. Monterey, CA.

MÖLLER, M. O. (2002). Structure and hierarchy in real-time systems. Tech. Rep. DS-02-1, BRICS, Department of Computer Science.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, **38** 114–117.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. and Malik, S. (2001). Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*. ACM Press, New York, NY, USA.

Musa, J. D., Iannino, A. and Okumoto, K. (1987). *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill Book Company, Whippany, NJ.

NASA (1996). Ariane 5—flight 501 failure report by the inquiry board. Tech. rep., National Aeronautics and Space Administration. Accessed through: `ftp://ftp.hq.nasa.gov/`.

NASA (1999). Mars climate orbiter mishap investigation board phase I report. Tech. rep., National Aeronautics and Space Administration. Accessed through: `ftp://ftp.hq.nasa.gov/`.

Neumann, P. G. (1995). *Computer-Related Risks*. Addison Wesley.

Nonnengart, A. and Weidenbach, C. (2001). Computing small clause normal forms. In *Handbook of Automated Reasoning* (A. Robinson and A. Voronkov, eds.), vol. I, chap. 6. Elsevier Science B.V., 335–367.

Nyberg, M. and Krysander, M. (2003). Combining AI, FDI, and statistical hypothesis-testing in a framework for diagnosis. In *Proceedings of IFAC Safeprocess'03*. Washington, USA.

Ouaknine, J. and Worrell, J. (2005). On the decidability of metric temporal logic. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE Computer Society, Washington, DC, USA.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley, New York.

Peyton Jones, S. (2005). Haskell 98 language and libraries. The revised report. `http://www.haskell.org/`.

Pham, H. (ed.) (2003). *Handbook of Reliability Engineering*. Springer-Verlag, London, UK.

Pike, L., Miner, P. and Torres, W. (2004). Model checking failed conjectures in theorem proving: a case study. Tech. Rep. NASA/TM–2004–213278, NASA Langley Research Center. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/unproven.html`.

PNUELI, A. (1977). The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*. IEEE Computer Society Press, Providence, Rhode Island.

PNUELI, A. (1986). Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current trends in concurrency. Overviews and tutorials*. Springer-Verlag, New York, NY, USA.

POOLE, D. (1988). A logical framework for default reasoning. **36** 27–47.

POOLE, D. (1994a). Default logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2: Nonmonotonic Reasoning* (D. Gabbay, C. Hogger and J. Robinson, eds.). Oxford University Press, Oxford.

POOLE, D. (1994b). Representing diagnosis knowledge. *Ann. Math. Artif. Intell.*, **11** 33–50.

POOLE, D. L. (1989). Normality and Faults in Logic-Based Diagnosis. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. Detroit.

POWELL, D. (1995). Failure mode assumptions and assumption coverage. Tech. Rep. 91462, LAAS/CNRS, Toulouse, France.

PRETSCHNER, A. (2003). *Zum modellbasierten funktionalen Test reaktiver Systeme*. Ph.D. thesis, Institut für Informatik der Technischen Universtität München.

PROCTER, P. (ed.) (1995). *Longman Dictionary of Contemporary English*. Longman Group Ltd, Harlow.

QUEILLE, J. and SIFAKIS, J. (1982). Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, vol. 137 of *Lecture Notes in Computer Science*. Springer-Verlag, New York.

RASKIN, J.-F. (1999). Logics, automata and classical theories for deciding real time.

RASKIN, J.-F. and SCHOBBENS, P.-Y. (1997). State clock logic: A decidable real-time logic. In *HART* (O. Maler, ed.), vol. 1201 of *Lecture Notes in Computer Science*. Springer.

RASKIN, J.-F. and SCHOBBENS, P.-Y. (1999). The logic of event clocks—decidability, complexity and expressiveness. *Journal of Automata, Languages, and Combinatorics*, **4** 247–286.

REITER, R. (1977). On closed world data bases. In *Logic and Data Bases*.

REITER, R. (1980). A logic for default reasoning. *Artificial Intelligence*, **13** 81–132.

REITER, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, **32** 57–95.

REITER, R. and DE KLEER, J. (1987). Foundations of assumption-based truth maintenance systems. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-87)*, vol. 1. MIT Press.

ROMBERG, J. and BAUER, A. (2004). Loose synchronization of event-triggered networks for distribution of synchronous programs. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT)*. Association for Computing Machinery, New York, NY.

ROTH, D. (1996). On the hardness of approximate reasoning. *Artif. Intell.*, **82** 273–302.

RUSHBY, J. (1993). Formal methods and the certification of critical systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA. Also issued under the title "Formal Methods and Digital Systems Validation for Airborne Systems" as NASA Contractor Report 4551, December 1993.

SAFRA, S. (1988). On the complexity of omega-automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science, FoCS'88*. IEEE Computer Society Press, Los Alamitos, California.

SAMPATH, M., SENGUPTA, R., LAFORTUNE, S., SINNAMOHIDEEN, K. and TENEKETZIS, D. (1994). Failure diagnosis using discrete event models. In *Proceedings of 33rd IEEE Conference on Decision and Control*. IEEE, New York, NY.

SAMULOWITZ, H. and BACCHUS, F. (2005). Using SAT in QBF. In *CP* (P. van Beek, ed.), vol. 3709 of *Lecture Notes in Computer Science*. Springer.

SARSFIELD, L. P., STANLEY, W., LEBOW, C. C., ETTEDGUI, E. and HENNING, G. (2000). *Safety in the skies: Personnel and Parties in NTSB Aviation Accident Investigations—Master Volume.* No. MR-1122/1-ICJ in Monographs/Reports, RAND Institute for Civil Justice, Santa Monica, CA. Accessed through: http://www.rand.org/pubs/monograph_reports/MR1122.1/.

Schätz, B., Fleischmann, A., Geisberger, E. and Pister, M. (2005). Model-based requirements engineering with AutoRAID. In *GI Jahrestagung (2)* (A. B. Cremers, R. Manthey, P. Martini and V. Steinhage, eds.), vol. 68 of *LNI*. GI.

Schneider, F. B. (1987). Decomposing properties into safety and liveness using predicate logic. Tech. Rep. 87-874, Cornell University Computer Science Department.

Schnoebelen, P. (2002). The complexity of temporal logic model checking. In *Advances in Modal Logic* (P. Balbiani, N.-Y. Suzuki, F. Wolter and M. Zakharyaschev, eds.). King's College Publications.

Schumann, A., Pencolé, Y. and Thiébaux, S. (2004). Diagnosis of discrete-event systems using binary decision diagrams. In *Proceedings of the Fifteenth International Workshop on Principles of Diagnosis (DX'04)*.

Sheridan, D. (2002). Using fixpoint characterisations of LTL for bounded model checking. Tech. Rep. APES-41-2002, APES Research Group.

Sistla, A. P. (1994). Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, **6** 495–512.

Sterman, J. D. (2002). All models are wrong: Reflections on becoming a systems scientist. *System Dynamics Review*, **18** 500–531.

Streit, J. (2006). *Development of a programming-language-like temporal logic specification language*. Master's thesis, Institut für Informatik, Technische Universität München, Germany.

Stroustrup, B. (2000). *The C++ Programming Language*. Special ed. Addison-Wesley, Boston, MA, USA.

Struss, P. and Heller, U. (2001). G+DE—the generalized diagnosis engine. In *Proceedings of the 12th international workshop on the principles of diagnosis (DX01)*.

Struss, P. and Malik, A. (1997). Automated diagnosis of car-subsystems based on qualitative models. In *XPS*.

STRUSS, P. and PRICE, C. (2003). Model-based systems in the automotive industry. *AI Magazine*, **24** 17–34.

THOMAS, W. (1990). Automata on infinite objects. In *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. B, chap. 4. Elsevier Science Publishers B. V., 133–191.

THOMPSON, K. (1968). Regular expression search algorithm. *Communications of the ACM*, **11** 419–422.

TRIPAKIS, S. and YOVINE, S. (2001). Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, **18** 25–68.

