Model Checking Real-Time Systems based on Partition Refinement

Ronald lutje Spelberg
Model Checking Real-Time Systems
based on Partition Refinement

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 25 mei 2004 om 10:30 uur
door

Ronald Frank LUTJE SPELBERG

informatica ingenieur
geboren te Enschede
Dit proefschrift is goedgekeurd door de promotor: Prof.dr.ir. J. van Katwijk

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof.dr.ir. J. van Katwijk, Technische Universiteit Delft, promotor
Dr.ir. W.J. Toetenel, Technische Universiteit Delft, toegevoegd promotor
Prof.dr.ir. J.C.M. Baeten, Technische Universiteit Eindhoven
Prof.dr. K.G. Larsen, Universiteit van Aalborg
Prof.dr. H. Brinksma, Universiteit Twente
Dr. N. Halbwachs, Verimag Laboratory Grenoble
Prof.dr.ir. H.J. Sips, Technische Universiteit Delft
3.2.2 The minimal model generation algorithm

3.3 Abstract models for timed systems
3.3.1 Time-abstracting bisimulation
3.3.2 Abstract models and abstract paths

3.4 Region graphs and stability
3.4.1 Region graphs
3.4.2 Stability for region graphs
3.4.3 Characterizing stability

3.5 An example

3.6 An algorithm for parametric reachability analysis
3.6.1 Parametric reachability analysis
3.6.2 Refining the general algorithm
3.6.3 The algorithm
3.6.4 Correctness

3.7 Discussion

4 Evaluating TCTL properties
4.1 Introduction
4.2 Abstract models for TCTL verification
4.2.1 Normalizing properties
4.2.2 Divergence
4.2.3 Property variables
4.2.4 Bisimulation abstraction on D-CTL

4.3 Deciding fair CTL properties
4.3.1 Partial models
4.3.2 Deciding temporal operators using least fixed points
4.3.3 Deciding properties in non-fair CTL
4.3.4 Recognizing fair paths
4.3.5 Deciding temporal operators in fair CTL
4.3.6 Summarizing

4.4 Discussion

5 Symbolic state space representation
5.1 Introduction
5.2 Splitting and propagation
5.3 Splitting trees
5.3.1 An introduction to splitting trees
5.3.2 Definition of splitting trees
5.3.3 Restrictions

5.4 Propagation
5.4.1 Propagation over discrete edges
5.4.2 Propagation over time edges

5.5 Reduction
5.5.1 Reduction for difference constraints
5.5.2 Reduction in case of linear constraints
5.5.3 Reduction for enumerated types

5.6 Extensions
5.6.1 Optimizing reduction
5.6.2 Approximate reduction
# Model checking

## TCTL

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Introduction</td>
<td>131</td>
</tr>
<tr>
<td>6.2 Evaluation of TCTL properties</td>
<td>132</td>
</tr>
<tr>
<td>6.2.1 Partial region graphs</td>
<td>132</td>
</tr>
<tr>
<td>6.2.2 Deciding TCTL using region graphs</td>
<td>133</td>
</tr>
<tr>
<td>6.2.3 Decision rules for TCTL</td>
<td>134</td>
</tr>
<tr>
<td>6.2.4 Optimizations</td>
<td>142</td>
</tr>
<tr>
<td>6.3 Model checking</td>
<td>142</td>
</tr>
<tr>
<td>6.3.1 Introduction to the algorithm</td>
<td>143</td>
</tr>
<tr>
<td>6.3.2 Algorithm infrastructure</td>
<td>144</td>
</tr>
<tr>
<td>6.3.3 The algorithm</td>
<td>147</td>
</tr>
<tr>
<td>6.3.4 Extensions of the algorithm: optimal decision making</td>
<td>162</td>
</tr>
<tr>
<td>6.3.5 Correctness of the algorithm</td>
<td>164</td>
</tr>
<tr>
<td>6.4 Practical results</td>
<td>175</td>
</tr>
<tr>
<td>6.4.1 Two model checkers</td>
<td>175</td>
</tr>
<tr>
<td>6.4.2 Performance comparison</td>
<td>176</td>
</tr>
<tr>
<td>6.5 Discussion</td>
<td>178</td>
</tr>
</tbody>
</table>
Acknowledgements

Writing this thesis was a rather solitary way of spending time. It started somewhere in 1999 and although there were long periods of inactivity, it was always somewhere in the back of my mind. Knowing five years ago what I know now, I would definitely have turned to more fulfilling things to do. Obviously, the only one I have to thank for that is myself.

I most of all have to thank Hans Toetenel for being a stimulating mentor in research. Hans, it was a pleasure to work with you.

Furthermore, I thank my promotor Jan van Katwijk for facilitating my PhD work. I thank Nicolas Halbwachs, Kim Larsen, Ed Brinksma, Jos Baeten and Henk Sips for being a member of my promotion committee. I thank my former AIO-colleagues and roommates Klaas, Leon and Ruud for nice cooperations and stimulating discussions. Also, I thank other colleagues at Delft that made the SEPPC group a pleasant environment to work in. Finally, I would like to thank people at Thales, for granting me an occasional week of thesis writing.

Furthermore, I would like to take this opportunity to express my gratitude to my parents for having provided a stimulating and loving environment to grow up in. I also have to mention my little brother and friend Jeffrey: I am glad you find such happiness in finishing before me, although I prefer a card-playing competition. Finally, my last words are for Jacqueline: Thank you for your friendship, your love, for (reluctantly) tolerating a reluctantly absent husband and for being a wonderful mother to our equally wonderful daughter Carlijn.

Ronald lutje Spelberg
April 2004
Chapter 1

Introduction

Computer systems are more and more becoming an indispensable part of our daily life. As a consequence, our society has become highly dependent on computer hardware and software. This includes computer systems of which incorrect behaviour can have disastrous consequences. Often, this concerns what is called safety-critical systems, for example nuclear plant controllers and certain types of medical systems, but also systems in which errors can bring enormous costs, like mass-produced microprocessors and certain types of financial systems.

Also, computer systems are becoming increasingly complex. Technological advances in hardware technology have made it possible to build amazingly large and complex systems. This is particularly true for software systems, for which — due to the rapid increase in hardware performance — limitations on size are quickly becoming less relevant.

This has resulted in a situation in which building correct complex software systems is nearly impossible. The best that can be done is ensuring that for certain critical aspects, the correctness can shown to be likely. For hardware systems, potential complexity is somewhat more limited, but still correctness cannot always be guaranteed. As a result, in the short history of computers we have seen numerous examples of design errors with huge consequences. The Ariane 5 crash due to a software error and the Pentium processor hardware error are prominent examples. In particular with respect to software, an erroneous system seems to be the standard, rather than an exceptional condition.

The fact that it is not possible to guarantee a reasonable level of correctness for critical systems, gives rise to very low expectations concerning the level of correctness of less critical systems. And indeed, these expectations are easily fulfilled by the quality of current main-stream software applications. It is safe to say that building (sufficiently) correct software and hardware systems is still a major challenge — even more if one also takes into account other important characteristic dimensions of the development process, like development costs and longevity.

It is therefore not surprising that a lot of attention has been paid to technologies and tools that focus on improving the quality of software and hardware, in particular those aiming at avoiding critical and costly errors. This has resulted in the wide-spread usage of verification approaches like testing, simulation and peer reviewing. Typically, for modern software systems, the effort dedicated to verification is higher than the effort involved in actually building the system. Also, in computer science there has been quite an effort to improving the building process of software, through the introduction of a wide range of methods and techniques aimed at ensuring that the resulting computer systems have an
acceptable degree of correctness.

However, for reasons mentioned earlier, there is still a strong need for further development of techniques that have the potential to aid in producing more reliable systems. One direction that could least partly fulfill this need — in particular for systems in which errors have huge consequences — is that of formal methods.

Formal methods were introduced more than twenty years ago as a potential approach towards building rigorously correct systems. The field of formal methods propagates approaches to designing software and hardware systems that are firmly founded on mathematics. The basic idea is that if systems are designed in terms of mathematically-based formalisms then a basis is created for building systems that are "provably" correct. Despite much research, formal methods have not yet been adopted by the software industry.

However, in recent years an approach in formal methods has emerged which seems to have the potential to play a significant role in industrial software and hardware development practice. This thesis focuses on this particular field in formal methods, a formal verification approach called model checking. Model checking is an automated technique that can be used to verify formal specifications of hardware or software systems against formal specifications of properties. It is based on exhaustive exploration of state spaces. Its strength comes from the fact this can be done automatically — without any complex interference of humans. The major limitation is in its scalability — state spaces tend to explode as systems become more complex. Therefore, the major challenge in model checking is the search for techniques that allow efficient exploration of large state spaces.

This thesis addresses a certain class of verification problems, namely those that involve concurrent systems in which quantitative timing aspects play a relevant role — a class of systems in which formal verification is particularly relevant. First of all, concurrent systems have an inherent complexity — due to the non-determinism introduced by the specification of parallel execution — that makes it very hard to ensure correctness of even quite simple systems. Also, timing aspects play an essential role for many safety-critical systems, for example flight control systems and nuclear plant controllers. Additionally, such timing aspects introduce a high level of complexity due to the non-deterministic behaviour of the system's context.

Note that model checking approaches for real-time concurrent systems have a larger application area than just safety-critical systems. Because of its automatic nature, model checking can be useful in general for verifying critical aspects of designs, like for example complex communication protocol standards.

1.1 Background

Before defining the scope of this thesis, some background is provided. More elaborate introductions to model checking, including model checking for real-time systems, can be found in [40] and [68].

1.1.1 Formal methods

The term formal methods refers to the use of mathematical techniques in the development of computer hardware and software systems. The central idea behind formal methods is that mathematically-based languages enable precise and unambiguous modelling, and as a result also allow mathematical analysis. This characterization introduces a distinction
between two aspects of use of formal methods, namely formal specification and formal verification.

With formal specification we mean that the meaning of what is specified is unambiguously defined. By using a so-called formal specification language, one is forced to specify in an exact manner, resulting in better insight in the problem at hand. Formal specification makes errors, inconsistency, ambiguity and incompleteness explicit.

Formal specification enables formal verification, which exploits the exact interpretation of specifications to perform reasoning about that specification. Reasoning is applied to show that a specification has certain desired properties. It can be done using a variety of mathematically-based techniques. Formal methods differ in the emphasis that is put on each of the two aspects (specification and verification), although almost always some of each aspect will be involved.

There is great variation in how, and to which extent, mathematical techniques are used. Formal methods can be applied in the specification, design, construction and verification of systems, with different levels of rigour. The most general approach is to apply a formal approach integrally in one or more steps in the development cycle. In particular in the earliest development steps, this results in biggest payoff in terms of early error detection, eventually leading to low development costs. However, this does require a high level of maturity in terms of languages, techniques and tools.

Another view on the application of formal methods is that a formal method provides support for a specific part of a development process. In this view, formal methods are seen as a tool that can be selectively used when appropriate. This can be done in two ways. First of all, critical components can be selected which are subjected to rigorous development approach based on formal methods, while other, less critical, parts are developed using "standard" development techniques. A control unit for a nuclear reactor would be a likely candidate. Another way to selectively use formal methods, is to apply them to selected aspects of a system. In that case, the focus is often on verifying certain specific properties of a system. Modelling then focuses on creating abstract models that only take into account those aspects that are of relevance to the property. The model checking approach is most likely to fit in the latter category. It is typically applied by modelling an abstraction of a part of a system with the intention to verify a specific desired property of that system.

A formal method can be defined as a combination of formal languages, techniques and supporting tools. The formal languages are needed to model systems and their properties. Techniques are needed to analyse specifications and support modelling activities. Finally, tooling is essential in any formal method, since relying on humans for correct modelling and analysis defies the primary goal of formal methods. In any case, for formal methods to be successful a lot of attention has to be paid to integration with existing practice in system and software engineering methods and tooling. This is a factor that seems to be hampering the broader introduction of formal methods.

Roughly, one could say that the main focus in this thesis is on defining concepts and techniques that can be used to build a verification tool, namely a model checker.

1.1.2 Formal specification

For specification languages, a distinction can be made between informal, semiformal and formal languages. Natural languages are typically informal languages because both syntax and semantics of these languages is not formally defined. Semi-formal and formal
languages on the other hand, do have a formally defined syntax — there is a well-defined mathematical description of the set of valid programs/specifications for that language. Many of the notations used in modern software engineering methods are semiformal. Finally, formal languages are distinguished from semi-formal languages by having a formal semantics. This means that the meaning of a specification is defined in terms of an underlying well-defined mathematical model, referred to as its semantics.

Underlying models can be based on well-established mathematical theories like set theory or propositional logic. In other cases the underlying models are more specific for the language that is given semantics. Such models are often relatively simple, and mathematical soundness can easily be shown. An example of such an underlying model is the concept of transition systems — basically states and transitions between these states — which is the underlying model of the formalisms described in this thesis.

The term specification language can be seen as more general than the term programming language. A program written in some programming language can be seen as a description that can be used to build a concrete machine, by running it on computer hardware. An essential aspect of a programming language is that it results in complete and explicit specifications — it produces executable models. A specification language on the other hand, also allows abstraction — it can produce models that abstract from implementation details. This can be a general form of abstraction, but abstraction can also focus on a certain aspect of a system, or even only a very specific property of a system. For formal specification languages the ability to abstract is essential, because it allows the specification of systems at different levels of abstraction — which is a basis for formal verification.

A distinction can be made between two different approaches to formal specification, namely model-oriented specification and property-oriented specification [34]. In the model-oriented style of specification, a model of the system is build at some level of abstraction. The property-oriented approach to specification defines only relevant characteristics of the system that is modelled. Instead of building a complete model by describing its structure, only constraints are stated that hold the essence of what is being modelled. The property-oriented approach is more abstract, because a specifier can conveniently focus on relevant aspects without being bothered with internal details of the model. It therefore tends to lead to specifications that are free of implementation bias. On the other hand, model-oriented specifications are easier to relate to, because they define abstract machines. Model checking is a verification approach that uses a combination of a model-oriented and a property-oriented specification language, the former specifying systems and the latter specifying desirable properties of these systems.

Traditionally, there has been a differentiation between notations for sequential, data-intensive systems on the one hand (for example VDM [67], and Z [99]), and notations for modelling concurrency and real-time behaviour on the other hand (for example Petri nets [93], Statecharts [54], process algebras [84, 14] and timed automata [8]). While the former are usually not able to naturally model concurrency and real-time behaviour, the latter often do not allow the modelling of data, or only allow the modelling of some primitive data elements. The two main types of specification languages have lead to distinct areas in formal methods, although languages have been proposed that combine both worlds in one notation, for example LOTOS [20, 107] and MTCCS [102, 70]. These languages tend to be more complex, both syntactically as well as semantically.

If in modelling systems, quantitative time plays a role, then real-time formalisms are needed. These formalisms allow the modelling of events in time — not only their relative
ordering but also their distance in time. One of the best-known real-time model-oriented formalism is timed automata [8], but many others exist — for example timed process algebras like ATP [87], timed ACP [13] and MTCCS [70]. A further extension of real-time formalisms are hybrid system specification languages, which allow the modelling of physical phenomena that vary continuously at a different rate. This is particularly important for the analysis of systems in which the behaviour of the physical environment has to be taken into account.

As was mentioned already, the work described in this thesis is based on a dual language approach. A model-oriented language based on timed automata is used for specifying (aspects of) real-time concurrent systems, and a property-oriented language is used for specifying desired properties of such systems. In the light of the earlier mentioned distinction between description focus and verification focus, the work described in this thesis is primarily focused on verification. This means that modelling is primary seen as a means to enable verification.

1.1.3 Formal verification

A specification notation with a formal semantics enables formal verification. Three aspects are involved in the application of formal verification: notions of correctness; techniques that state how correctness statements can be proved; and tools to support or automate the verification process.

The most basic notion of correctness is syntactical correctness. A specification is syntactically correct if it satisfies the syntactical rules defined for the language. For a formal specification to be useful, it also has to be semantically correct, meaning that it is consistent with the semantics definition of the language — and thus can be given a semantic interpretation. Syntactic and semantic correctness are mere prerequisites to be able to perform formal verification, but are often not regarded as formal verification.

Formal verification usually means that one is interested in the validity of some correctness statement about one or more formal specifications. This could for example be a completeness criterion (are all situations covered?), or a safety criterion (can something bad happen?). Often, a correctness statement involves another formal specification, at a higher level of abstraction. In that case, a verification problem consists of two specifications, one specifying requirements that the other must satisfy. The nature of the relation between the two specifications differs. It could be that one states a very high-level property that the other must satisfy, or that a refinement relation exists between the two specifications. The two specifications can use the same specification languages or could use two different specification languages having a common semantic foundation. As was stated already, this thesis is based on a dual language approach.

Two fundamental techniques to verification can be discerned, namely deductive verification and model-based verification. Deductive verification — often referred to as theorem proving — is the more traditional approach to verification. Using a set of axioms and a set of inference rules, formal reasoning is used to prove correctness of more abstract models or properties. For such reasoning, tool support — like for example PVS [96] or Isabelle [91] — is indispensable. The main disadvantage of deductive verification is that it requires a large amount of intensive user interaction. Model-based verification on the other hand, does typically not have this disadvantage. The basic idea is that one exhaustively inspects the model defined by a formal specification. In other words, the state space of a system is generated in some form and an algorithm is applied to verify properties on
that state space. The advantage is that verification becomes largely automated. On the other hand, the inherent limitation of this approach is in the explosion of the state space. State spaces of complex systems tend to be intractably large, often inhibiting exhaustive exploration. However, in the past decade techniques have appeared that have pushed the limit of these approaches up to a point where verification of industrial-size problems in some cases becomes feasible.

Usually, model-based verification approaches apply a model-oriented specification language like timed automata [8] or Promela [64] to define system models, together with a property-oriented language (like LTL [94], CTL [38], or TCTL [2]) to specify desired properties of these systems. As an alternative to this heterogeneous approach, model-based verification can also be applied in a homogeneous fashion to verify refinement relations between two (model-oriented) specifications. Usually the term model checking is associated with model-based verification in general, but sometimes model checking is also used to only denote the temporal logic based approach.

The two approaches to formal verification are often considered as complementary, each having its own balance between level of automation and complexity of verification problems. Also, combining the two approaches seems to be a promising direction to arrive at integrated more powerful verification approaches. For example, a theorem prover can be used to derive abstract models which can then be checked using a model checker.

Note that the two types of verification have analogues in commonly used approaches to (non-formal) verification of computer systems. The ultimate form of non-formal model-based analysis is testing, where the system itself is analysed, rather than a model. Another model-based technique is simulation, which does operate on a model of the actual system. In that sense, the difference between between formal and traditional (non-formal) approaches is not that big — model checking can be seen as exhaustive simulation. Counterparts of deductive verification are for example analytical approaches to schedulability analysis and performance analysis.

Tooling is an indispensable aspect of any industrially useful formal method. Formal verification without tool support implies that it relies on correctness of human reasoning and documentation, which would be much in conflict with the goals of formal methods. Therefore, sophisticated tool support can be seen as an enabler for formal verification. For model checking this is very obvious, since this approach is founded on the idea of huge amounts of repetitive calculations.

Furthermore, when developing practical model checking techniques it is essential to also build implementation for these techniques. This allows one to evaluate performance in terms of time and data resources. For model checking techniques — because of their heuristic character — this is an important aspect which is hard to analyse without any implementation of the techniques. The model checking approach discussed in this thesis has also been implemented in a tool.

### 1.1.4 Specifying real-time systems and properties

In this thesis, focus is on model checking in the context of real-time systems. There are many definitions for the terms real-time and real-time systems. In this thesis, we use the following rather simplistic definition for a real-time system: a system in which quantitative time plays a relevant role. For our purposes, this means that quantitative time aspects are relevant for our verification problem. In contrast, the term non-real-time will be used to indicate that the ordering of events may be relevant, but not their timing.
For real-time systems, system and property specification formalisms are needed that are able to model quantitative time-bounded behaviour. In the field of model checking real-time systems, timed automata have become more or less a standard approach to specification. Timed automata where introduced in 1990 [7], and have been used extensively in modelling various sorts of real-time systems — usually with the intention of verification. The timed automaton formalism is an extension of finite automata with clocks. Clocks are variables ranging over nonnegative real values that measure the progress of time. The semantics of timed automata enforces that all clocks increase their value at the same rate. Clocks can be reset upon transitions and constraints on clock values can be used as guards on transitions. Timed automata are based on a so-called dense time model, because clocks are real-valued variables that increase in a continuous manner. This contrasts with discrete time models, which use clocks that progress with discrete time ticks.

Several variations of the original formalism exist, many of them extending the expressiveness of the formalism in various ways. Also, there are differences in underlying semantic models and concrete syntax. Timed automata formalisms are a typical example of a specification notation that is focussed on verification. This is emphasized by the fact that the expressiveness of a particular timed automaton variant is often a direct result of the capabilities of the model checking approach for which it is intended to serve as input language. The work in this thesis uses a variant of timed automata called XTG[12].

For model checking such timed automaton specifications, a notion of correctness is needed. This can simply be a reachability property: Can some state be reached within some time? Is there is possibility of deadlock? For more complex properties a property specification language is needed. For property specification in model checking techniques, temporal logic has traditionally been a standard approach. Temporal logics extend standard logics with means to express qualitative timing aspects, i.e. the order in which events occur. Since their introduction into computer science [94], temporal logics have been widely used, both as system specification language, as well as property specification language. Several temporal logics exist, but from the perspective of model checking, there are two main types of temporal logic, namely linear-time temporal logic and branching-time temporal logic. Both have been successfully applied in model checking, each being associated with a particular "school" in temporal logic model checking.

As an example, a temporal logic property could express that whenever \( p \) becomes true then inevitably \( q \) must always become true while in the meantime \( r \) must hold. In the temporal logic CTL this is expressed as \( AG(p \Rightarrow (r \text{ AU } q)) \). CTL, introduced in [38], is a branching-time temporal logic often used in model checking.

Temporal logics like CTL have been extended with capabilities to express quantitative timing aspects. While standard temporal logics only focus the order of events, real-time temporal logics also deal with the time at which these events occur. As an example, we could extend the previous example by also demanding that \( q \) must become true within 10 time-units from the moment that \( p \) became true. This is expressed in TCTL — an extension of CTL — as \( AG(p \Rightarrow z:=0. (r \text{ AU } (q \land z \leq 10))) \). TCTL is the property language that will be the used in the work in this thesis.

### 1.1.5 Model checking

Model checking [40] was introduced in the early eighties [38, 95]. The central idea behind it is to verify whether or not a system model satisfies a property by exploring its complete set of states. By examining all of a system’s behaviour, one can decide whether
or not it behaves conform specified properties. Initially, the capabilities of model checking algorithms were limited — only systems with relatively small state spaces could be checked. However, in the past two decades many new concepts have been introduced in the field, making model checking more and more feasible for dealing with realistic verification problems.

The main challenge of model checking is that non-trivial systems have very large, often infinite state spaces. Inspecting each state explicitly is usually infeasible, no matter how resourceful the available computer is. Therefore, the main challenge in applying model checking is in finding algorithms that allow exhaustive evaluation of a state space, without explicitly enumerating all its elements. Somehow the state space of a system has to be reduced to a finite evaluation space that can be evaluated within available resources. To be of any use, such a reduction must be proven formally correct, that is, evaluating the reduced space must lead to conclusions that are valid for the original state space. Obviously, a reduced evaluation space would have to be derived directly from the model, without generating the original state space first.

Several approaches to realizing smaller state spaces have been investigated. First of all, there are reduction techniques that are focused on leaving out states that are not strictly needed because their presence can be seen as being "covered" by other states. A prominent example of this technique is partial order reduction [92, 52, 25]. Basically, partial order reduction exploits the fact that if the ordering of transitions is immaterial, then only one of the orderings needs to be taken into account. Another technique is symmetry reduction [65, 48, 25], where one exploits the symmetry that is present in many systems. This is often useful in case a system is a parallel composition in which at least two components are instantiations of the same process. These reduction techniques are aimed at reducing the complexity that originates from parallel processes, either by reducing the expansion caused by interleaving (partial order reduction), or by reducing the expansion caused by process replication (symmetry reduction).

The second major approach to realizing reduced state spaces is based on the idea that equivalent states can be integrated into more abstract states. In that case more abstract models are built in which the abstract states represent sets of states of the original model. Such techniques are called often referred to as symbolic model checking techniques. A key aspect of these techniques is the availability of a representation for sets of states that does not require the enumeration of their elements. Thus, symbolic model checking is enabled by the availability of mechanisms for efficient representation and manipulation of symbolic state spaces. The earliest and probably also best-known application of symbolic model checking is the non-real-time finite-state model checking approach of [35, 83]. There, binary decision diagrams (BDD’s) [33] are used to represent state spaces of systems whose states and transition relation can be modelled in terms of Boolean formulas. In this field, model checking has been quite successful, especially in the field of hardware verification. In fact, the discovery of the BDD-based representation for finite symbolic state spaces led to the breakthrough of model checking as a useful technique. Using this representation, model checkers like SMV [35] were built that allowed the verification of quite large models.

BDD-based model checking is designed to reduce complexity of finite state spaces introduced by interleaving and finite data elements. Being limited to finite-state models, this approach is not applicable to formalisms that are based on a dense time model or that have infinite data elements. In this field also symbolic model checking approaches exist, but these are usually build around efficient representations for sets of data valuations. This second category of symbolic model checking is often applied in real-time model checking to
deal with the infinite state spaces caused by real-valued clocks, but the technique is equally effective in dealing with state-space complexity caused by variables. Here, symbolic state space representations like Difference Bound Matrices (DBM’s) [47] are used to represent abstract states defined by sets of constraints on data or clocks. The work in this thesis falls in this second category of model checking approaches.

Finally, state spaces can also be reduced by avoiding that irrelevant states are considered. Given a system model and a property, the model may have states that do not need be considered for deciding the satisfaction of that particular property. A very simple example of this idea is that when checking whether or not a state with some quality \( p \) is reachable, then there is no need to explore successors of a state for which it is known already that it has this quality \( p \). In general, this means that one would like to avoid approaches which first build a complete state space and then — as a separate step — evaluate the correctness notion. Also, one would like to avoid considering states that are not reachable from the initial states of the system. Model checking approaches that do so are sometimes referred to as local. Non-local algorithms are not able to check during exploration whether or not states are reachable or not, which means that also unreachable states are taken into account.

Sometimes techniques are to a large extent complementary, which means that such approaches can be combined. Also, note that the field of model checking is far more complex than what is described here. More elaborate discussions can be found in [40] and [68].

### 1.1.6 Key aspects of a symbolic model checker

One could distinguish three key aspects of any symbolic model checking technique. Although these aspects can not always be viewed in isolation, it is useful to consider them as different aspects of a symbolic model checking problem.

1. **State space exploration**: An algorithm is needed to explore and evaluate symbolic state spaces. Given a system specification — which is in fact an implicit description of a state space — such an algorithm explores the state space in a symbolic manner, i.e. by operating on sets of states rather than single states.

2. **Symbolic state space representation**: A symbolic exploration algorithm is only useful if an implicit representation for symbolic states is available. This representation should allow efficient storage and manipulation of symbolic state spaces. The availability and efficiency of such a representation is a very important issue for any symbolic model checking approach as it has great influence on the feasibility and performance of the model checking approach.

3. **State space evaluation**: To solve verification problems using a symbolic model checking approach, an evaluation technique is needed that can decide the verification problem by inspecting the symbolic state space. The evaluation mechanism should allow the model checker to decide properties as efficiently as possible. Ideally, properties are decided as soon as the portion of the state space that has been explored provides enough information to do so.

The above distinction between three aspects of symbolic model checking will be used throughout this thesis. Obviously, there are links between these aspects. In particular, the state space representation has to allow dealing with symbolic states as they result from the exploration approach. Also the evaluation approach will set requirements for the way state spaces are explored.
These key aspects of a model checking technique can be related to two key characteristics of a resulting model checker. One could say that a model checker can be characterized as follows.

1. **The type of verification problems the model checker is targeted at.** First of all, model checkers are usually focused on a category of verification problems, for example finite-state systems, or (dense) real-time systems. Furthermore, model checkers will differ in the richness of the verification problems that can be checked. This concerns for example the expressiveness of a temporal logic or the expressiveness of the data language that is used to manipulate data and clocks.