TUERK, T. and SCHNEIDER, K. (2005). From PSL to LTL: A formal validation in HOL. In *TPHOLs* (J. Hurd and T. F. Melham, eds.), vol. 3603 of *Lecture Notes in Computer Science*. Springer.

TURING, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, **42** 230–265.

VADHAN, S. P. (2001). The complexity of counting in sparse, regular, and planar graphs. *SIAM J. Compu.*, **31** 398–427.

VALIANT, L. G. (1979). The complexity of computing the permanent. *Theor. Comput. Sci.*, **8** 189–201.

VARCHMIN, J.-U. (ed.) (2005). *Diagnose von E/E-Systemen im Automobil.* EuroForum, Düsseldorf, Germany.

VARDI, M. Y. and WOLPER, P. (1986). An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS'86).* IEEE Computer Society Press, Washington, D.C., USA.

WADLER, P. (1992). The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press, New York, NY, USA.

WARMER, J. and KLEPPE, A. (1998). *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley.

WESTHEAD, M. and NADJM-TEHRANI, S. (1996). Verification of embedded systems using synchronous observers. In *FTRTFT '96: Proceedings of the 4th Interna-*

tional Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems. Springer-Verlag, London, UK.

ZELLER, A. (2005). *Why programs fail—A guide to systematic debugging.* Morgan Kaufmann.

ZHANG, H. (1997). Sato: An efficient propositional prover. In *CADE-14: Proceedings of the 14th International Conference on Automated Deduction.* Springer-Verlag, London, UK.

Appendix A

SALT translation schema

THIS APPENDIX, which resembles in parts the one presented by Streit [2006], describes how the SALT language (§3.3) is translated into LTL and TLTL and thereby defines the formal semantics of SALT.

The translation of past operators is left out for brevity, unless stated otherwise. It follows the same schema as the translation of the future operators. The translation of timed operators is described in section A.6. The other sections of this appendix refer to untimed SALT.

The translation is done in several phases:

- Expansion of user-defined macros.
- Replacement of non-core SALT operators. Several SALT operators are replaced by expressions made out of a small set of core operators.
- Translation of core SALT into RLTL. The SALT operators are replaced by RLTL expressions. RLTL includes all LTL operators as well as the acc and rej operators (corresponding to the SALT exception operators `accepton` and `rejecton`) and the exclusive and inclusive `stop` operators for future and past (introduced during the translation of `upto` and `between`).
- Translation of RLTL into LTL/TLTL. The translation of the RLTL operators requires weaving their end conditions into the whole sub-expression.
- Optimisation. The LTL/TLTL expression is optimised using a number of optimisation patterns.
- LTL/TLTL output. The LTL/TLTL expression is printed in the desired output syntax. This might require expressing certain operators through others (like $\mathbf{U}_w$ through $\mathbf{U}$). Also, extended TLTL operators may be replaced by pure TLTL.

Each translation step is described in form of a translation function $\mathcal{T}(\varphi)$ that is applied by choosing the first translation that matches the current expression. Trivial translations that just descend recursively into the arguments of an operator, such as $\mathcal{T}(\varphi \wedge \psi) = \mathcal{T}(\varphi) \wedge \mathcal{T}(\psi)$, are left out in the following.

The LTL operators used during translation are:

true	$true$	until	$\mathbf{U}$	since	$\mathbf{S}$
false	$false$	weak until	$\mathbf{U}_w$	back to	$\mathbf{S}_w$
logical negation	$\neg$	globally	$\mathbf{G}$	historically	$\mathbf{H}$
logical and	$\wedge$	eventually	$\mathbf{F}$	once	$\mathbf{O}$
logical or	$\vee$	next	$\mathbf{X}$	previous	$\mathbf{Y}$
logical implication	$\Rightarrow$	weak next	$\mathbf{X}_w$	weak previous	$\mathbf{Y}_w$
logical equivalence	$\Leftrightarrow$				

And for timed expressions additionally:

timed until	$\mathbf{U}_{\sim c}$	timed since	$\mathbf{S}_{\sim c}$
timed weak until	$\mathbf{U}_{w\sim c}$	timed weak since	$\mathbf{S}_{w\sim c}$
timed globally	$\mathbf{G}_{\sim c}$	timed historically	$\mathbf{H}_{\sim c}$
timed eventually	$\mathbf{F}_{\sim c}$	timed once	$\mathbf{O}_{\sim c}$
event predicting	$\rhd_{\sim c}$	event recording	$\lhd_{\sim c}$

The RLTL operators used are:

accept	acc
reject	rej
exclusive stop	$\text{stop}_{\text{excl}}$
inclusive stop	$\text{stop}_{\text{incl}}$

A.1 Replacement of non-core Salt operators

A.1.1 `never`

$$\mathcal{T}(\texttt{never}\ \varphi) \quad = \quad \neg\mathbf{F}\mathcal{T}(\varphi)$$

A.1.2 `releases`

$$\mathcal{T}(\varphi\ \texttt{releases}\ \psi) \quad = \quad \mathcal{T}(\psi\ \texttt{until incl weak}\ \varphi)$$

A.1.3 `nextn`

$$\begin{array}{lll} \mathcal{T}(\texttt{nextn[=}n\texttt{]}\,\varphi) & & = \\ & \text{if } n = 0: & \mathcal{T}(\varphi) \\ & \text{else:} & \mathbf{X}\mathcal{T}(\texttt{nextn[=}n-1\texttt{]}\,\varphi) \end{array}$$

$$
\begin{array}{lll}
\mathcal{T}(\texttt{nextn[}n\texttt{..}m\texttt{]}\varphi) & = & \mathcal{T}(\texttt{nextn[=}n\texttt{]}(\texttt{nextn[<=}m-n\texttt{]}\varphi)) \\
\mathcal{T}(\texttt{nextn[<=}n\texttt{]}\varphi) & = & \\
\qquad\qquad \text{if } n=0\text{:} & & \mathcal{T}(\varphi) \\
\qquad\qquad \text{else:} & & \varphi \vee \mathbf{X}\mathcal{T}(\texttt{nextn[<=}n-1\texttt{]}\varphi) \\
\mathcal{T}(\texttt{nextn[<}n\texttt{]}\varphi) & = & \mathcal{T}(\texttt{nextn[<=}n-1\texttt{]}\varphi) \\
\mathcal{T}(\texttt{nextn[>=}n\texttt{]}\varphi) & = & \mathcal{T}(\texttt{nextn[=}n\texttt{]}\mathbf{F}\varphi) \\
\mathcal{T}(\texttt{nextn[>}n\texttt{]}\varphi) & = & \mathcal{T}(\texttt{nextn[>=}n+1\texttt{]}\varphi)
\end{array}
$$

A.1.4 `occurring`

$$
\begin{array}{lll}
\mathcal{T}(\texttt{occurring[=}n\texttt{]}\varphi) & = & \\
\qquad\qquad \text{if } n=0\text{:} & & \neg\mathbf{F}\mathcal{T}(\varphi) \\
\qquad\qquad \text{if } n=1\text{:} & & \neg\mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi) \wedge ((\mathcal{T}(\varphi)\ \mathbf{U}_w\ \neg\mathbf{F}\mathcal{T}(\varphi)))^{1} \\
\qquad\qquad \text{else:} & & \neg\mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi) \wedge (\mathcal{T}(\varphi)\ \mathbf{U}\ (\neg\mathcal{T}(\varphi) \wedge \\
 & & \mathcal{T}(\texttt{occurring[=}n-1\texttt{]}\varphi)))) \\
\mathcal{T}(\texttt{occurring[}n\texttt{..}m\texttt{]}\varphi) & = & \\
\qquad\qquad \text{if } n=0\text{:} & & \mathcal{T}(\texttt{occurring[<=}m\texttt{]}\varphi) \\
\qquad\qquad \text{if } n=1\text{:} & & \neg\mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi) \wedge (\mathcal{T}(\varphi)\ \mathbf{U}_w\ (\neg\mathcal{T}(\varphi) \wedge \\
 & & \mathcal{T}(\texttt{occurring[<=}m-1\texttt{]}\varphi))))^{1} \\
\qquad\qquad \text{else:} & & \neg\mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi) \wedge (\mathcal{T}(\varphi)\ \mathbf{U}\ (\neg\mathcal{T}(\varphi) \wedge \\
 & & \mathcal{T}(\texttt{occurring[}n-1\texttt{..}m-1\texttt{]}\varphi)))) \\
\mathcal{T}(\texttt{occurring[<=}n\texttt{]}\varphi) & = & \neg\mathcal{T}(\texttt{occurring[>=}n+1\texttt{]}\varphi) \\
\mathcal{T}(\texttt{occurring[<}n\texttt{]}\varphi) & = & \neg\mathcal{T}(\texttt{occurring[>=}n\texttt{]}\varphi) \\
\mathcal{T}(\texttt{occurring[>=}n\texttt{]}\varphi) & = & \\
\qquad\qquad \text{if } n=0\text{:} & & \mathit{true} \\
\qquad\qquad \text{if } n=1\text{:} & & \mathbf{F}\mathcal{T}(\varphi) \\
\qquad\qquad \text{else:} & & \mathbf{F}(\mathcal{T}(\varphi) \wedge (\mathbf{F}(\neg\mathcal{T}(\varphi) \wedge \\
 & & \mathcal{T}(\texttt{occurring[>=}n-1\texttt{]}\varphi)))) \\
\mathcal{T}(\texttt{occurring[>}n\texttt{]}\varphi) & = & \mathcal{T}(\texttt{occurring[>=}n+1\texttt{]}\varphi)
\end{array}
$$

A.1.5 `holding`

[1]Notice that the last occurrence of φ may last forever.

$$
\begin{array}{lll}
\mathcal{T}(\texttt{holding[=}n\texttt{]}\,\varphi) & = & \\
\qquad\text{if } n = 0: & & \neg\mathbf{F}\mathcal{T}(\varphi) \\
\qquad\text{if } n = 1: & & \neg\mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi) \wedge \mathbf{X}_w\neg\mathbf{F}\mathcal{T}(\varphi)))^2 \\
\qquad\text{else:} & & \neg\mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi) \wedge \mathbf{X}\mathcal{T}(\texttt{holding[=}n-1\texttt{]}\,\varphi))
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{T}(\texttt{holding[}n\texttt{..}m\texttt{]}\,\varphi) & = & \\
\qquad\text{if } n = 0: & & \mathcal{T}(\texttt{holding[<=}m\texttt{]}\,\varphi) \\
\qquad\text{if } n = 1: & & \neg\mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi)\wedge \\
 & & \mathbf{X}_w\mathcal{T}(\texttt{holding[<=}m-1\texttt{]}\,\varphi))^2 \\
\qquad\text{else:} & & \neg\mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi)\wedge \\
 & & \mathbf{X}\mathcal{T}(\texttt{holding[}n-1\texttt{..}m-1\texttt{]}\,\varphi))
\end{array}
$$

$$\mathcal{T}(\texttt{holding[<=}n\texttt{]}\,\varphi) = \neg\mathcal{T}(\texttt{holding[>=}n+1\texttt{]}\,\varphi)$$

$$\mathcal{T}(\texttt{holding[<}n\texttt{]}\,\varphi) = \neg\mathcal{T}(\texttt{holding[>=}n\texttt{]}\,\varphi)$$

$$
\begin{array}{lll}
\mathcal{T}(\texttt{holding[>=}n\texttt{]}\,\varphi) & = & \\
\qquad\text{if } n = 0: & & \mathit{true} \\
\qquad\text{if } n = 1: & & \mathbf{F}\mathcal{T}(\varphi)^2 \\
\qquad\text{else:} & & \mathbf{F}(\mathcal{T}(\varphi) \wedge \mathbf{X}\mathcal{T}(\texttt{holding[>=}n-1\texttt{]}\,\varphi))
\end{array}
$$

$$\mathcal{T}(\texttt{holding[>}n\texttt{]}\,\varphi) = \mathcal{T}(\texttt{holding[>=}n+1\texttt{]}\,\varphi)$$

A.1.6 Regular expressions, part I

The `?` and `+` repetition operators can be expressed by the more general `*` operator as follows:

$$\mathcal{T}(\varphi\texttt{?}) = \mathcal{T}(\varphi\texttt{*[<=1]})$$

$$\mathcal{T}(p\texttt{+}) = \mathcal{T}(p\texttt{*[>=1]})$$

The different variants of the `*` repetition operator are translated as follows into core Salt, where only sequences and the `*[>=`n`]` repetition operator exist. The empty sequence is denoted by ε.

$$
\begin{array}{lll}
\mathcal{T}(\varphi\texttt{*[=}n\texttt{]}) & = & \\
\qquad\text{if } n = 0: & & \varepsilon \\
\qquad\text{if } n = 1: & & \mathcal{T}(\varphi) \\
\qquad\text{else:} & & \mathcal{T}(\varphi\texttt{;}\varphi\texttt{*[=}n-1\texttt{]})
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{T}(\varphi\texttt{[}n\texttt{..}m\texttt{]}) & = & \\
\qquad\text{if } n = 0: & & \mathcal{T}(\varphi\texttt{*[<=}m\texttt{]})
\end{array}
$$

[2] A special case for $n = 1$ is required for situations where there is no next state, because either a surrounding `upto` ended or because we reached time zero in the past. In these situations, even $\mathbf{X}\,\mathit{true}$ would be false, although the conditions for the `holding` operator have been fulfilled.

$$
\begin{array}{llll}
 & \text{else:} & & \mathcal{T}(\varphi\texttt{*[=}n-1\texttt{]}\,\texttt{;}\,\varphi\texttt{*[<=}m-n\texttt{]}\,\texttt{;}\,\varphi)^{3} \\
\mathcal{T}(\varphi\texttt{*[<=}n\texttt{]}) & & = & \\
 & \text{if } n = 0\text{:} & & \varepsilon \\
 & \text{else:} & & \varepsilon \vee \mathcal{T}(\overbrace{\varphi \vee \varphi\,\texttt{;}\,(\varphi \vee \varphi\,\texttt{;}\,(\dots))}^{n})^{4} \\
\mathcal{T}(\varphi\texttt{*[<}n\texttt{]}) & & = & \mathcal{T}(\varphi\texttt{*[<=}n-1\texttt{]}) \\
\mathcal{T}(p\texttt{*[>}n\texttt{]}) & & = & \mathcal{T}(p\texttt{*[>=}n+1\texttt{]})
\end{array}
$$

A.1.7 Iteration operators

The iteration operators are translated as follows:

$$
\begin{array}{lcl}
\mathcal{T}(\texttt{allof}\ list) & = & \bigwedge_{\varphi \in list} \mathcal{T}(\varphi) \\
\mathcal{T}(\texttt{noneof}\ list) & = & \neg \bigvee_{\varphi \in list} \mathcal{T}(\varphi) \\
\mathcal{T}(\texttt{someof}\ list) & = & \bigvee_{\varphi \in list} \mathcal{T}(\varphi) \\
\mathcal{T}(\texttt{exactlyoneof}\ list) & = & \bigvee_{\varphi \in list} (\mathcal{T}(\varphi) \wedge \neg \bigvee_{\psi \in list, \psi \neq \varphi} \mathcal{T}(\psi))
\end{array}
$$

A.2 Translation of core Salt into RLTL

A.2.1 until

$$
\begin{array}{lcl}
\mathcal{T}(\varphi\ \texttt{until excl req}\ \psi) & = & \mathcal{T}(\varphi)\ \mathbf{U}\ \mathcal{T}(\psi) \\
\mathcal{T}(\varphi\ \texttt{until excl opt}\ \psi) & = & (\mathbf{F}\mathcal{T}(\psi)) \Rightarrow (\mathcal{T}(\varphi)\ \mathbf{U}\ \mathcal{T}(\psi)) \\
\mathcal{T}(\varphi\ \texttt{until excl weak}\ \psi) & = & \mathcal{T}(\varphi)\ \mathbf{U}_w\ \mathcal{T}(\psi) \\
\mathcal{T}(\varphi\ \texttt{until incl req}\ \psi) & = & \mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi) \wedge \mathcal{T}(\psi)) \\
\mathcal{T}(\varphi\ \texttt{until incl opt}\ \psi) & = & (\mathbf{F}\mathcal{T}(\psi)) \Rightarrow (\mathcal{T}(\varphi)\ \mathbf{U}\ (\mathcal{T}(\varphi) \wedge \mathcal{T}(\psi))) \\
\mathcal{T}(\varphi\ \texttt{until incl weak}\ \psi) & = & \mathcal{T}(\varphi)\ \mathbf{U}_w\ (\mathcal{T}(\varphi) \wedge \mathcal{T}(\psi))
\end{array}
$$

A.2.2 upto

[3] The trailing φ is necessary for correct translation when followed by a : sequence operator.