2. **The performance of the model checker.** This concerns performance both in terms of time as well as data resources. This relates to a central question in model checking: How well does a specific approach scale up to larger — real life — verification problems? The type of verification problems that can be handled is usually strongly related to the symbolic state space representation. More complex systems and properties tend to lead to more complex representations. This in turn leads to model checkers that require more data and time resources. The exploration approach is often relatively insensitive to the type of verification problem, but does influence the performance of the model checker. Finally, the expressiveness of the property language has a direct influence on the evaluation approach.

### 1.2 Scope of the thesis

This thesis focuses on the application of partition refinement techniques in model checking real-time systems. Partition refinement is an exploration approach that is based on iterative refinement of a partition. It builds a partition that can be used as an abstract model of the complete state space, such that the abstract model can be used to verify the property of interest.

In retrospective, the main objectives of the work that resulted in this thesis can be summarized as follows.

1. Investigate whether partition refinement is a valid approach to model checking real-time systems. Current successful techniques are all based on variations of symbolic forward or backward analysis. In this thesis it is investigated whether partition refinement is a valid alternative to these techniques.

2. Focus on model checking approaches for "complex" real-time model checking problems — more complex than reachability analysis on standard timed automata. In particular, this concerns parametric systems that allow more complex manipulations on clocks than resets, other data types besides clocks, arbitrary linear constraints, urgency, and TCTL properties with fairness constraints.

3. Investigate efficient state space representation approaches for partition refinement model checking. In current model checking techniques, specific representations are used for state space representation. Partition refinement techniques impose different requirements on state space representations, for which alternative approaches may be more suitable.

4. Develop a systematic approach to fair TCTL verification in partition refinement techniques. Usually, the way TCTL properties are evaluated is strongly linked to the exploration approach that is used. Here, evaluation of TCTL properties is investigated from a more generic perspective, from which specific evaluation approaches for partition refinement can be derived.
5. Design and implementation of a concrete algorithm for efficient partition-refinement based model checking targeted at real-time systems.

1.2.1 Contributions of this thesis

In this thesis a model checking algorithm for real-time systems based on partition refinement is developed. Below are brief discussions of our solutions for the three earlier identified key issues in symbolic model checking — partition refinement (the exploration approach), splitting trees (the state space representation) and TCTL verification (the evaluation approach). Also, attention is paid to the type of verification problems that the approach is targeted at.

We use the term verification problem for the combination of a system specification and a property specification for that system — in our case an XTG specification together with a TCTL specification. The aim was to use a relatively rich system and property language framework, allowing the specification of relatively ”complex” verification problems.

From verification point of view, essential aspects of XTG and TCTL are the specification of urgency, the inclusion of fairness conditions, the specification of parametric systems, and the fact that linear constraints and assignments are allowed. Specification of urgency — i.e. the ability to define that edges should be taken ”as soon as possible” — is very useful in modelling, but somewhat complicates a partition refinement model checking algorithm. Fairness constraints have quite some impact on the algorithm, although the approach to dealing with fairness is partly transferred from existing approaches. The ability to specify parametric verification problems is a very useful quality of a model checking tool, and proved not to introduce much additional complexity in a partition refinement approach. An interesting aspect of our approach is that we do not only allow parameters everywhere in system specification, but also in properties.

Finally, the data types allowed in verification problems are very strongly related to the verification approach. In principle, we could allow any data type, but restrictions are defined based on the verification tool that operates on the resulting specifications. In our case this results in two different restrictions, each for a different version of the tool that we developed — one for difference constraints and one for linear constraints. The ability to deal with linear constraints is very valuable. Many efficient model checkers are based on the restriction to difference constraints, while many verification problems require the representation of linear constraints. Furthermore, most non-trivial parametric verification problems require the capability to deal with linear constraints. We are not aware of any concrete partition refinement approach that allows verification of such a broad spectrum of verification problems.

In terms of the key aspects of a symbolic model checking technique, the state space exploration method of our approach is partition refinement. It is roughly based on the abstract minimal model generation algorithm of [24], but it also incorporates ideas from [77] and [97]. Time-abstraction bisimulation [105] is used as the notion of equivalence based on which abstract models are produced. Other partition refinement approaches in literature usually only define algorithms at an abstract level, still leaving room for a lot of design choices. Related to this is that hardly any partition refinement approach is actually implemented in a tool. We define a relatively detailed algorithm for partition-refinement based model checking. Based on relatively generic partition refinement concepts, a concrete model checking approach is developed, requiring the inclusion of a TCTL decision making approach, as well as a symbolic state space representation.
We apply partition refinement to TCTL verification. A generic approach is defined for deciding TCTL properties of systems using abstract models based on time-abstracting bisimulation. This is done by means of a transformation towards a CTL verification problem on the abstract model. In [61] also the idea of transformation from TCTL to CTL is used, but not in terms of the generic scope of abstract models generated by time-abstracting bisimulation. There, detailed region graph equivalence [2] is used as the basis for abstract models, which leads to much larger models.

Concerning the combination of TCTL model checking and partition refinement, probably the approach described in [104] — which was developed around the same time as ours — is closest to our approach. Concerning the reduction from TCTL to CTL their approach seems to be more efficient and simpler than ours. Specifically, using a smarter transformation from TCTL to CTL, the construction of a property graph can be avoided, resulting in a more elegant approach.

A generic approach to deciding CTL problems that is based on partially explored state spaces, is defined. An important quality of this approach is that it is optimal, meaning that given a CTL property and a partially explored state space, it should decide satisfaction or dissatisfaction of that property for any reachable state, whenever this is possible. It is generic because only a decision approach is defined, which can be applied to any partial state space, regardless of how it was derived. There is a vast amount of work on verification approaches for CTL [39, 83, 19]. However, in these approaches the way that CTL properties are decided is specific for that approach. Also, choices are often made that may be efficient for that model checking approach, but are not optimal in the sense discussed above. An interesting aspect is that we want to have decision criteria for partially explored state spaces. This allows one to reason about the decision approach that is to be actually implemented in a model checking technique. Then also efficiency considerations related to the evaluation of decision rules have to be taken into account.

This is particularly relevant for partition refinement approaches, because the construction of the abstract model is in general much more expensive than its exploration. Thus, early decision making can be used to avoid unnecessary construction of parts of the symbolic state space. For other symbolic approaches — in particular the standard, fixpoint-based approach [83] — this is not an issue because there decision making is strongly connected to the exploration approach.

Finally, a new approach to symbolic state space representation, called splitting trees, is described. It is a relatively simple approach to the representation of symbolic states identified by constraints on clocks and variables. The splitting tree representation fits naturally with partition refinement based model checking approaches. Usually, canonical states space representations are used in symbolic model checking. However, for partition refinement approaches, such canonical representations seem to be relatively inefficient. The reason for this is that for partition-refinement based exploration only a very limited set of operation on the symbolic state space is needed, which does not justify the effort of maintaining a canonical representation. Note that splitting trees had been used in model checking before [98, 109] to allow efficient propagation of splitting, but not for state space representation. In our approach, splitting trees have a dual use. Apart from the original usage for the splitting propagation, splitting trees are also used for the representation of symbolic states.

Summarizing, a systematic approach is taken to develop a partition refinement model checking approach for fair TCTL and XTG. For each of the three above identified key problems of symbolic model checking, solutions are developed. Subsequently, these three
aspects are joined to produce the model checking approach. Also, it is shown that the resulting model checking approach is roughly comparable with other approaches in terms of data and time performance. This means that partition refinement approaches can be regarded as a valid model checking approach.

Originally, our approach was based on a somewhat different technique called region product graphs [11]. This was partly based on the ideas of [98], although there the context was verification of a real-time extension of the mu-calculus. The work presented here can be seen as a further evolution of the ideas in [11], mainly in the TCTL verification aspect.

1.2.2 Relations to other work

A high-level overview of work that is relevant for the work in this thesis is given. In the appropriate chapters, more detailed discussions of related work can be found.

The modelling aspects discussed in this thesis (chapter 2) have their foundation in a broad range of work on timed automata and temporal logic. XTG is a variant of timed automata [8], which is probably closest to timed safety automata [63]. We added parametric specification in the spirit of [10], the possibility to define urgent edges (like in [18]) and a value-passing communication model [30]. Our version of TCTL is a generalization of timed CTL [63]. Additionally, fairness is introduced like in [62], and parameter variables are allowed.

Other approaches to partition refinement in real-time model checking have been described [4, 109, 105, 98]. In terms of these approaches, our approach combines elements of the latter three. The use of time-abstracting bisimulation as underlying equivalence is based on [105]. More elaborate discussions on time-abstracting bisimulations can be found in [106]. The idea of backward propagation of splits and values is inspired by [98].

As was noted already, our approach to TCTL verification is to some extent similar to that of [104] — although there a more efficient solution for the transformation to CTL is described. Our approach to decision making for CTL properties applies results concerning fixpoint characterization of CTL [38, 83] and fair CTL verification [39]. To our knowledge, similar fundamental approaches to (explicit) CTL decision making cannot be found in literature.

The usage of splitting trees for propagation of splitting was introduced in [109] and is also used in [98]. Our additional use of splitting trees for representation purposes is not found in other work, although there are some similarities with work on canonical approaches [47, 75, 85]. In particular, our reduction method for difference constraints is similar to the normalization algorithm for DBM’s [47].

There have only been a limited number of implementations of partition refinement-based model checkers — Kronos [74, 106] seems to be the only publicly available partition-refinement model checker for real-time systems. Other tools for real-time model checking include HyTech [60], Kronos [110] and Uppaal [74], of which the latter is most mature. These tools are all based on symbolic forward or backward reachability analysis, rather than partition refinement.

1.2.3 Organization of the thesis

Chapter 2 takes care of the necessary groundwork. The timed-automata-based formalism of XTG and a matching variation of TCTL is defined. Also, a common semantic model is defined (timed transition systems), and the two formalisms are given a meaning in terms
Chapter 1 Introduction

of this semantic model. Besides TCTL, also CTL is defined since it is needed for shaping the verification approach.

Chapter 3 studies elementary aspects of partition refinement, starting from the abstract minimal model generation algorithm of [24]. Time-abstracting bisimulation is defined as the equivalence notion based on which abstract models are generated and region graphs are introduced as a representation for symbolic models. Furthermore, the concept of stability is introduced to be able to characterize region graphs as correct abstract models. Finally, a refined partition refinement algorithm is presented in the context of reachability analysis. This algorithm can be seen as an intermediate step towards the TCTL model checking algorithm presented in chapter 6. It is a refinement of the abstract minimal model generation algorithm but is still not specific for our verification context (discussed in chapter 4) and our state space representation (discussed in chapter 5).

Chapter 4 addresses from a general perspective the problem of deciding TCTL properties on abstract models generated by time-abstracting bisimulation. It does not make any assumptions on the form of the algorithm in which such a decision approach would be applied. A two-step approach is applied. First the TCTL decision problem is reduced to a CTL decision problem, which primarily involves dealing with divergence as well as property variables including clocks. After that a generic solution for dealing with the CTL decision problem is presented.

Chapter 5 is dedicated to defining a state space representation that fits to a partition refinement approach. This includes finding efficient implementations of the manipulations on the symbolic state space as they turned up in the refined partition refinement algorithm of chapter 3, namely splitting and propagation. It is shown that a non-standard approach to state space representation exists that fits well to the requirements of partition refinement algorithms. A definition is given, together with definitions for the required state space operations. Also, possible improvements of the representation are discussed.

In chapter 6 the work of chapters 3 to 5 is combined resulting in a concrete TCTL model checking algorithm. First, the general TCTL evaluation approach of chapter 4 is used to define a concrete decision approach in terms of valuations and decision rules, which can be readily integrated in a partition refinement algorithm. Then the algorithm is described in pseudocode and its correctness is discussed in a semi-formal manner. The model checking algorithm has been implemented in two model checking tools called PMC and LPMC, which are also discussed in chapter 6. In particular, results of applying the tool to some benchmark verification problems are given and compared to the performance of other tools.

Finally, chapter 7 presents conclusions. Note that each chapter is closed with a discussion section, which contains some conclusions that are specific for that chapter.
Chapter 2

Modelling Real-Time Systems and Properties

To be able to perform verification, a formal description of systems and their properties is needed, as well as a common underlying semantic model. This chapter does the essential groundwork of formally defining systems and their properties. It defines a semantic model, a system specification language and a property specification language.

2.1 Basic models

The XTG language [12] that will be defined in the next section of this chapter combines the modelling of two quite different aspects of a system, namely its control characteristics and its data characteristics. In the abstract definition of XTG this separation will be kept as much as possible. Our primary focus is on control characteristics of the language. The form of the data language however will become relevant when considering model checking strategies for systems, as there is a strong connection between the expressiveness of the data language and the limitations of model checking techniques. Here, a limited abstract model of the data language is defined that is sufficient for concisely defining the operational part of the semantics of XTG.

Below some notation is listed that will be used in this thesis. $\mathbb{N}$, $\mathbb{R}$, $\mathbb{R}^{\geq 0}$ and $\mathbb{R}^{> 0}$ are used to denote the set of natural numbers, real numbers, non-negative real numbers and positive real numbers, respectively. $B = \{\text{true}, \text{false}\}$ denotes the set of Boolean logic values. We use $\langle x, y, \ldots \rangle$ to denote tuples. Dashes are used to indicate irrelevant values, as in $\langle -, x, -, - \rangle$, where only $x$ appears to be relevant. Furthermore

$\mathcal{P}(S) = \{ s \mid s \subseteq S \}$ denotes the power set of $S$

$S = \bigcup_{n \in \mathbb{N}} S^n$ denotes the set of tuples over $S$

2.1.1 The data model

At this stage, we want to make as few assumptions on the data model as possible. A general set of variables is assumed, which can have different types. Only the subset of clock variables has to be made explicit, to be able to conveniently deal with timing aspects. Since a dense time model is used, clocks are real-valued variables.
In the same spirit, value expressions are only defined abstractly. It is only assumed that an evaluation function for value expressions is available. At some point, we will want to be able to require that a value expression is a Boolean value expression. Therefore, the subset of Boolean value expressions is also made explicit.