[4] The schema used here repeats φ less times than the straightforward translation $\varphi\texttt{*[=}\dots\texttt{]} \vee \varphi\texttt{*[=}\dots\texttt{]} \vee \dots$.

$$
\begin{array}{lll}
\mathcal{T}(\varphi \texttt{ upto excl req } b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{G}\psi\text{:} & & (\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b \\
\qquad \text{if } \mathcal{T}(\varphi) = \neg\mathbf{F}\psi\text{:} & & (\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b \\
\qquad \text{else:} & & (\mathbf{F}b) \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)^5 \\
\mathcal{T}(\varphi \texttt{ upto excl opt } b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{F}\psi\text{:} & & \neg((\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{else:} & & (\mathbf{F}b) \Rightarrow (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)^5 \\
\mathcal{T}(\varphi \texttt{ upto excl weak } b) & = & (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b) \\
\mathcal{T}(\texttt{req } \varphi \texttt{ upto excl req } b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{G}\psi\text{:} & & \neg b \wedge ((\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{if } \mathcal{T}(\varphi) = \neg\mathbf{F}\psi\text{:} & & \neg b \wedge ((\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{else:} & & (\mathbf{F}b) \wedge \neg b \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b)^5 \\
\mathcal{T}(\texttt{req } \varphi \texttt{ upto excl opt } b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{F}\psi\text{:} & & \neg((\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{else:} & & (\mathbf{F}b) \Rightarrow (\neg b \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b))^5 \\
\mathcal{T}(\texttt{req } \varphi \texttt{ upto excl weak } b) & = & \neg b \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b) \\
\mathcal{T}(\texttt{weak } \varphi \texttt{ upto excl req } b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{G}\psi\text{:} & & (\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b \\
\qquad \text{if } \mathcal{T}(\varphi) = \neg\mathbf{F}\psi\text{:} & & (\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b \\
\qquad \text{else:} & & (\mathbf{F}b) \wedge (b \vee (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b))^5 \\
\mathcal{T}(\texttt{weak } \varphi \texttt{ upto excl opt } b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{F}\psi\text{:} & & b \vee \neg((\neg\psi \text{ stop}_{\text{excl}}\ b)\ \mathbf{U}\ b) \\
\qquad \text{else:} & & (\mathbf{F}b) \Rightarrow (b \vee (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b))^5 \\
\mathcal{T}(\texttt{weak } \varphi \texttt{ upto excl weak } b) & = & b \vee (\mathcal{T}(\varphi) \text{ stop}_{\text{excl}}\ b) \\
\mathcal{T}(\varphi \texttt{ upto incl req } b) & = & (\mathbf{F}b) \wedge (\mathcal{T}(\varphi) \text{ stop}_{\text{incl}}\ b) \\
\mathcal{T}(\varphi \texttt{ upto incl opt } b) & = & (\mathbf{F}b) \Rightarrow (\mathcal{T}(\varphi) \text{ stop}_{\text{incl}}\ b) \\
\mathcal{T}(\varphi \texttt{ upto incl weak } b) & = & \\
\qquad \text{if } \mathcal{T}(\varphi) = \mathbf{G}\psi\text{:} & & \neg(\neg b\ \mathbf{U}\ \neg(\psi \text{ stop}_{\text{incl}}\ b)) \\
\qquad \text{if } \mathcal{T}(\varphi) = \neg\mathbf{F}\psi\text{:} & & \neg(\neg b\ \mathbf{U}\ (\psi \text{ stop}_{\text{incl}}\ b)) \\
\qquad \text{else:} & & (\mathcal{T}(\varphi) \text{ stop}_{\text{incl}}\ b)^5
\end{array}
$$

A.2.3 `from`

$$\mathcal{T}(\varphi \texttt{ from incl req } a) \quad = \quad (\neg a)\ \mathbf{U}\ (a \wedge \mathcal{T}(\varphi))$$

[5] The specialised translations exist only for optimisation reasons.

$$
\begin{array}{lll}
\mathcal{T}(\varphi\ \texttt{from incl opt}\ a) & = & \\
\quad\text{if } \mathcal{T}(\varphi) = \mathbf{G}\psi\text{:} & & \mathbf{G}(a \Rightarrow \mathbf{G}\psi) \\
\quad\text{if } \mathcal{T}(\varphi) = \neg\mathbf{F}\psi\text{:} & & \mathbf{G}(a \Rightarrow \neg\mathbf{F}\psi) \\
\quad\text{else:} & & (\neg a)\ \mathbf{U}_w\ (a \wedge \mathcal{T}(\varphi))^6 \\
\mathcal{T}(\varphi\ \texttt{from excl req}\ a) & = & (\neg a)\ \mathbf{U}\ (a \wedge \mathbf{X}\mathcal{T}(\varphi)) \\
\mathcal{T}(\varphi\ \texttt{from excl opt}\ a) & = & (\neg a)\ \mathbf{U}_w\ (a \wedge \mathbf{X}\mathcal{T}(\varphi))
\end{array}
$$

A.2.4 `between`

$$\mathcal{T}(\varphi\ \texttt{between}\ a, b) = \mathcal{T}((\varphi\ \texttt{upto}\ b)\texttt{from}\ a)$$

A.2.5 Exception operators

$$
\begin{array}{lll}
\mathcal{T}(\varphi\ \texttt{accepton}\ b) & = & \mathcal{T}(\varphi)\ \text{acc}\ b \\
\mathcal{T}(\varphi\ \texttt{rejecton}\ b) & = & \mathcal{T}(\varphi)\ \text{rej}\ b
\end{array}
$$

A.2.6 Regular expressions, part II

The `*[>=`n`]` repetition operator is translated as follows. Its translation depends on the next element ψ in the sequence and the sequence operator.

$$
\begin{array}{lll}
\mathcal{T}(p\texttt{*[>=0]}\,\texttt{;}\,\psi) & = & p\ \mathbf{U}\ \mathcal{T}(\psi) \\
\mathcal{T}(p\texttt{*[>=}n\texttt{]}\,\texttt{;}\,\psi) & = & p\ \mathbf{U}\ \mathcal{T}(\overbrace{p\texttt{;}p\texttt{;}\ \ldots\ \texttt{;}p\texttt{;}}^{n}\ \psi) \\
\mathcal{T}(p\texttt{*[>=0]}\,\texttt{:}\,\psi) & = & \mathcal{T}(\psi) \vee \mathcal{T}(p\texttt{*[>=1]}\,\texttt{:}\,\psi) \\
\mathcal{T}(p\texttt{*[>=}n\texttt{]}\,\texttt{:}\,\psi) & = & p\ \mathbf{U}\ \mathcal{T}(\overbrace{p\texttt{;}p\texttt{;}\ \ldots\ \texttt{;}p\texttt{:}}^{n}\ \psi)
\end{array}
$$

For the translation of the sequence operators, we have to define the length of a regular

[6] The specialised translations exist only for optimisation reasons.