**Definition 2.1 (data language).** For the data language an abstract set of variable identifiers and values is defined:

- \( V \): a set of (clocks and data) variables
- \( V_C \subseteq V \): the subset of clock variables
- \( \text{Val} \): possible values

Given a set of variables \( V \), we assume the following domains:

- \( \text{Expr}_V \): value expressions over the set of variables \( V \)
- \( \text{Bexpr}_V \subseteq \text{Expr}_V \): denotes the subset of Boolean expressions

For the semantics, we need assignments of values to variables, referred to as valuations.

**Definition 2.2 (valuations).** A valuation is a mapping from variables to values. If \( V \) is a set of variables, then \( \text{Env}_V \) denotes the domain of valuations over \( V \):

\[
\text{Env}_V = V \rightarrow \text{Val}
\]

We use \( \rho \) and \( \xi \), as well as primed and subscripted variants, to range over valuations. If \( \rho \in \text{Env}_V \) and \( \rho' \in \text{Env}_{V'} \), with \( V \cap V' = \emptyset \), then the union of two valuations, \( (\rho + \rho') \in \text{Env}_{V \cup V'} \), is defined as follows:

\[
(\rho + \rho')(v) = \begin{cases} 
\rho(v) & \text{if } v \in V \\
\rho'(v) & \text{if } v \in V'
\end{cases}
\]

**Definition 2.3 (evaluation functions).** Given the data language of definition 2.1, the following semantic evaluation functions are assumed to exist:

- \( \mathcal{V}_V \): \( \text{Expr}_V \rightarrow ((V \rightarrow \text{Val}) \rightarrow \text{Val}) \)
- \( \mathcal{T}_V \): \( (\text{Expr}_V \cup V) \rightarrow \mathcal{P}(\text{Val}) \)

The first evaluation function evaluates value expressions, while the second associates a type with each value expression and variable. Types are interpreted as sets of possible values. The following (mostly trivial) assumptions are made:

- Type correctness of value expressions: \( \forall e \in \text{Expr}_V . \forall \rho \in (V \rightarrow \text{Val}) . \mathcal{V}_V[e](\rho) \in \mathcal{T}_V[e] \).
- Boolean value expressions evaluate to Boolean values: \( \forall b \in \text{Bexpr}_V . \forall \rho \in (V \rightarrow \text{Val}) . \mathcal{T}_V[b] = \mathbb{B} \).
- The set of values contains at least the nonnegative real values (needed for clocks) and Boolean values (needed for Boolean expressions): \( \{ \mathbb{R}^{\geq 0} \cup \mathbb{B} \} \subseteq \text{Val} \).
- Clocks are nonnegative real-valued variables: \( \forall v \in V_C . \mathcal{T}_V[v] = \mathbb{R}^{\geq 0} \).

Finally, the ‘\&’ operator on \( \text{Bexpr} \)’s is assumed, having the expected meaning \( \mathcal{V}_V[b_1 \land b_2] \rho = \mathcal{V}_V[b_1] \rho \land \mathcal{V}_V[b_2] \rho \).

In our modelling languages, changes to valuations are modelled using updates. An update is a set of assignments, an assignment being a tuple \( \langle v, e \rangle \) denoting the assignment of the value of an expression \( e \) to a variable \( v \).
Definition 2.4 (updates). An update \( u \) is an element of \( \mathcal{P}(V \times \text{Expr}_V) \) where \( V \) and \( V' \) are variable sets, such that \( \forall (v_1, e_1), (v_2, e_2) \in u . v_1 \neq v_2 \). The application of an update \( u \) to a valuation \( \rho \), denoted \( \rho[u] \), is defined as follows:

\[
\rho[u](v) = \begin{cases} 
\mathcal{V}_V[e] \rho & \text{if } (v, e) \in u \\
\rho(v) & \text{otherwise}
\end{cases}
\]

Note that assignments in an update are not interpreted as sequential actions, but as simultaneous actions.

To be able to express progress of time we will be needing the following function.

Definition 2.5 (time progress). Let \( V \) be a set of variables, with a subset \( V_C \subseteq V \) of clock variables. Then \( \rho[+\delta] \) denotes the application of a function \( \text{Env}_V \times \mathbb{R} \rightarrow \text{Env}_V \) that increases each clock in \( V \) by \( \delta \):

\[
\rho[+\delta](v) = \begin{cases} 
\rho(v) + \delta & \text{if } v \in V_C \\
\rho(v) & \text{otherwise}
\end{cases}
\]

Also \( \rho[-\delta] \) is written as a shortcut for \( \rho[+(\,-\,\delta)] \).

Thus given a valuation \( \rho \), \( \rho[+\delta] \) gives the values derived from \( \rho \) by "letting time pass" for \( \delta \) time units.

The term zone is used to denote a set of valuations.

Definition 2.6 (zones and respect). Given a set of variables \( V \), a zone is an element of \( \mathcal{P}(\text{Env}_V) \). A zone \( Z \in \mathcal{P}(\text{Env}_V) \) respects a Boolean expression \( b \in \text{Bexpr}_V \) if

\[
\{ \rho \in Z \mid \mathcal{V}_V[b] \rho = \text{true} \} \in \{ Z, \emptyset \}
\]

Thus a zone respects a Boolean value expressions if either all valuations in that zone satisfy the value expression or all valuations do not satisfy the value expression.

We will often want to characterize Boolean expressions by the set of valuations that satisfy these expressions, using the concept of characteristic sets.

Definition 2.7 (characteristic set). Given a boolean expression \( e \in \text{Bexpr}_V \), the characteristic set of \( e \), denoted as \( [e]_V \), is a zone, defined as follows:

\[
[e]_V = \{ \rho \in \text{Env}_V \mid \mathcal{V}_V[e] \rho = \text{true} \}
\]

Also \( [\neg e]_V \) is used as a shorthand for \( \text{Env}_V \setminus [e]_V \).

For notational convenience we will mostly use \( [e] \) rather than \( [e]_V \), if \( V \) is evident from the context.

2.1.2 Labelled transition systems

Our underlying model for real-time systems is what we will refer to as timed transition systems. In literature other names can be found for similar models, e.g. timed (Kripke) structures and labelled timed transition systems. Our interpretation of a timed transition system can be defined as a special type of a so-called labelled transition system. The latter is defined below. Although a separate definition for labelled transition systems is not needed for the semantics of our formalisms, we do present a definition here, because this concept will be needed in the description of our verification approach.
Definition 2.8 (labelled transition system). A labelled transition system is a tuple \( M = \langle S, S_0, T \rangle \), where

- \( S \) is a set of states
- \( S_0 \subseteq S \), with \( S_0 \neq \emptyset \) is a subset of initial states
- \( T \subseteq S \times Lab \times S \) is a transition relation where \( Lab \) is a set of transition labels.

The notation \( s^{lab}_{T}s' \) is used to indicate a transition \( \langle s, s' \rangle \in T \). Labels are sometimes omitted, which has the following meaning: \( s \rightarrow_{T} s' \iff \exists lab \in Lab . s^{lab}_{T}s' \).

Finally, \( lab_{T}^{*} \) denotes the transitive closure of \( lab_{T} \), and \( \rightarrow_{T}^{*} \) denotes the transitive closure of \( \rightarrow_{T} \). If \( T \) is clear from the context it may be omitted.

Here, the states of a labelled transition system are only abstractly defined. The actual interpretation of states depends on the concrete model for which a labelled transition system provides semantics. As will be shown later, for XTG states are interpreted as tuples consisting of a control component and a valuation component for variables.

In literature, different variations of labelled transition systems can be found. These all have in common that there is a set of states together with a set of transitions between states. Initial states are not always made explicit. Also, some models have a single initial state, while others — like ours — define a set of initial states. The reason for this is that we want to perform parametric verification. Then one needs sets of initial states that are differentiated by parameter values.

Furthermore, there is often a way to define observable behaviour. This can be done by labelling states with atomic properties and/or by labelling transitions with action labels. Our model does have labelled transitions, but atomic properties have not been modelled as part of the labelled transition systems. When dealing with labelled transition systems in an abstract manner, atomic properties are simply defined as subsets of the set of states.

As a consequence, a state \( s \in S \) satisfies an atomic property \( p \subseteq S \) if \( s \in p \). Therefore, the set of possible atomic properties can be characterized as \( AP = \mathcal{P}(\mathcal{P}(S)) \). When states have concrete interpretation, atomic properties become value expressions, usually over the control and data state of a system. Appropriate evaluation functions are then needed for interpreting such atomic properties.

The following functions are used to allow convenient reasoning over predecessors and successors.

Definition 2.9 (\textit{succ} and \textit{pred}). Let \( T \subseteq S \times Lab \times S \) be a transition relation on a set of states \( S \) and a set of labels \( Lab \). Then let \( succ \) and \( pred \) denote functions such that for any \( s \in S \) and any \( SS \subseteq S \),

\[
\begin{align*}
\text{succ}_{T}(s) & = \{ s' \in S \mid s^{i}_{T}s' \} \\
\text{pred}_{T}(s) & = \bigcup_{l \in Lab} \text{succ}_{T}^{l}(s) \\
\text{succ}_{T}(SS) & = \bigcup_{s \in SS} \text{succ}_{T}^{l}(s) \\
\text{pred}_{T}(s) & = \bigcup_{l \in Lab} \text{pred}_{T}^{l}(s) \\
\text{pred}_{T}(SS) & = \bigcup_{s \in SS} \text{pred}_{T}^{l}(s)
\end{align*}
\]

If clear from the context \( T \) is often omitted.

When reasoning about systems, we are often interested in the set of paths that it generates.

Definition 2.10 (paths). Given a labelled transition system \( M = \langle S, S_0, T \rangle \), a path is a possibly infinite sequence of transitions

\[
\psi : s_0^{lab_0}_{T}s_1^{lab_1}_{T} \cdots
\]

Chapter 2  Modelling Real-Time Systems and Properties
Given a path $\psi$ as above, $\psi(i)$ denotes the sub-path of $\psi$ that starts at $s_i$:

$$\psi(i) = s_i^{lab_i} \xrightarrow{T} s_{i+1}^{lab_{i+1}} \xrightarrow{T} \cdots$$

The set of paths of a labelled transition system $M$ is denoted as $\Psi_M$.

### 2.1.3 Timed transition systems

Labelled transition systems are used to reason about non-real-time systems. For real-time systems, timed transition systems are used, which can be seen as a specialization of labelled transition systems. The key characteristic of a timed transition system is that the passing of time is modelled by labelling transitions with nonnegative real numbers. As a consequence, two types of transitions exist: discrete transitions, which are given a label from a label set $Lab$, and time transitions, which are labelled with a nonnegative real value. This value models the amount of time that elapses when taking the transition.

**Definition 2.11 (timed transition system).** A timed transition system is defined as a tuple $\langle S, S_0, T \rangle$, where

- $S$ is a set of states
- $S_0 \subseteq S$ represents the subset of initial states
- $T \subseteq S \times (\mathbb{R}_{\geq 0} \cup Lab) \times S$ is a transition relation, where $Lab$ is a set of labels such that $T$ has the following properties:
  
  - For all $s, s', s'' \in S$ and for all $\delta \in \mathbb{R}_{\geq 0}$, if $s \xrightarrow{T} s'$ and $s' \xrightarrow{\delta} s''$ then $s' = s''$.
  
  - For all $s, s' \in S$ and for all $\delta, \delta' \in \mathbb{R}_{\geq 0}$, $s \xrightarrow{\delta + \delta'} s'$ if and only if for some $s'' \in S$, both $s \xrightarrow{\delta} s''$ and $s'' \xrightarrow{\delta'} s'$.
  
  - For each $s \in S$, $s \xrightarrow{T} s$.

Transitions labelled with a delay value are denoted as time transitions and transitions with a transition label from $Lab$ are denoted as discrete transitions. The fact that discrete transitions are given a label is not essential for our modelling and verification approach. In fact, all discrete edges could have been given a fixed label $\mu$. However, for modelling the verification approach, it is convenient to use these labels to refer to edges of the system from which the transition originates.

The three properties included in definition 2.11 do not play a great role in this thesis, since these will naturally follow from the modelling language that generate the semantic models. They mostly concern essential but common-sense qualities of time.

For formal verification, the notion of runs of a timed transition system is important. Runs can be seen as the real-time equivalent of paths.

**Definition 2.12 (runs of a timed transition system).** Given a timed transition system $M = \langle S, S_0, T \rangle$, a run $\pi$ of $M$ is an infinite sequence of transitions from $T$:

$$\pi : s_0 \xrightarrow{T} s_1 \xrightarrow{T} \cdots$$
\( \delta_\pi(i) \) denotes the delay associated with the \( i \)-th transition of a run \( \pi = s_0 \overset{lab_0}{\rightarrow}_T s_1 \overset{lab_1}{\rightarrow}_T \cdots \):

\[
\delta_\pi(i) = \begin{cases} 
0 & \text{if lab}_i \in \text{Lab} \\
lab_i & \text{if lab}_i \in \mathbb{R}^\geq 0
\end{cases}
\]

\( \Pi_M \) denotes the set of runs of \( M \), \( \Pi_M(s) \) denotes the set runs that start from \( s \). A position on a run \( \pi = s_0 \overset{\delta_0}{\rightarrow}_T s_1 \overset{\delta_1}{\rightarrow}_T \cdots \) is a pair \( \langle i, t \rangle \in \mathbb{N} \times \mathbb{R}^\geq 0 \), where \( t < \delta_\pi(i) \). Positions are ordered as follows:\n
\( \langle i, t \rangle \ll \langle i', t' \rangle \iff i < i' \lor (i = i' \land t \leq t') \).

The state at position \( \langle i, t \rangle \), denoted as \( \Xi_\pi(i, t) \) is defined as follows:

\[
\Xi_\pi(i, t) = s_i[+t]
\]

The elapsed time until \( \langle i, t \rangle \) on the run \( \pi \), denoted \( \Delta_\pi(i, t) \) is defined as follows:

\[
\Delta_\pi(i, t) = t + \sum_{j=0}^{i-1} \delta_\pi(j)
\]

The set of runs associated with a timed transition system can be seen as representing all possible behaviours of a system.