expression:

$$|\varphi| := \begin{cases} |\varepsilon| & = 0 \\ |p| & = 1 \\ |p\texttt{*[>=}n\texttt{]}| & = \bot \\ |\varphi_1 ; \varphi_2| & = |\varphi_1| + |\varphi_2| \\ |\varphi_1 : \varphi_2| & = |\varphi_1| + |\varphi_2| - 1 \end{cases}$$

The sequence operators are then translated as follows:

$$\begin{aligned} \mathcal{T}((\varphi_1 \vee \varphi_2) ; \psi) &= \begin{cases} \text{if } |\varphi_1| \neq |\varphi_2| : & \mathcal{T}(\varphi_1 ; \psi) \vee \mathcal{T}(\varphi_2 ; \psi) \\ \text{else:} & \mathcal{T}((\varphi_1 \vee \varphi_2) ; \psi) \end{cases} \\ \mathcal{T}(\varphi ; \psi) &= \mathcal{T}(\varphi) \wedge \mathbf{X}^{|\varphi|} \mathcal{T}(\psi) \\ \mathcal{T}((\varphi_1 \vee \varphi_2) : \psi) &= \begin{cases} \text{if } |\varphi_1| \neq |\varphi_2| : & \mathcal{T}(\varphi_1 : \psi) \vee \mathcal{T}(\varphi_2 : \psi) \\ \text{else:} & \mathcal{T}((\varphi_1 \vee \varphi_2) : \psi) \end{cases} \\ \mathcal{T}(\varphi : \psi) &= \begin{cases} \text{if } \varphi = \varepsilon : & \mathcal{T}(\psi) \\ \text{else:} & \mathcal{T}(\varphi) \wedge \mathbf{X}^{|\varphi|-1} \mathcal{T}(\psi) \end{cases} \end{aligned}$$

A.3 Translation of RLTL into LTL

During this step, the rej and acc operators (RLTL equivalents of the SALT exception operators) as well as the stop operators (introduced during the translation of `upto` and `between`) are replaced by pure LTL expressions. This requires weaving the end conditions into all sub-expressions of the argument. The innermost operators are replaced first, so that the translation process does not have to deal explicitly with nested operators.

A.3.1 acc

$$\begin{aligned} \mathcal{T}(b \text{ acc } a) &= b \vee a \\ \mathcal{T}((\neg\varphi) \text{ acc } a) &= \neg\mathcal{T}(\varphi \text{ rej } a) \\ \mathcal{T}((\varphi \wedge \psi) \text{ acc } a) &= \mathcal{T}(\varphi \text{ acc } a) \wedge \mathcal{T}(\psi \text{ acc } a) \\ \mathcal{T}((\varphi \vee \psi) \text{ acc } a) &= \mathcal{T}(\varphi \text{ acc } a) \vee \mathcal{T}(\psi \text{ acc } a) \\ \mathcal{T}((\varphi \mathbin{\mathbf{U}} \psi) \text{ acc } a) &= \mathcal{T}(\varphi \text{ acc } a) \mathbin{\mathbf{U}} \mathcal{T}(\psi \text{ acc } a) \\ \mathcal{T}((\mathbf{X}\varphi) \text{ acc } a) &= (\mathbf{X}\mathcal{T}(\varphi \text{ acc } a)) \vee a \\ \mathcal{T}((\mathbf{G}\varphi) \text{ acc } a) &= \neg(\neg a \mathbin{\mathbf{U}} \neg\mathcal{T}(\varphi \text{ acc } a)) \\ \mathcal{T}((\mathbf{F}\varphi) \text{ acc } a) &= \mathbf{F}\mathcal{T}(\varphi \text{ acc } a) \end{aligned}$$

The translation of $\Rightarrow$, $\Leftrightarrow$, $\mathbf{U}_w$ and $\mathbf{X}_w$ is done using the corresponding LTL equivalents in A.5.

A.3.2 rej

$$\begin{array}{lcl}
\mathcal{T}(b \text{ rej } r) & = & b \wedge \neg r \\
\mathcal{T}((\neg\varphi) \text{ rej } r) & = & \neg\mathcal{T}(\varphi \text{ acc } r) \\
\mathcal{T}((\varphi \wedge \psi) \text{ rej } r) & = & \mathcal{T}(\varphi \text{ rej } r) \wedge \mathcal{T}(\psi \text{ rej } r) \\
\mathcal{T}((\varphi \vee \psi) \text{ rej } r) & = & \mathcal{T}(\varphi \text{ rej } r) \vee \mathcal{T}(\psi \text{ rej } r) \\
\mathcal{T}((\varphi \mathbf{\ U\ } \psi) \text{ rej } r) & = & \mathcal{T}(\varphi \text{ rej } r) \mathbf{\ U\ } \mathcal{T}(\psi \text{ rej } r) \\
\mathcal{T}((\mathbf{X}\varphi) \text{ rej } r) & = & (\mathbf{X}\mathcal{T}(\varphi \text{ rej } r)) \wedge \neg r \\
\mathcal{T}((\mathbf{G}\varphi) \text{ rej } r) & = & \mathbf{G}\mathcal{T}(\varphi \text{ rej } r) \\
\mathcal{T}((\mathbf{F}\varphi) \text{ rej } a) & = & \neg r \mathbf{\ U\ } \mathcal{T}(\varphi \text{ rej } r)
\end{array}$$

The translation of $\Rightarrow$, $\Leftrightarrow$, $\mathbf{U}_w$ and $\mathbf{X}_w$ is done using the corresponding LTL equivalents in A.5.

A.3.3 $\text{stop}_{\text{incl}}$

$$\begin{array}{lcl}
\mathcal{T}(b \text{ stop}_{\text{incl}}\ s) & = & b \\
\mathcal{T}((\neg\varphi) \text{ stop}_{\text{incl}}\ s) & = & \neg\mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s) \\
\mathcal{T}((\varphi \wedge \psi) \text{ stop}_{\text{incl}}\ s) & = & \mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s) \wedge \mathcal{T}(\psi \text{ stop}_{\text{incl}}\ s) \\
\mathcal{T}((\varphi \vee \psi) \text{ stop}_{\text{incl}}\ s) & = & \mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s) \vee \mathcal{T}(\psi \text{ stop}_{\text{incl}}\ s) \\
\mathcal{T}((\varphi \mathbf{\ U\ } \psi) \text{ stop}_{\text{incl}}\ s) & = & (\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s)) \mathbf{\ U\ } \mathcal{T}(\psi \text{ stop}_{\text{incl}}\ s) \\
\mathcal{T}((\mathbf{X}\varphi) \text{ stop}_{\text{incl}}\ s) & = & \neg s \wedge \mathbf{X}\mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s) \\
\mathcal{T}((\mathbf{X}_w\varphi) \text{ stop}_{\text{incl}}\ s) & = & s \vee \mathbf{X}\mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s) \\
\mathcal{T}((\mathbf{G}\varphi) \text{ stop}_{\text{incl}}\ s) & = & \neg(\neg s \mathbf{\ U\ } \neg\mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s)) \\
\mathcal{T}((\mathbf{F}\varphi) \text{ stop}_{\text{incl}}\ s) & = & (\neg s) \mathbf{\ U\ } \mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s) \\
\mathcal{T}((\varphi \text{ S } \psi) \text{ stop}_{\text{incl}}\ s) & = & \mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s) \text{ S } \mathcal{T}(\psi \text{ stop}_{\text{incl}}\ s)^7 \\
\mathcal{T}((\mathbf{Y}\varphi) \text{ stop}_{\text{incl}}\ s) & = & \mathbf{Y}\mathcal{T}(\varphi \text{ stop}_{\text{incl}}\ s)^7
\end{array}$$

The past stop operators are translated in a similar way as the future stop operators, but affecting only past operators. The translation of $\mathbf{U}_w$, $\Rightarrow$ and $\Leftrightarrow$ is done using the corresponding LTL equivalents in A.5.

A.3.4 $\mathrm{stop}_{\mathrm{excl}}$

$$
\begin{array}{lcl}
\mathcal{T}(b \text{ stop}_{\text{excl}}\ s) & = & b \\
\mathcal{T}((\neg\varphi) \text{ stop}_{\text{excl}}\ s) & = & \neg\mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \\
\mathcal{T}((\varphi \wedge \psi) \text{ stop}_{\text{excl}}\ s) & = & \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \wedge \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s) \\
\mathcal{T}((\varphi \vee \psi) \text{ stop}_{\text{excl}}\ s) & = & \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s) \vee \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s) \\
\mathcal{T}((\varphi\ \mathbf{U}\ \psi) \text{ stop}_{\text{excl}}\ s) & = & (\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s))\ \mathbf{U}\ (\neg s \wedge \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s)) \\
\mathcal{T}((\varphi\ \mathbf{U}_w\ \psi) \text{ stop}_{\text{excl}}\ s) & = & \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)\ \mathbf{U}_w\ (s \vee \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s)) \\
\mathcal{T}((\mathbf{X}\varphi) \text{ stop}_{\text{excl}}\ s) & = & \mathbf{X}(\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)) \\
\mathcal{T}((\mathbf{X}_w\varphi) \text{ stop}_{\text{excl}}\ s) & = & \mathbf{X}(s \vee \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)) \\
\mathcal{T}((\mathbf{G}\varphi) \text{ stop}_{\text{excl}}\ s) & = & \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)\ \mathbf{U}_w\ s \\
\mathcal{T}((\mathbf{F}\varphi) \text{ stop}_{\text{excl}}\ s) & = & (\neg s)\ \mathbf{U}\ (\neg s \wedge \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)) \\
\mathcal{T}((\varphi\ \mathrm{S}\ \psi) \text{ stop}_{\text{excl}}\ s) & = & \mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)\ \mathrm{S}\ \mathcal{T}(\psi \text{ stop}_{\text{excl}}\ s)^7 \\
\mathcal{T}((\mathbf{Y}\varphi) \text{ stop}_{\text{excl}}\ s) & = & \mathbf{Y}\mathcal{T}(\varphi \text{ stop}_{\text{excl}}\ s)^7
\end{array}
$$

The past stop operators are translated in a similar way as the future stop operators, but affecting only past operators. The translation of $\Rightarrow$ and $\Leftrightarrow$ is done using the corresponding LTL equivalents in A.5.

A.4 Optimisation

The following equivalences are used for optimisation:

[7] Notice how the future stop operator affects only future operators and leaves the past operators unchanged. The past operators not listed here are translated similarly.

$$
\begin{array}{lcl}
\mathit{true}\ \mathbf{U}\ \varphi & \Leftrightarrow & \mathbf{F}\varphi \\
\neg\mathbf{F}\neg\varphi & \Leftrightarrow & \mathbf{G}\varphi \\
\mathbf{G}\mathbf{G}\varphi & \Leftrightarrow & \mathbf{G}\varphi \\
\mathbf{F}\mathbf{F}\varphi & \Leftrightarrow & \mathbf{F}\varphi \\
\neg\varphi\ \mathbf{U}\ \varphi & \Leftrightarrow & \mathbf{F}\varphi \\
\mathbf{G}(\varphi\ \mathbf{U}_w\ \psi) & \Leftrightarrow & \mathbf{G}(\varphi \vee \psi) \\
\varphi\ \mathbf{U}_w\ (\varphi \wedge \psi) & \Leftrightarrow & \neg(\neg\psi\ \mathbf{U}\ \neg\varphi) \\
(\varphi \vee \psi)\ \mathbf{U}\ \psi & \Leftrightarrow & \varphi\ \mathbf{U}\ \psi
\end{array}
$$

Furthermore, Boolean operators with constant arguments (e. g., $\mathit{true} \wedge a$) are eliminated.

A.5 Operator replacement

The following equivalences are used to express certain operators through others if necessary for the current output syntax.

$$
\begin{array}{lcl}
\mathbf{G}\varphi & \Leftrightarrow & \neg(\mathit{true}\ \mathbf{U}\ \neg\varphi) \\
\mathbf{F}\varphi & \Leftrightarrow & \mathit{true}\ \mathbf{U}\ \varphi \\
\mathbf{X}_w\psi & \Leftrightarrow & \neg\mathbf{X}(\neg\varphi) \\
\varphi\ \mathbf{U}_w\ \psi & \Leftrightarrow & \begin{cases} \text{if } |\psi| \leq |\varphi|^8\text{:} & \neg(\neg\psi\ \mathbf{U}\ (\neg\varphi \wedge \neg\psi)) \\ \text{else:} & (\varphi\ \mathbf{U}\ \psi) \vee \mathbf{G}\varphi \end{cases} \\
\neg(\neg\varphi\ \mathbf{U}\ \neg\psi) & \Leftrightarrow & \varphi\ \mathbf{R}\ \psi
\end{array}
$$

A.6 Translation of timed operators

A.6.1 Timed Salt into timed RLTL

$$
\begin{array}{lcl}
\mathcal{T}(\texttt{next timed[}\sim c\texttt{]}\varphi) & = & \rhd_{\sim c}\mathcal{T}(\varphi) \\
\mathcal{T}(\varphi\ \texttt{until timed[}\sim c\texttt{]}\ \psi) & = & \mathcal{T}(\varphi)\ \mathbf{U}_{\sim c}\ \mathcal{T}(\psi) \\
\mathcal{T}(\varphi\ \texttt{until timed[}\sim c\texttt{] weak}\ \psi) & = & \mathcal{T}(\varphi)\ \mathbf{U}_{w\sim c}\ \mathcal{T}(\psi) \\
\mathcal{T}(\varphi\ \texttt{until timed[}\sim c\texttt{] excl req}\ \psi) & = & \mathcal{T}(\varphi)\ \mathbf{U}_{\sim c}\ \mathcal{T}(\psi)
\end{array}
$$

[8] As a heuristic estimation for the size of a formula, the number of temporal operators in the formula is used.

$$
\begin{array}{lcl}
\mathcal{T}(\varphi \; \texttt{until timed[}\sim c\texttt{] excl opt} \; \psi) & = & (\mathbf{F}_{\sim c}\mathcal{T}(\psi)) \Rightarrow \\
 & & (\mathcal{T}(\varphi) \; \mathbf{U}_{\sim c} \; \mathcal{T}(\psi)) \\
\mathcal{T}(\varphi \; \texttt{until timed[}\sim c\texttt{] excl weak} \; \psi) & = & \mathcal{T}(\varphi) \; \mathbf{U}_{w\sim c} \; \mathcal{T}(\psi) \\
\mathcal{T}(\varphi \; \texttt{until timed[}\sim c\texttt{] incl req} \; \psi) & = & \mathcal{T}(\varphi) \; \mathbf{U}_{\sim c} \; (\mathcal{T}(\varphi) \wedge \mathcal{T}(\psi)) \\
\mathcal{T}(\varphi \; \texttt{until timed[}\sim c\texttt{] incl opt} \; \psi) & = & (\mathbf{F}_{\sim c}\mathcal{T}(\psi)) \Rightarrow (\mathcal{T}(\varphi) \; \mathbf{U}_{\sim c} \\
 & & (\mathcal{T}(\varphi) \wedge \mathcal{T}(\psi))) \\
\mathcal{T}(\varphi \; \texttt{until timed[}\sim c\texttt{] incl weak} \; \psi) & = & \mathcal{T}(\varphi) \; \mathbf{U}_{w\sim c} \; (\mathcal{T}(\varphi) \wedge \mathcal{T}(\psi)) \\
\mathcal{T}(\texttt{timed[}\sim c\texttt{]} \; \varphi \; \texttt{releases} \; \psi) & = & \mathcal{T}(\psi) \; \mathbf{U}_{w\sim c} \; (\mathcal{T}(\psi) \wedge \mathcal{T}(\varphi)) \\
\mathcal{T}(\texttt{always timed[}\sim c\texttt{]} \; \varphi) & = & \mathbf{G}_{\sim c}\mathcal{T}(\varphi) \\
\mathcal{T}(\texttt{eventually timed[}\sim c\texttt{]} \; \varphi) & = & \mathbf{F}_{\sim c}\mathcal{T}(\varphi)
\end{array}
$$