We will need the concept of reachability.

**Definition 2.13 (reachability).** Given two states \( s \) and \( s' \) of a timed transition system \( M = \langle S, S_0, T \rangle \), we say that \( s' \) is reachable from \( s \), written \( s \overset{\tau}{\rightarrow} s' \), if there exists a run \( \pi = s_0 \overset{\delta_0}{\rightarrow}_T s_1 \overset{\delta_1}{\rightarrow}_T \cdots \).

A state \( s \) is reachable, if there is a \( s_0 \in S_0 \) such that \( s_0 \overset{\tau}{\rightarrow} s \).

Given a timed transition system, runs can be constructed that do not match our expectations of how a real-time system can behave. When investigating properties of our system, we do not want to take into account such behaviours. In particular, the type of behaviour we want to ignore is non-divergence.

**Definition 2.14 (divergence).** A run \( \pi \) is divergent if and only if

\[
\forall d \in \mathbb{R}^\geq 0 . \exists \langle i, t \rangle . \Delta_\pi(i, t) = d.
\]

The set of all divergent runs of a timed transition system \( M \) is denoted by \( \bar{\Pi}_M \). \( \bar{\Pi}_M(s) \) denotes the set of divergent runs starting from \( s \).

By focussing on the set of divergent runs, we exclude those behaviours that do not progress in time. A related concept is that of non-zenoness.

**Definition 2.15 (non-zenoness).** A timed transition system \( M = \langle S, S_0, T \rangle \) is non-zeno if from each reachable state \( s \in S \) there is a divergent run \( \pi \in \bar{\Pi}_M(s) \). It is zeno if it is not non-zeno.

A non-zeno model is sound in the sense that it does not contain reachable states from which time cannot diverge. Preferably, we would only like to consider non-zeno models, since in zeno models time can become blocked, which does not correspond to any realistic behaviour. See [63] for more information on this subject.

An important aspect of runs and paths is fairness. The concept of fairness is used to exclude certain behaviour that is not considered to be realistic for the system that is being modelled. This concerns "un-fair" behaviour in which for certain non-deterministic choices, some transitions are consistently ignored. This can be avoided by requiring that specific states are visited infinitely often.
Definition 2.16 (fairness). Given a timed transition system \( M = \langle S, S_0, T \rangle \), a fairness property is a set of fairness predicates, where a fairness predicate is a subset of \( S \). Thus, a fairness property is an element of \( \mathcal{P}(\mathcal{P}(S)) \).

Let \( F \) be a fairness property. A run is \( F \)-fair if and only if for each \( f \in F \), there are infinitely many states \( s \) along the run such that \( s \in f \). For paths, fairness is defined in the same way. The set of all \( F \)-fair (and divergent) runs of a timed transition system \( M \) is denoted as \( \Pi^{F}_{\delta M} (\tilde{\Pi}^{F}_{\delta M}) \), while the set of \( F \)-fair paths of a labelled transition system is denoted as \( \Psi^{F}_{M} \). \( \Pi^{F}_{\delta M}(s) \), \( \tilde{\Pi}^{F}_{\delta M}(s) \) and \( \Psi^{F}_{M}(s) \) have the expected analogous meaning.

Here, fairness predicates are defined as sets of states. For concrete systems, fairness properties will be defined as predicates over states, like atomic properties.

2.2 XTG

XTG [11, 12] ("extended timed graphs") is a formalism for defining models of real-time parallel systems. It is based on the timed automaton formalism introduced by Alur and Dill [7]. The timed automaton formalism and its variations (e.g. [63, 73, 88]) have become a much-used formalism for modelling real time systems. XTG can be seen as an extension of a variation called timed safety automata [63].

2.2.1 A brief introduction to XTG

Essentially, an XTG is a finite state machine augmented with clocks and data. Clocks are non-negative real-valued variables that increase at the same fixed rate. The increasing of clocks models the progress of time. The execution of transitions of an XTG can be guarded and enforced by constraints on clocks and data. Guards define conditions under which an edge may execute, while location invariants state conditions under which control may reside at that location. The most common use of invariants is to enforce the departure from some location. Upon execution, a transition may update values of clocks and data.

As was mentioned already, several versions of timed automata have been proposed in literature, which differ mainly in timing semantics, communication/synchronization models, and capabilities of the data model. Compared to the timed safety automata of [63], XTG adds urgency, has more extended data modelling, has a more refined communication model and adds parameters.

The basic representation for XTG is in the form of a single automaton. Additionally, XTG systems are defined, which allow the specification of multiple automata — possibly representing parallel processes — which synchronize and exchange values. The communication model is based on the synchronous value passing model of value passing CCS [84, 30]. For example, an edge labelled with a synchronization \( l!e \) must synchronize with an edge labelled with \( l?v \), where \( l \) is a synchronization label (or channel name), \( e \) is value expression, and \( v \) is a variable. This results in a synchronized transition in which the value of \( e \) is assigned to the variable \( v \). In XTG this is generalized to multiple simultaneous value passings, possibly in both directions. This has proved to be quite useful in modelling scheduling approaches [27].

In principle, XTG does not commit to a specific data language. Only a few assumptions are made, one of them being that clocks have to be available. In practice, limitations will be put on the expressiveness of the data language that stem from the capabilities of the verification tool. This means for our case that we define two different levels of
Chapter 2 Modelling Real-Time Systems and Properties

expressiveness, each of them leading to specifications that can be used as input for two different corresponding tools. Note that data elements can be specific for an automaton in the system, or can be part of a set of shared data elements. Thus, shared data forms a second communication model, next to value-passing synchronizations, that can be used.

XTG provides a means for expressing urgency. Edges can be marked as urgent, indicating that these have to be executed immediately upon enabling — without letting time pass. This general form of urgency allows convenient modelling of edges that trigger on data or time conditions.

A general approach to parametric verification is chosen. A subset of the (non-clock) variables of the system is designated as parameters. These parameters can be used anywhere in the system specification (i.e. in guards, invariants, and updates) and the property specification (i.e. in atomic properties). The only concrete modelling difference between parameters and normal variables is that parameters are not given initial values. Our approach to modelling is based on the ideas applied in HyTech [59, 10].

In figure 2.1, an example XTG system is depicted. It models two aspects of a simple coffee machine, namely an input component that allows a user to request coffee (the left-hand automaton) and the part of the coffee machine that outputs the coffee (the right-hand automaton). The user (which is not shown, the system shown here is incomplete) requests a coffee, by pushing the coffee button, which is modelled as a synchronization on coffee!. Subsequently, it can increase the strength of the coffee by pushing the stronger button once or several times, depending on the required strength. Initially the strength — modelled by the real variable x — is equal to 1, while it can at most be 5. Each push on the button increases the strength with one, until the maximum is reached in which case the button is disabled (modelled by the guard \( x < 5 \)). If for \( p \) time-units, the strength was not increased, it triggers the dispenser component to pour the coffee. This is modelled by a synchronization on the pour channel, passing the strength of the coffee to the dispenser. The clock c keeps track of time, and since the edge is urgent, once \( p \) time units have passed, the pour command is immediately issued. When synchronizing on pour, the dispenser automaton moves to the second location, recording the strength value in y. The second location models the actual pouring of the coffee with strength y, which is not detailed here. The only thing that is known is that the time it takes to pour the coffee is dependent of its strength, but — due to not modelled factors in pouring the coffee — is not exactly known. This is modelled by an invariant-guard combination, constraining the value of the clock x := x + 1
\[ x < 5 \]
\[ c \geq p \]
\[ d := 0 \]
\[ d < 3y \]
\[ d > 2y \]

Figure 2.1: An example XTG system
2.2.2 An abstract syntax for XTG

This section defines an abstract syntax for XTG. Note that to be useful, the abstract syntax has to be supplemented with a concrete syntax and a mapping from this concrete syntax to the abstract syntax. The concrete syntax is the basis for tool-based verification. For XTG such a concrete syntax is defined, which also addresses the typing of variables. In the descriptions here this is abstracted away by means of the data language model of definition 2.1. The concrete syntax is however not considered relevant for the discussions in this thesis. Also, as was shown in figure 2.1, there is an (incomplete) convenient graphical representation for XTG, which is very useful for illustration purposes. It is used in this thesis to present XTG examples.

Definition 2.17 (XTG). An XTG is a tuple \( \langle V, V_C, V_P, \rho_0, L, l_0, I, E, U \rangle \), where

- \( V \) defines a set of variables
- \( V_C \subseteq V \) defines a set of clock variables
- \( V_P \subseteq (V \setminus V_C) \) defines a set of parameters
- \( \rho_0 \in \text{Env}(V \setminus V_P) \) defines an initial valuation for non-parameter variables
- \( L \) is a set of location identifiers
- \( l_0 \in L \) identifies the initial location
- \( I : L \rightarrow \text{Bexpr}_V \) assigns an invariant to each location
- \( E \subseteq L \times \text{Bexpr}_V \times \mathcal{P}(V \times \text{Expr}_V) \times L \) is a set of edges. An edge is a tuple \( \langle l, g, u, l' \rangle \), where
  - \( l \in L \) is the source location,
  - \( g \in \text{Bexpr}_V \) is a guard,
  - \( u \in \mathcal{P}(V \times \text{Expr}_V) \) is an update, and
  - \( l' \in L \) is the destination location.
- \( U : E \rightarrow \mathbb{B} \) identifies the subset of urgent edges that satisfies the following two properties:
  - \( \forall \langle l, g, u, l' \rangle \in E . \forall \langle v, e \rangle \in u . T_V[v] = T_V[e] \)
  - Each edge \( e = \langle l, g, u, l' \rangle \in E \) is urgency-safe, which means that
    \[ \forall \rho \notin Z_e . (\forall \delta > 0 . \rho[+\delta] \notin Z_e) \lor (\exists \delta > 0 . \rho[+\delta] \in Z_e \land \forall \delta' . 0 < \delta' < \delta \Rightarrow \rho[+\delta'] \notin Z_e) \]
    where \( Z_e = \{ \rho \in [g] | \rho[u] \in [I(l')] \} \) is the enabling condition for \( e \).

The first condition ensures type correctness of assignments. The reason for the urgency-safety restriction is that an urgent edge with a non-strict lower time-bound does not have a useful meaning. Consider for example a location with a single outgoing urgent edge with a guard \( c > 1 \), where \( c \) is a clock (see figure 2.2). Suppose that currently \( c \) has the value 0. The meaning of urgency is that an edge is taken as soon as possible, but such a moment does not exists for our example. Whatever moment is chosen, there is always a moment an earlier moment that also satisfies the guard. This would mean that the edge is never
taken, and that time is blocked. So the system of figure 2.2 is zeno, since time cannot progress. In the example, changing the guard to $c \geq 1$ would make the edge urgency-safe.

In fact, the condition formulated in definition 2.17 is a bit too strict. One can think of situations that do not satisfy the condition but which do not cause a problem. On the other hand, in practice it is more useful to introduce clear constraints. The following straightforward conditions ensure urgency-safety:

- strict lower clock bounds on guards of urgent edges are not allowed.
- strict lower clock bounds on invariants of locations with incoming urgent edges are not allowed.

where a lower clock bound is a constraint that puts a lower bound on values of one or more clocks. Note that these conditions are even more strict than those in definition 2.17.

Concerning the modelling of parametric systems, there is in fact only a single difference between parameters and non-parameter variables, which concerns the initial valuation $\rho_0$. Parameters are not assigned initial values, while non-parameter variables do have initial values. Obviously, parameters cannot be clocks.

In this thesis we will most of the times use $c$ and $d$ to denote clocks, $x, y, z$ to denote non-clock variables, and $v$ to denote values.

Below, two restricted forms of XTG are defined, which will be used later to distinguish between different verification problems.

**Definition 2.18 (simple XTG and linear XTG).** A *simple XTG* is an XTG that satisfies the conditions below.

- Besides clocks, it only has variables for integers, real and enumerations.
- Constraints on integer and real variables are limited to difference constraints, i.e. constraints of the form $x - y \sim c$ or $x \sim c$, with $\sim \in \{<, \leq\}$, where $x$ and $y$ are real variables and $c$ is a value. Updates on integer and real variables are limited to $x := y + c$ or $x := c$, where $x$ and $y$ are either both integer or real variables and $c$ is a constant.
- Clocks and real variables (including parameters) are not mixed in a single constraint or update.

A *linear XTG* is an XTG that satisfies the following conditions.

- Besides clocks, it only has variables for integers, reals and enumerations.
- Constraints and updates on integer variables are constrained as above.
- Constraints on real variables are limited to linear inequalities. Expressions in updates on real variables are limited to linear expressions.
2.2.3 Operational semantics for XTG

The operational semantics of XTG is defined by a mapping from XTG specifications to timed transition systems. This results in timed transition systems in which states are not abstract like in definition 2.11, but are given an interpretation in terms of the locations and variables of the XTG it originates from. More precisely, states are tuples \( \langle l, \rho \rangle \), where \( l \) is an XTG location and \( \rho \) is a valuation for the XTG’s variables.

Definition 2.19 (operational semantics of XTG). The operational semantics of an XTG \( \langle V, V_C, V_P, \rho_0, L, l_0, I, E, U \rangle \) is a timed transition system \( \langle S, S_0, T \rangle \) where

- \( S = \{ \langle l, \rho \rangle \in L \times Env_V \mid V[I(l)](\rho) \} \)
- \( S_0 = \{ \langle l_0, \rho \rangle \mid \forall v \in (V \setminus V_P). \rho(v) = \rho_0(v) \} \).
- \( T \subseteq S \times (E \times \mathbb{R}_{\geq 0}) \times S \) such that for any \( e = \langle l, g, u, l' \rangle \in E \) and \( \{ \langle l, \rho \rangle, \langle l', \rho[u] \rangle \} \subseteq S \):

\[
\langle l, \rho \rangle \stackrel{e}{\longrightarrow} T \langle l', \rho[u] \rangle \iff V_V[g](\rho)
\]

and for any \( \delta \geq 0 \) and any \( \{ \langle l, \rho \rangle, \langle l', \rho[+\delta] \rangle \} \subseteq S \):

\[
\langle l, \rho \rangle \stackrel{\delta}{\longrightarrow} T \langle l, \rho[+\delta] \rangle \iff \forall d \cdot 0 \leq d < \delta \Rightarrow (V_V[I[l]](\rho[+d]) \text{ and } \langle l, \rho[+d] \rangle \notin S_{\text{urg}})
\]

where \( S_{\text{urg}} = \{ \langle l, \rho \rangle \in S \mid \exists \langle l, g, u, l' \rangle \in U. (V_V[g](\rho) \text{ and } V_V[I[l']](\rho[u])) \} \)

The set of states is formed by the set of all location-valuation combinations for which the valuation satisfies the invariant of the location. The set of initial states is the subset of the states corresponding to the initial location and for which the non-parameter variables have their initial values. If there are no parameters in the XTG, then \( S_0 \) is a singleton set.

The definition of \( T \) shows that there is a discrete \( e \)-labelled edge between two states if the valuation of the origin satisfies the guard of \( e \) and the valuation of the destination is the result of applying the update of \( e \) to the valuation of the origin. There is a \( \delta \)-labelled time edge between two states if the valuation of the destination edges properly records the passing of \( \delta \) time units, and all states that are "passed" while increasing the clocks with \( \delta \) satisfy the invariant of the location of the states and do not have enabled urgent outgoing edges. \( S_{\text{urg}} \) denotes the set of states from which at least one urgent edge is enabled. From such states, no time edges can depart. Note that zero-delay transitions are always allowed.

From the semantics definition follows immediately that the required properties of timed transition systems (definition 2.11) hold if the XTG has at least one clock. If an XTG has no clocks, then time is irrelevant.

We extend the \( \rho[+\delta] \) operation to states, such that \( \langle l, \rho \rangle[+\delta] = \langle l, \rho[+\delta] \rangle \).

2.2.4 XTG systems

An XTG system is formed by the parallel composition of a set of potentially communicating XTG’s. The semantics of such a parallel composition can be defined in two compatible ways. First, XTG systems could be directly interpreted in terms of timed transition systems. Such a definition could then replace the semantics of XTG as was introduced above. The second approach is to define a translation from a system of XTG’s to a single XTG. Both approaches can convey the same semantics, but the latter is more convenient for describing our verification approach. The concept of XTG systems is essential for convenient modelling but hardly plays a role in the discussions on the verification approach, which is primarily defined in terms of (single) XTG’s.
An XTG system is defined as a set of XTG’s, together with a set of shared data variables and a set of communication channels between the individual XTG’s. To be able to define the communication channels, the concept of value passing expression is defined.

Definition 2.20 (value passing expression). Let \( \text{lab} = \{l_1, l_2, \ldots\} \) be a set of communication labels, and let \( \overline{\text{lab}} = \{\overline{l}_1, \overline{l}_2, \ldots\} \) denote a set of complementary labels. Let \( V \) denote a set of variables. A value passing expression is a tuple \( \langle c, ia, oa \rangle \) where

- \( c \in \text{lab} \cup \overline{\text{lab}} \) identifies a communication channel,
- \( ia \in \mathcal{V} \) is a possibly empty tuple (see section 2.1 of variables, and
- \( oa \in \mathcal{E}^{\text{Expr}} \) is a possibly empty tuple of value expressions over \( V \).

Let \( \text{VP} \) denote the set of possible value passing expressions, and \( \text{VP}_V \) those that range over variable set \( V \).

Two communication labels are referred to as complementary, if one is an overlined version of the other (thus \( l \) and \( \overline{l} \) are complementary).

In concrete syntax, a value passing expression \( \langle l, \langle v_1, v_2, \ldots \rangle, \langle e_1, e_2, \ldots \rangle \rangle \) is written as \( l?v_1?v_2\ldots!e_1!e_2\ldots \), where \( v_1, v_2, \ldots \) denote variables and \( e_1, e_2, \ldots \) denote value expressions. Most commonly, value expressions only transfer a single value or no value at all. In the latter case, they become pure synchronizations. Value passing expressions come with a notion of direction, implemented by the label names. Only value passing expressions with complementary labels can be matched for actual communication, for example \( \text{coffee}!3 \) and \( \text{coffee?strength} \), where \( \text{strength} \) is a variable.

Definition 2.21 (XTG system). An XTG system is a tuple \( X = \langle \text{GV}, \text{GV}_C, \text{GV}_P, gp_0, G, C \rangle \), where

- \( \text{GV} \) is a set of global variables
- \( \text{GV}_C \subseteq \text{GV} \) is a set of global clock variables
- \( \text{GV}_P \subseteq (\text{GV} \setminus \text{GV}_C) \) is a set of global parameters
- \( gp_0 \in \text{Env}(\text{GV} \setminus \text{GV}_P) \) defines the initial values for the global non-parameter variables
- \( G = \langle g_1 \ldots g_2 \rangle \) is a tuple of XTG’s
- \( C : \mathcal{E} \to (\text{VP} \cup \{\bot\}) \), where \( \mathcal{E} = \bigcup_{\langle v_1, \ldots, v_n, e, \ldots \rangle \in G} E \), such that for each \( \langle v_1, \ldots, v_n, e, \ldots \rangle \in G \), for each \( e \in E \), \( C(e) \in \{\bot, \text{VP}_V\} \).

such that for each \( e, e' \in \mathcal{E} \) with \( C(e) = \langle l, \langle v_1, \ldots, v_n \rangle, \langle e_1, \ldots, e_m \rangle \rangle \) and \( C(e') = \langle l', \langle v'_1, \ldots, v'_n' \rangle, \langle e'_1, \ldots, e'_m' \rangle \rangle \rangle \), if \( l \) and \( l' \) are complementary, then \( n = m' \) and \( m = n' \) and \( \forall i \in \{1, \ldots, n\} \cdot \mathcal{T}_V[v_i] = \mathcal{T}_V[e'_i] \) and \( \forall i \in \{1, \ldots, m\} \cdot \mathcal{T}_V[e_i] = \mathcal{T}_V[e'_i] \).

Thus, an XTG system is defined by a global state \( \text{GV} \), a set \( G \) of single XTG’s, and a function \( C \) assigning value passing expressions to some of the edges of the graphs. If \( C(e) = \bot \) then no value passing is associated with \( e \). The final constraint in the definition only serves to ensure that value expressions with matching labels have matching types. We assume that the identifiers used for locations and local variables are globally unique.

On several occasions in this thesis we want to extend an XTG system \( X \) with an additional automaton that does not communicate with the XTG’s in \( X \). This simple form of extension is formalized below.
Definition 2.22 (extending XTG systems). Given an XTG system \( X = \langle GV, GV_C, GV_P, gp_0, \langle g_0 \ldots g_n \rangle, C \rangle \) and an XTG \( g \), the extension of \( X \) with \( g = \langle L, \ldots, L, E, \perp \rangle \) is defined to result in the XTG system \( X' = \langle GV, GV_C, GV_P, gp_0, \langle g_0 \ldots g_n, g \rangle, C' \rangle \), where

\[
C'(e) = \begin{cases} 
\perp & \text{if } e \in E \\
C(e) & \text{otherwise}
\end{cases}
\]

If \( l = \langle l_1 \ldots l_n \rangle \) is a location of the global graph (see definition 2.24) corresponding to \( X \), and \( l' \) is a location of \( g \), then we let \( l + l' \) denote the location \( \langle l_1, \ldots, l_n, l' \rangle \) of the global graph corresponding to \( X' \).

Continuing with the semantics for XTG systems, we require the following definition.

Definition 2.23 (synchronizations). Let \( vp_1 = \langle l, \langle v_1 \ldots v_n \rangle, \langle e_1 \ldots e_m \rangle \rangle \in VP_V \) and let \( vp_2 = \langle l', \langle v_1' \ldots v_n' \rangle, \langle e_1' \ldots e_m' \rangle \rangle \in VP_V \). Then the function \( sync(vp_1, vp_2) \in (P((V \cup V') \times Expr(V \cup V'))) \cup \{\perp\} \) is defined as follows

\[
sync(vp_1, vp_2) = \begin{cases} 
\bigcup_{i \in \{1, \ldots, n\}} \langle v_i, e_i' \rangle \cup \bigcup_{i \in \{1, \ldots, m\}} \langle v_i', e_i \rangle & \text{if } l \text{ and } l' \text{ are complementary} \\
\perp & \text{otherwise}
\end{cases}
\]

\( sync(vp_1, vp_2) \) returns \( \perp \) if \( vp_1 \) and \( vp_2 \) do not match, which is the case if the synchronization labels are not complementary. If the two value passing expressions match, then an update is produced that is the result of combining the two expressions. Note that in that case it follows from definition 2.21, that \( n = m' \) and \( m = n' \).

Definition 2.24 (semantics of parallel composition). Given an XTG system \( X = \langle GV, GV_C, GV_P, gp_0, \langle g_0 \ldots g_n \rangle, C \rangle \) with \( g_i = \langle \{V_i, V_{C_i}, V_{P_i}, \rho_0, L_i, l_0, I_i, E_i, u_i \} \rangle \), the global graph corresponding to \( X \) is an XTG \( \langle V, V_C, V_P, \rho_0, L, l_0, I, E, U \rangle \), where

- \( V = \bigcup_{i=1}^{n} V_i \cup GV \)
- \( V_C = \bigcup_{i=1}^{n} V_{C_i} \cup GV_C \)
- \( V_P = \bigcup_{i=1}^{n} V_{P_i} \cup GV_P \)
- \( \forall v \in V. \rho_0(v) = \begin{cases} \rho_0(v) & \text{if } v \in V_i. i \in \{1, \ldots, n\} \\
g_0(v) & \text{if } v \in GV \end{cases} \)
- \( L = \prod_{i=1}^{n} L_i \)
- \( l_0 = \langle l_0_1 \ldots l_0_n \rangle \)
- \( \forall \langle l_1 \ldots l_n \rangle \in L. I_i(l_1 \ldots l_n) = \bigwedge_{i \in \{1, \ldots, n\}} I_i(l_i) \)
- \( E \) and \( U \) are defined as follows. For any \( i, j \in \{1, \ldots, n\} \), for any \( urg \in B \),

\[
\exists e_1 = \langle l_i, g, u, l'_i \rangle \in E_i . \quad C(e_1) = \perp \quad \text{and} \quad U(e_1) = urg \iff \exists e = \langle \{l_1 \ldots l_n\}, g, u, \{l'_1 \ldots l'_n\} \rangle \in E . \quad \forall k \in \{\{1, \ldots, n\} \setminus \{i\}\} \implies l_k = l'_k \quad \text{and} \quad U(e) = urg
\]

\[
\exists e_1 = \langle l_i, g_1, u_1, l'_i \rangle \in E_i . \quad \exists e_2 = \langle l_j, g_2, u_2, l'_j \rangle \in E_j . \quad \exists e = \langle \{l_1 \ldots l_n\}, g, u, \{l'_1 \ldots l'_n\} \rangle \in E . \quad \forall k \in \{\{1, \ldots, n\} \setminus \{i, j\}\} \implies l_k = l'_k \quad \text{and} \quad u = u_1 \cup u_2 \cup sync(C(e_1), C(e_2)) \quad \text{and} \quad g = g_1 \land g_2 \quad \text{and} \quad U(e) =urg
\]
The definitions of $E$ and $U$ deserve some explanation. An edge in the global graph originates either from one edge of one of the constituent graphs or — as a consequence of synchronization — from two matching edges from two different graphs. In the first case, the original edge must not have a value passing expression associated with it, since edges with a value passing expression are required to synchronize. The resulting global edge is then given the guard, the update and the urgency attribute from the local edge. In case the edge is the result of a synchronization, the two value passing expressions must have matched. Then the guard of the global edge is the conjunction of those of the local edges. The update of the global edge is a combination of the updates of the local edges and the update that results from the synchronization. The global edge is urgent, if either one of the local edges is.

An XTG system is urgency-safe if the resulting global graph is urgency-safe. The convenient condition that was formulated in section 2.2.2 does not straightforwardly carry over to XTG systems, because of the effects of synchronization. The following conditions on XTG systems are sufficient, but more restrictive than the conditions of section 2.2.2.

- strict lower clock bounds on guards of edges are not allowed.
- strict lower clock bounds on invariants of locations are not allowed.

The difference is that the conditions now apply to all edges, rather than just the urgent ones, due to the fact that a non-urgent edge may synchronize with an urgent edge.

2.3 TCTL

Our property specification language TCTL is a real-time temporal logic, based on CTL. CTL (Computation Tree Logic) [39] is a branching time logic, that has been the basis for many model checking approaches. TCTL (Timed CTL) was introduced in [2] as an extension of CTL that is able to deal with quantitative time. Several versions of TCTL have been introduced (e.g. [2, 63]). These are mostly comparable, although some minor differences exist. Our version of TCTL can be seen as an (minor) extension of that of [63].

2.3.1 A brief introduction to TCTL

The core syntax of TCTL defines two temporal operators, $A\phi$ and $E\phi$. The formula $\phi_1 \land A\phi_2$ is satisfied in a state if for all computation paths starting from that state, there is a state along it which satisfies $\phi_2$, and until that time $\phi_1$ is satisfied. $\phi_1 \land E\phi_2$ is satisfied if there is at least one such computation path. Derived — and more commonly used — operators are: $EF\phi$ (There is path on which there is state satisfying $\phi$), $EG\phi$ (There is a path on which every state satisfies $\phi$), $AF\phi$ (On all paths there is some state satisfying $\phi$), and $AG\phi$ (On all paths every state satisfies $\phi$). Figure 2.3 illustrates the latter four operators. In this figure, fragments of computation trees are shown — black circles represent states that satisfy $p$, white circles represent states that do not.

Real-time extensions of CTL either augment temporal operators with time bounds, or introduce so-called reset quantifiers. An example formula of the first approach is $AF_{\leq 10} p$, expressing that $p$ will always become true sometime within ten time-units. Using a reset quantifier the same property can be expressed as $z.AF(p \land z \leq 10)$, where $z$. implicitly expresses that $z$ is set to zero at the beginning of each path. We use a similar approach but generalize the reset quantifier to assignments to specification clocks and variables. In our TCTL variant, the example specification would become $z := 0.AF(p \land z \leq 10)$. 