A.6.2 Timed RLTL into extended TLTL

acc (`accepton`):

$$
\begin{array}{lcl}
\mathcal{T}((\triangleright_{\sim c}\varphi) \text{ acc } a) & = & a \vee \triangleright_{\sim c}\mathcal{T}(\varphi \text{ acc } a) \\
\mathcal{T}((\varphi \; \mathbf{U}_{\sim c} \; \psi) \text{ acc } a) & = & \mathcal{T}(\varphi \text{ acc } a) \; \mathbf{U}_{\sim c} \; \mathcal{T}(\psi \text{ acc } a) \\
\mathcal{T}((\varphi \; \mathbf{U}_{w\sim c} \; \psi) \text{ acc } a) & = & \begin{cases} \text{if } |\psi| \leq |\varphi|^{9}\text{:} & \neg(\neg\mathcal{T}(\psi \text{ acc } a) \; \mathbf{U}_{\sim c} \\ & (\neg\mathcal{T}(\varphi \text{ acc } a) \wedge \neg\mathcal{T}(\psi \text{ acc } a))) \\ \text{else:} & (\mathcal{T}(\varphi \text{ acc } a) \; \mathbf{U}_{\sim c} \; \mathcal{T}(\psi \text{ acc } a)) \vee \\ & \neg(\neg a \; \mathbf{U}_{\sim c} \; \neg\mathcal{T}(\varphi \text{ acc } a)) \end{cases} \\
\mathcal{T}((\mathbf{G}_{\sim c}\varphi) \text{ acc } a) & = & \neg(\neg a \; \mathbf{U}_{\sim c} \; \neg\mathcal{T}(\varphi \text{ acc } a)) \\
\mathcal{T}((\mathbf{F}_{\sim c}\varphi) \text{ acc } a) & = & \mathbf{F}_{\sim c}\mathcal{T}(\varphi \text{ acc } a)
\end{array}
$$

rej (`rejecton`):

$$
\begin{array}{lcl}
\mathcal{T}((\triangleright_{\sim c}\varphi) \text{ rej } r) & = & \neg r \wedge \mathbf{X}(\neg r \; \mathbf{U} \; \mathcal{T}(\varphi \text{ rej } r)) \wedge \triangleright_{\sim c}\mathcal{T}(\varphi \text{ rej } r)^{10} \\
\mathcal{T}((\varphi \; \mathbf{U}_{\sim c} \; \psi) \text{ rej } r) & = & \mathcal{T}(\varphi \text{ rej } r) \; \mathbf{U}_{\sim c} \; \mathcal{T}(\psi \text{ rej } r)
\end{array}
$$

[9] As a heuristic estimation for the size of a formula, the number of temporal operators in the formula is used.

$$
\begin{aligned}
\mathcal{T}((\varphi\ \mathbf{U}_{w\sim c}\ \psi) \text{ rej } r) &= \begin{cases} \text{if } |\psi| \leq |\varphi|^{11}\text{:} & \neg(\neg\mathcal{T}(\psi \text{ rej } r)\ \mathbf{U}_{\sim c} \\ & (\neg\mathcal{T}(\varphi \text{ rej } r) \wedge \neg\mathcal{T}(\psi \text{ rej } r))) \\ \text{else:} & (\mathcal{T}(\varphi \text{ rej } r)\ \mathbf{U}_{\sim c}\ \mathcal{T}(\psi \text{ rej } r)) \vee \\ & \mathbf{G}_{\sim c}\mathcal{T}(\varphi \text{ rej } r) \end{cases} \\
\mathcal{T}((\mathbf{G}_{\sim c}\varphi) \text{ rej } r) &= \mathbf{G}_{\sim c}\mathcal{T}(\varphi \text{ rej } r) \\
\mathcal{T}((\mathbf{F}_{\sim c}\varphi) \text{ rej } r) &= \neg r\ \mathbf{U}_{\sim c}\ \mathcal{T}(\varphi \text{ rej } r)
\end{aligned}
$$

stop operators: The stop operators do not influence timed operators, i. e., any timed operator and its arguments are left unchanged.

A.6.3 Extended TLTL into pure TLTL

$$
\begin{aligned}
\mathcal{T}(\varphi\ \mathbf{U}_{\sim c}\ \psi) &= (\mathcal{T}(\varphi)\ \mathbf{U}\ \mathcal{T}(\psi)) \wedge (\mathcal{T}(\psi) \vee \rhd_{\sim c}\mathcal{T}(\psi)) \\
\mathcal{T}(\varphi\ \mathbf{U}_{w\sim c}\ \psi) &= (\mathcal{T}(\varphi)\ \mathbf{U}\ \mathcal{T}(\psi)) \vee (\mathcal{T}(\varphi) \wedge \neg \rhd_{\sim c}\neg\mathcal{T}(\varphi)) \\
\mathcal{T}(\mathbf{G}_{\sim c}\varphi) &= \mathcal{T}(\varphi) \wedge \neg(\rhd_{\sim c}\neg\mathcal{T}(\varphi)) \\
\mathcal{T}(\mathbf{F}_{\sim c}\varphi) &= \mathcal{T}(\varphi) \vee \rhd_{\sim c}\mathcal{T}(\varphi)
\end{aligned}
$$

[10] The $\mathbf{X}$ is required because $\rhd_{\sim c}\varphi$ is not supposed to match occurrences of φ at the current state, but $\mathbf{U}$ would.

[11] As a heuristic estimation for the size of a formula, the number of temporal operators in the formula is used.

Appendix B

Runtime reflection on the web: Obtaining the files

This appendix gives additional information on how to actually obtain available implementations of the methods described in this thesis, and points to accompanying example use-cases. In particular, this appendix points to the source code and implementations of an optimising Salt compiler, an interactive web front-end to Salt, an example implementation, and example use-cases of the different domain-independent analysis layers that constitute the runtime reflection framework; that is, monitoring and diagnosis.

B.1 Tools and source code

An optimising compiler for Salt is available free of charge, under an open source license, namely the GNU General Public License (see Free Software Foundation [1991]), from the web page depicted in Fig. B.1). Besides extensive documentation on Salt, its compiler and the according language features, this web page also features an interactive front-end to a remote Salt compiler. That is, users may be testing Salt without having to install a local copy of the compiler on their machines. The front-end accepts a Salt specification as textual input and returns a translation of the specification either in SPIN or SMV syntax (see Fig. B.2).

The layers monitoring and diagnosis of the runtime reflection framework are also available free of charge, under the same open source license (GPL), from a web page depicted in Fig. B.3. This web page contains both the source code as well as the example outlined previously in §4.4.2, demonstrating the application of runtime reflection in a real-world C++ application. Moreover, the web page also provides a custom logging layer, which is written in C++, although runtime reflection as described in this thesis is not constrained to just one type of logging facility. The logging layer available on the web page was used to perform the case study described in §4.4.2.

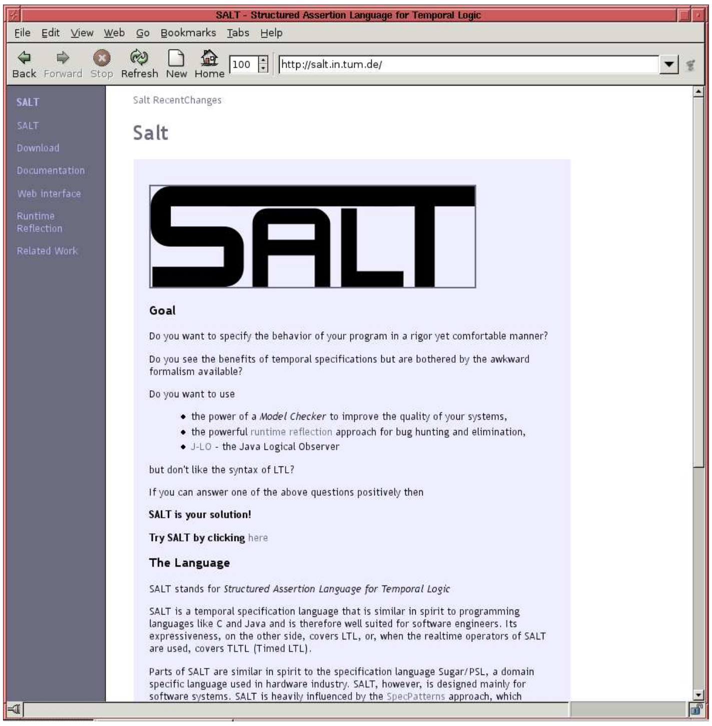


Fig. B.1: The SALT web page, `http://salt.in.tum.de/`.