A secondary benefit from this approach is that we can use symbolic constants in our property formulas (as suggested in [56]). By introducing variables that are not part of the system specifications, additional properties can be expressed. Consider for example, the TCTL formula $AG (t := count. AF (count = t + 1))$ in which $count$ is a system variable and $t$ is property specification variable. It expresses that at any moment, $count$ will always become incremented at some future moment.

Like for example in [62], we will interpret TCTL under fairness conditions. This means that the semantics of TCTL will only consider fair runs of the system. Thus, in fact the version of TCTL we use could be termed as fair TCTL, analogous to fair CTL [39].

### 2.3.2 Syntax and semantic of CTL

Before discussing TCTL, first the syntax and semantics of CTL is defined. CTL will be needed in the definition of the model checking approach in the coming chapters. We present a fully abstract syntax and semantics for (fair) CTL. It is defined purely on labelled transition systems, without any reference to the specification that generated the labelled transition system. Thus, the states are uninterpreted.

**Definition 2.25 (CTL syntax).** Given a labelled transition system $\langle S, S_0, T \rangle$, CTL formulas are generated by the following grammar:

$$\phi ::= p \mid \phi \lor \phi \mid \neg \phi \mid \phi AU \phi \mid \phi EU \phi \mid EX \phi$$

where $p \in \mathcal{P}(\mathcal{P}(S))$ denotes an atomic property.

**Definition 2.26 (CTL semantics).** Let $M = \langle S, S_0, T \rangle$ be a labelled transition system, $F$ a fairness property for $M$ and $\phi$ be a CTL formula for $M$. Then for a state $s \in S$, $s \models^F_M \phi$ indicates that $s$ satisfies $\phi$ under the fairness condition $F$. $s \models^F_M \phi$ is inductively defined as follows ($\Psi^F_M(s)$ was defined in definition 2.10):

- $s \models^F_M p \iff s \in p$
- $s \models^F_M \phi_1 \lor \phi_2 \iff s \models^F_M \phi_1$ or $s \models^F_M \phi_2$
- $s \models^F_M \neg \phi_1 \iff \neg s \models^F_M \phi_1$
- $s \models^F_M \phi_1 AU \phi_2 \iff$ for all paths $\psi \in \Psi^F_M(s)$, $\psi \models^F_M \phi_1 U \phi_2$
- $s \models^F_M \phi_1 EU \phi_2 \iff$ there exists a path $\psi \in \Psi^F_M(s)$, such that $\psi \models^F_M \phi_1 U \phi_2$
- $s \models^F_M EX \phi \iff$ there exists an $s' \in \text{succ}_T(s)$ such that $s' \models^F_M \phi$

where

- $\psi \models^F_M \phi_1 U \phi_2 \iff$ there is a state $s_i$ on $\psi = s_0 \rightarrow s_1 \rightarrow \cdots$ such that $s_i \models^F_M \phi_2$ and for all $j \in \{0, \ldots, i - 1\}$, $s_j \models^F_M \phi_1$
2.3.3 Syntax and semantics of TCTL

In the previous section, CTL was defined as a property language for abstract labelled transition systems. Its syntax and semantics only refer to the labelled transition system it is defined on, without referring to any aspects of a modelling formalism that generated it. This is expressed by the fact that the atomic properties and fairness properties are simply sets of states of the labelled transition system. However, for the use TCTL as property language it is essential to take into account the fact that the timed transition system is generated by an XTG model. This means that we take into account the structure of the states of a timed transition system that is generated by an XTG model. The atomic proposition $p$ in the above syntax is replaced by expressions evaluated on the structure of states. Two types of such expressions are defined:

- Boolean value expressions that specify conditions on the value component of a state, and
- location expressions that specify conditions on the location component of a state. The latter are of the form $g@l$, and express the fact that the graph $g$ of the system is currently at location $l$.

For the first, syntax and semantics is simply that of the Boolean value expressions as defined in definition 2.1. For location expressions, some new structures are needed.

**Definition 2.27 (location expressions).** Let $X = (\tau, \cdot, \cdot, (g_1 \ldots g_n), \cdot)$ be an XTG system with $g_i = (\cdot, \cdot, \cdot, L_i, \cdot, \cdot, \cdot, \cdot)$. A location expression for $X$ is a construct $g@l$, where $g_i \in \{g_1, \ldots, g_n\}$ and $l \in L_i$. Let $LE_X$ denote the set of possible location expressions for $X$. Given an XTG system $X$, we define the evaluation function for location expressions $L_X : LE_X \rightarrow (\prod_{i=1}^n L_i \rightarrow B)$ as follows.

$$L_X[g@l](\langle l_1 \ldots l_n \rangle) = \begin{cases} \text{TRUE} & \text{if } \exists i \in \{1, \ldots, n\}. g_i = g \text{ and } l_i = l \\ \text{FALSE} & \text{otherwise} \end{cases}$$

A location expression evaluates whether or not for a graph in an XTG system control is at some specified location. Thus, $g@l$ will be true if for graph $g$ control is at location $l$.

Now that models are available for the atomic properties of our property specification language, we can define the syntax and semantics of TCTL.

**Definition 2.28 (TCTL syntax).** Let $X$ be an XTG system and $G = (V, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot)$ the global graph corresponding to $X$. Then a TCTL specification for $X$ is a tuple $\langle PV, PV_C, PV_P, \phi \rangle$, where $PV$ is a set of property variables, having a subset $PV_C \subseteq PV$ of clock variables and a subset $PV_P \subseteq (PV \setminus PV_C)$ of parameters, and where $\phi$ is a TCTL formula. Valid TCTL formulas for $X$ are generated by the following syntax:

$$\phi ::= \text{le} \mid b \mid \phi \lor \phi \mid \neg \phi \mid \phi \ AU \phi \mid \phi EU \phi \mid u. \phi$$

where

- $\text{le} \in LE_X$
- $b \in Bexpr_{(V \cup PV)}$ denotes a Boolean value expression ranging over variables from the system as well as those of the property.
- $u \in \mathcal{F}(PV \times Expr_{(V \cup PV)})$ denotes a property update, which is an update that assigns values to property variables and of which the expressions range over system and property variables.
such that whenever a property variable \( x \in PV \) occurs in a Boolean expression or in the right-hand side of an update, it must be in the scope of a property update that binds \( x \) to some value. \( \square \)

In TCTL update operators have precedence over temporal operators and temporal operators have precedence over Boolean operators. In concrete syntax we write property updates defined by a single assignment as \( x := c. \phi \), while non-singleton property updates are written as \( \{ x := e_1, y := e_2 \}. \phi \).

The transformation of derived CTL and TCTL operators to core syntax is defined by the following rules.

\[
EF \phi = \text{true} \ AW \phi \quad AG \phi = \neg EF \neg \phi \\
AF \phi = \text{true} \ AU \phi \quad EG \phi = \neg AF \neg \phi \\
AX \phi = \neg EX \neg \phi \\
\phi_1 \land \phi_2 = (\neg \phi_1) \lor (\neg \phi_2) \quad \phi_1 \Rightarrow \phi_2 = (\neg \phi_1) \lor \phi_2
\]

Like atomic properties, fairness properties have to be made specific for the modelling language. The definition below gives a syntax for fairness properties for XTG and defines fair runs of an XTG. It is an XTG-specific interpretation of definition 2.16.

**Definition 2.29 (fairness for XTG).** A fairness property for an XTG system \( X \) is a set of fairness predicates, where valid fairness predicates are generated by:

\[
f := le \mid \neg le \mid b
\]

where, \( le \in LE_X \) and \( b \in Bexpr_V \), \( V \) being the set of variables of the global graph corresponding to \( X \).

A run \( \pi \) of a timed transition system \( M \) corresponding to an XTG system is fair with respect to a fairness property \( F = \{ f_1, \ldots, f_n \} \), denoted \( F \)-fair, if for each \( f_i \in F \), there are infinitely many states \( s \in \pi \), such that \( s \models f_i \). \( \square \)

Fairness predicates coincide with the atomic properties of TCTL. Therefore we can also use the satisfaction relation \( \models \) for fairness predicates.

The definition below gives the semantics for TCTL, which is similar to those found in literature and partly based on [62] and [111].

**Definition 2.30 (TCTL semantics).** Let \( M = \langle S, S_0, \tau \rangle \) be a timed transition system generated by an XTG \( G \) which is the global graph corresponding to an XTG system \( X \), and let \( F \) be a fairness condition for \( X \). Let \( \langle PV, PV_C, PV_P, \phi \rangle \) be a TCTL specification, and let \( \langle l, \rho \rangle \in S \) be a state from \( M \). Then we write \( \langle l, \rho \rangle \models^F_M \phi \) to denote that \( \phi \) is satisfied at state \( \langle l, \rho \rangle \) of \( M \), given fairness condition \( F \).

Then \( \langle l, \rho \rangle \models^F_M \phi \) if and only if \( \langle l, \rho \rangle \models^F_{M, \xi} \phi \), where \( \xi \in Env_{PV} \) is an arbitrary initial valuation for the property variables and the satisfaction relation \( \langle l, \rho \rangle \models^F_{M, \xi} \phi \) is inductively defined as follows (\( \Delta_\pi(i, t) \) was defined in definition 2.12):

\[
\langle l, \rho \rangle \models^F_{M, \xi} b \quad \iff \quad \forall_V [b](\rho + \xi) = \text{true} \\
\langle l, \rho \rangle \models^F_{M, \xi} le \quad \iff \quad L_X[l](\rho + \xi) = \text{true} \\
\langle l, \rho \rangle \models^F_{M, \xi} \phi_1 \lor \phi_2 \quad \iff \quad \langle l, \rho \rangle \models^F_{M, \xi} \phi_1 \lor \langle l, \rho \rangle \models^F_{M, \xi} \phi_2 \\
\langle l, \rho \rangle \models^F_{M, \xi} \neg \phi_1 \quad \iff \quad \not\langle l, \rho \rangle \models^F_{M, \xi} \phi_1 \\
\langle l, \rho \rangle \models^F_{M, \xi} \phi_1 AU \phi_2 \quad \iff \quad \text{for all } \pi \in \Pi^F_{M}(\langle l, \rho \rangle), \pi \models^F_{M, \xi} \phi_1 \ U \phi_2 \\
\langle l, \rho \rangle \models^F_{M, \xi} \phi_1 EU \phi_2 \quad \iff \quad \text{there exists a } \pi \in \Pi^F_{M}(\langle l, \rho \rangle) \text{ such that } \pi \models^F_{M, \xi} \phi_1 \ U \phi_2 \\
\langle l, \rho \rangle \models^F_{M, \xi} u. \phi \quad \iff \quad \langle l, \rho \rangle \models^F_{M, \xi}[u] \phi
\]
where
\[
\pi \models^F_{M, \xi} \phi_1 U \phi_2 \iff \text{there exists a position } (i, t) \text{ such that }
\Xi_{\pi}(i, t) \models^F_{M, \xi + \Delta_{\pi}(i, t)} \phi_2 \text{ and for all } (i', t') \ll (i, t),
\Xi_{\pi}(i', t') \models^F_{M, \xi + \Delta_{\pi}(i', t')} \phi_1 \lor \phi_2
\]

We may write \(|-\), \(\models_M\) or \(\models^F\) instead of \(\models^F_{M}\) if the meaning is clear from the context.

To be able to deal with the property variables of TCTL, an additional valuation \(\xi\) is introduced for hold values for these variables. Having such a property variable valuation, the semantics of the update operator becomes evident. Given a property valuation \(\xi\), \(u. \phi\) is satisfied in case \(\phi\) is satisfied given an updated property valuation in which the update \(u\) has been applied.

The semantics of \(\phi_1 U \phi_2\) deserves some comments. First of all, it has to ensure that the property specification clocks are increased with the amount of time that elapses. Whenever a subproperty (\(\phi_1\) or \(\phi_2\)) is evaluated against a state along a path then \(\xi\) has to be updated such that the clocks in \(\xi\) reflect the time that has elapsed. This is done by adding \(\Delta_{\pi}(i, t)\) to each of the clocks. The reason that \(\phi_1\) or \(\phi_2\) (rather than just \(\phi_1\)) has to be satisfied until \(\phi_2\) is satisfied lies is the dense-time model that is applied. For example, if we would only require \(\phi_1\) to hold until \(\phi_2\) is satisfied, then the property \(c := 0. (c \leq 1 AU c > 1)\) would not hold. See [63].

The term characteristic set of a property will be used to denote the set of states that satisfy that property, as defined below.

**Definition 2.31 (characteristic set).** The characteristic set of a TCTL formula \(\phi\) for a timed transition system \(M = (S, S_0, T)\) denotes the subset of states of \(M\) that satisfy \(\phi\) under the fairness condition \(F\):

\[
\models_F^M = \{ s \in S \mid s \models_F^M \phi \}
\]

If their interpretation is clear from the context, the subscript and superscript may be omitted. A similar definition was already given for expressions (definition 2.7).

TCTL is interpreted over the set of divergent runs \(\Pi_M\). The need for a divergence semantics can be illustrated by means of some simple examples. In the first verification problem of figure 2.4 one would expect the property not to hold, since the system has to leave the left-hand location immediately. Without a divergence semantics this would not be the case because there is a non-divergent run that stays in the first location forever.

![Figure 2.4: Illustrating the relevance of divergence](image-url)
The second example is similar. Here the divergence semantics is needed to enforce that the edge to the third location is eventually taken. The run that loops in through the first two locations is not divergent since no more than two time-units can elapse along it.

The semantics defined above defines a satisfaction relation for TCTL formulas on states of a timed transition system. But since we are aiming at parametric verification, we are looking for a semantics that states the relation between parameter values and the satisfaction of TCTL properties. This is provided by the semantics definition below, which is based on that of definition 2.30.

**Definition 2.32 (semantics of TCTL for XTG).** Let \( X \) be an XTG system, and \( G = (V, V_C, V_P, \rho_0, L, I_0, I, E, U) \) the corresponding global graph. Let \( M \) be the timed transition system corresponding to \( G \), and let \( F \) be a fairness property for \( X \). Furthermore, let \( \text{prop} = (P V, PV_C, PV_P, \phi) \) be a property specification for \( X \). Furthermore, given a valuation \( \nu \), let \( \nu|_V \) denote the restriction of \( \nu \) to the variables in \( V \), i.e. \( \nu|_V \in \text{Env}_V \) and \( \forall v \in V . \nu|_V(v) = \nu(v) \).