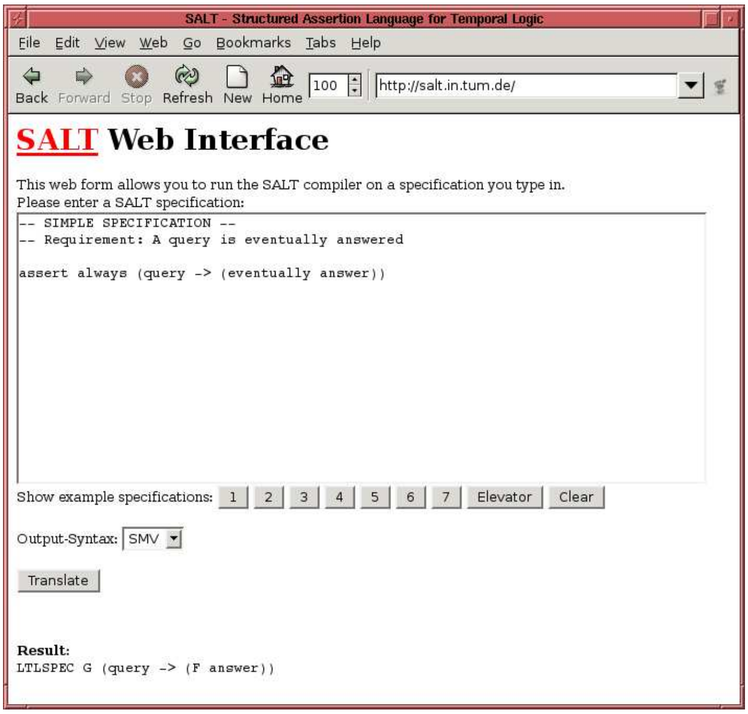


Fig. B.2: The SALT web interface.

Fig. B.3: The runtime reflection web page, `http://runtime.in.tum.de/`.

B.2 CASE-tool integration: SALT in AutoFocus2

Additionally, SALT has been integrated in the freely available CASE-tool AUTOFOCUS2 (see also `http://www4.in.tum.de/~af2/`). Note that AUTOFOCUS2 is not open source, however, free of charge. AUTOFOCUS2 is the successor of AUTOFOCUS as previously described in §4.1, for instance. Like AUTOFOCUS, AUTOFOCUS2 is a CASE-tool for the design, development, and testing of distributed embedded systems. It provides a number of graphical notations that are visually similar to the UML-notation, and allows model simulation as well as code generation for various target languages (e. g., Java, C, Ada). Moreover, AUTOFOCUS2 designs can be verified using external model checkers, such as SMV. For this purpose, AUTOFOCUS2 performs a number of predefined model abstractions and translates the model into a finite-space model that is understood by the model checker. The user in turn has to specify properties in temporal logic which the model then has to fulfil. For this purpose, AUTOFOCUS2 offers two possibilities: it allows the direct input of LTL properties, as well as SALT specifications, which then get translated into plain LTL before invoking the model checking process. This is also depicted in Fig. B.4.

SALT as described in this thesis is integral part of the AUTOFOCUS2 distribution, and its installation and usage outlined in the program's manuals.

As such SALT serves not only as an interface to the runtime reflection framework, but

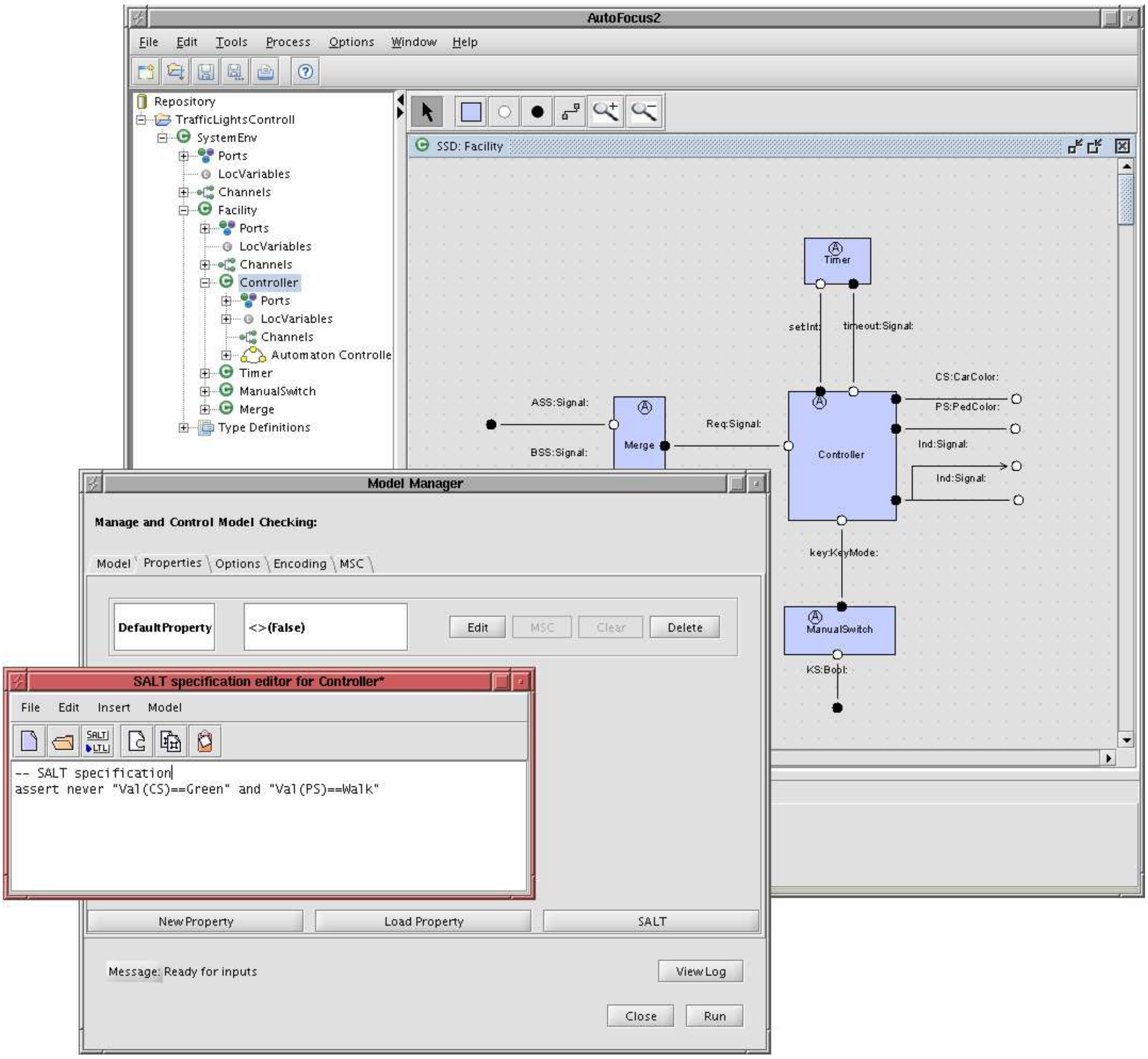


Fig. B.4: The model checking interface and SALT editor in AUTOFOCUS 2.

to other design and verification frameworks that incorporate standard model checking facilities.

Index