A parameter valuation \( \nu \in \text{Env}(V_P \cup PV_P) \) is a solution for \( X \) and \( \text{prop} \) if for an arbitrary \( (\xi, \rho_0 + \nu|_V) \models^F \text{prop} \). The solution set for \( X \) and \( \text{prop} \) is the set of all solutions. An expression \( \text{sol} \in Bexpr(V_P \cup PV_P) \) is a solution expression for \( X \) and \( \text{prop} \) if it describes the solution set \( Q \), i.e. if \( [\text{sol}](V_P \cup PV_P) = Q \).

Thus the solution set holds all the parameter valuations that lead to a (non-parametric) XTG system that satisfy the property, while a solution expression is a constraint on the parameters that characterizes the solution set.

### 2.4 Discussion

In this chapter we defined modelling formalisms for model-based and property-based specification of real-time systems, as well as formal models that enable formal reasoning over specifications. In doing so, we installed the basic structure for the coming chapters. Many aspects of what was presented here are based on known results from literature. XTG can be seen as a generalization of timed safety automata [63]. Our timed automaton variation has some important properties which make it a useful tool in automatic verification. Among other things, it allows the modelling of a generic form of urgency, which proved to be quite useful in specifying verification problems. Urgency is now a concept that is available in many model checking tools [58]. Also, we have chosen a simple but powerful approach to modelling parametric systems, based on ideas applied in HyTech [10, 59]. Alternative — more restricted — ways of modelling parametric systems can be found in [108, 22]. In [21] parametric verification is applied to hybrid automata, where parameters are also allowed in the specification of the rates of continuous variables. In [32] an approach to parametric property specification is presented that is more generic than ours (although systems are not parametric).

Although XTG can be conveniently presented in graphical format, the basic idea of our work has been that the language should be seen as an intermediate language focused on verification. Experiments were done with translations from other more user-oriented specification languages. In [81] a transformation from the real-time process algebra MTCCS [103, 70] to an earlier version of XTG was defined. The transformation defined there included the expansion of parallelism, leading to single automata. Also, work was done on translation of ASTRAL [51] specifications to an earlier version of XTG [27, 28]. Here, focus was on scheduling analysis, inspired by the work found in [41]. Finally, work
was done on translating formal JAVA-like specifications to XTG [71, 69]. Here, focus was on using model checking verification to analyse real-time programs.

The XTG formalism is defined in a general way, in the sense that it allows the inclusion of any data model that can be described using a denotational semantic model. Thus, the semantics of XTG is the result of a merging of two semantic models. The control structure is given meaning by an operational model, as is usual for timed automaton variants. The data manipulation model — although not much elaborated on in this thesis — is defined by a denotational semantics model. The advantage of this approach is that the definition of the operational model does not need many assumptions on the form of the data model. Only its abstract interface is defined. Other approaches either leave exact semantics implicit (most of the timed automaton approaches), or embed the data model in the operational semantics (for example [72, 55]). The latter approach may not always be natural when dealing with rich data models.

As was noted already, in our case limitations on the data model will usually stem from the verification tool to which the specifications are input. We identified two 'subsets' of XTG that are relevant for our verification approach. Chapter 5 comes back to this.

Concerning the operational part of the language, we learned that it would be convenient to have some more constructs to define systems of automata. In our concrete syntax we already provide some mechanisms for non-parametric instantiation of automata and relabelling and hiding of communication channels in the spirit of CCS [84]. However, it would be useful to have a more elaborate mechanism for instantiating automata. A simple extension would be to be able to define a set of automata, together with a specification of how instantiations are made of these automata and how these instantiations are composed to an XTG system. For example, given some XTG $G$ that synchronizes through $\text{out}$! and $\text{in}$? labels and operates on two variables $x$ and $y$, one could specify a composition as follows:

$$
( ( G[\text{out} \mapsto \text{chan1}, \text{in} \mapsto \text{chan2}, x \mapsto x_1, y \mapsto g] || \\
G[\text{out} \mapsto \text{chan2}, \text{in} \mapsto \text{chan1}, x \mapsto x_2, y \mapsto g]) \setminus \{\text{chan1}, \text{chan2}\} ) || (...)$$

The chosen syntax is not very sophisticated, but shows the use of such an extension. The expected effect of above shown fragment would be that the in and out synchronizations are coupled to two communication channels, which are hidden from possible other automata (through $\setminus\{\text{chan1}, \text{chan2}\}$), and which have private variables $x_1$ and $x_2$ and have a shared variable $g$.

A useful semantic extension is the general modelling of continuous variables. While clocks progress uniformly with time, continuous variables can progress at any rate, depending on the location of the automaton. This leads to what is usually called hybrid automata [6, 5]. XTG has also been extended to deal with continuous variables, but this was not shown here.

Our version of TCTL is a generalization of timed CTL in [63]. The notion of property clocks and reset operators in [63] is in our approach replaced by variables in general (including clocks) and general assignments to these variables. The extension with fairness is fairly standard, see for example [62]. Finally, our version of TCTL also allows the specification of parameters in properties, which is in fact similar to the approach in [32].
Chapter 3

Partition Refinement

3.1 Introduction

This section describes at an abstract level the symbolic exploration algorithm that is incorporated in the model checking approach developed in this thesis. It is based on the minimal model generation algorithm, introduced in [24]. The central idea behind it is to reduce the infinite state space of a system to a finite model, while still allowing the evaluation of properties that were specified over the original, infinite model. Many states in the state space are equivalent, or more precisely, their distinction is not relevant for verifying that the state space satisfies or dissatisfies the property of interest. Such equivalent states can be grouped into symbolic states, which are more abstract entities that represent sets of states of the original model. In this way smaller models are obtained, of which exploration is hoped to be feasible. To identify equivalent states, the notion of bisimulation [84] is used.

Given a model $M$ and a property $\phi$ (this could be simple reachability or a temporal logic formula), we are looking for an abstract model $M'$ and a property $\phi'$, such that for each state $s$ of $M$ an abstract state $\text{abs}(s)$ of $M'$ exists such that

$$s \models_M \phi \iff \text{abs}(s) \models_{M'} \phi'$$

This means that the verification problem is transformed to a simpler verification problem. Note that in some cases, $\phi'$ can be equal to $\phi$, while in other cases $\phi'$ has to be derived from $\phi$ by some transformation. To be of any use, $M'$ would have to be simpler than $M$. At the least, the part of $M'$ that is needed to prove or disprove $\phi'$ should be finite.

The basic idea of the minimal model generation algorithm is partition refinement. Starting from an initial partition of the state space, this partition is iteratively refined aiming to arrive at a partition that respects the required equivalence relation. The resulting partition then defines the abstract model we are looking for. Note that a partition of a set $S$ is a set of disjoint subsets of $S$ of which the union is $S$.

For our purposes, the minimal model generation algorithm has to be instantiated in the context of temporal logic model checking of real-time systems. Extensions of the minimal model generation algorithm to real-time systems are also described in [4] and [105]. In [109] an algorithm is presented for real-time systems that is based on a similar partition refinement approach. Like our approach, these extensions all build on notions of time-abstracting bisimulation, which could be seen as extensions of bisimulation concepts.
towards real-time systems. Our approach uses the notion of (strong) time-abstracting
bisimulation introduced in [105], and also includes some of the ideas introduced in [98].

This chapter can be seen as a first step towards the model checking algorithm that
will be described in chapter 6. It explores basic concepts, bisimulation and partition
refinement. It defines region graphs as a realization of the above mentioned abstract
models. It also gives a first partition refinement algorithm for reachability analysis. The
subsequent chapters build on this algorithm with the intention to arrive at an efficient
TCTL model checking algorithm based on partition refinement.

First, section 3.2 discusses the basic minimal model generation algorithm that is the
starting point of our approach, together with the bisimulation concept on which it is based.
Subsequently, section 3.3 extends these concepts to real-time systems, still without refer-
ence to concrete XTG and concrete models. Section 3.4 then defines concrete structures
that provide a means to realize abstract models based on XTG specifications. Based on
that, section 3.6 describes a first model checking algorithm for parametric reachability
analysis. In between, section 3.5 illustrates some of the concepts by means of a small
example.

3.2 Bisimulation and minimal model generation

At this point, we abstract from how this algorithm translates into a concrete algorithm for
partitioning timed state spaces that are generated by XTG specifications. We first discuss
the general algorithm using labelled transition systems. Given a labelled transition system
\( \langle S, S_0, T \rangle \) and a set of atomic properties, we build models that are induced by partitions
of the state space \( S \). Classes of these partitions represent sets of states that must have the
same behaviour with respect to the property to be verified. The concept of bisimulation,
defined below, is used to capture this idea of behavioural equivalence.

3.2.1 Bisimulation

Bisimulation was first described by Park [90]. Two states are bisimilar if they have the
same observables and they have equivalent transitions to equivalent states. For our pur-
poses we define bisimulation as follows.

**Definition 3.1 (bisimulation).** Let \( M = \langle S, S_0, T \rangle \) be a labelled transition system and
\( AP \in \mathcal{P}(\mathcal{P}(S)) \) a corresponding set of atomic properties. A relation \( B \subseteq S \times S \) is a
bisimulation on \( M \) if and only if for all \( \langle s_1, s_2 \rangle \in B \):

- \( \forall p \in AP . s_1 \in p \iff s_2 \in p \)
- \( \exists s'_1 \in S, \exists l \in \text{Lab} . s_1 \overset{l}{\rightarrow} s'_1 \implies \exists s'_2 \in S . s_2 \overset{l}{\rightarrow} s'_2 \text{ and } \langle s'_1, s'_2 \rangle \in B \)
- \( \exists s'_2 \in S, \exists l \in \text{Lab} . s_2 \overset{l}{\rightarrow} s'_2 \implies \exists s'_1 \in S . s_1 \overset{l}{\rightarrow} s'_1 \text{ and } \langle s'_1, s'_2 \rangle \in B \) \( \square \)

Thus, two states are bisimilar if they are equivalent with respect to the relevant atomic
properties and each transition from one of these states can be simulated by an equally
labelled transition such that both destination states are bisimulation-equivalent.

Our interest in bisimulation is its use for generating abstract models. A bisimulation
relation induces a partition of the state space, grouping states that are bisimilar. Based on
such partitions, abstract models can be defined of which states are bisimulation equivalence
classes of the original model. If we can prove that such abstract models preserve the ability
to verify the property of interest, then bisimulation can be used as a basis for reducing
models to simpler ones. For example, abstractions based on bisimulation preserve CTL properties [29]. This means that if one can construct a bisimulation-based abstract model of a state space, then the verification of a CTL property on that state space can be reduced to the problem of verifying that property on the smaller abstract model. As will be shown, for TCTL things are more complex.

Note that bisimulations can also be defined as relations $B \in S_1 \times S_2$ over states of different models. These can then be used to reason about equivalences between models.

When using bisimulation in the context of model checking, the choice of the atomic properties would most likely follow from the property that is to be verified. For example, in case of reachability analysis, there would be one atomic property consisting of those states to which reachability is to be checked and one that captures the initial states. When verifying a temporal logic formula, the set of atomic properties of that formula would be a likely candidate.

Given a model $M$, many abstract models exist — even $M$ itself is a bisimulation-abstract model of $M$. Naturally, we would aim at an abstract model that is as small as possible. We are therefore interested in the partition induced by the largest bisimulation relation. In other words, we are looking for the coarsest partition which corresponds to a bisimulation relation.

To be able to define an algorithm that derives partitions induced by a bisimulation relation, the concept of stability is needed.

**Definition 3.2 (stability).** Given a labelled transition system $\langle S, S_0, T \rangle$, and a partition $\rho$ of $S$, we say that a class $X \in \rho$ is stable with respect to a class $Y \in \rho$, if all states in $X$ have equally labelled transitions to states in $Y$:

$$\forall l \in \text{Lab} . \ ((\exists s \in X . \exists s' \in Y . s \xrightarrow{l} s') \Rightarrow (\forall s \in X . \exists s' \in Y . s \xrightarrow{l} s'))$$

A class is stable with respect to $\rho$ if it is stable with respect to all classes in $\rho$, a partition $\rho$ is stable if all classes containing reachable states are stable with respect to $\rho$.

If all classes in a partition are stable and respect all atomic properties, then the partition defines a bisimulation relation. This follows directly from above definition and the definition of bisimulation.

### 3.2.2 The minimal model generation algorithm

The minimal model generation algorithm [23, 24] is a general algorithm for computing the partition induced by a greatest bisimulation relation. It is a so-called partition refinement algorithm, because it computes this partition by means of iterative refinement.

A key characteristic of the algorithm is that abstract models are generated that are as small as possible. More precisely, given an initial partition, it computes the coarsest partition that refines the initial partition and corresponds to a bisimulation relation. A second important characteristic of the algorithm is that it is a local algorithm, which means that it takes into account the initial states of a system. The algorithm focuses on that part of the state space that is reachable from the initial states. This contrasts with global algorithms, which would also refine unreachable classes. This is a waste of effort, since unreachable states have no meaning for the verification problem. The third important characteristic is that it can operate from an implicit definition of a model, thus without first explicitly generating the model. The algorithm allows the partition to be generated
on-the-fly, thus directly from the description of the system, without first generating its state space. Obviously, this is essential for the application in symbolic model checking.

Figure 3.1 presents the minimal model generation algorithm. Given a labelled transition system \( \langle S, S_0, T \rangle \), and an initial partition \( \rho_0 \), it constructs the set of reachable classes of the partition induced by the greatest bisimulation on \( \langle S, S_0, T \rangle \) that respects \( \rho_0 \). The following variables are used in the algorithm:

- \( \rho \): The current partition
- \( \alpha \subseteq \rho \): The reachable classes of the current partition
- \( \sigma \subseteq \alpha \): The stable reachable classes of the current partition

Furthermore, the following functions are assumed to exist:

- \( \text{succ}^l(X, \rho) \): Gives the set of \( l \)-successor classes (the classes in \( \rho \) that have a state to which there is a \( l \)-labelled transition from a state in \( X \)):
  \[ \text{succ}^l(X, \rho) = \{ Y \in \rho \mid \exists s \in X, s' \in Y . s \xrightarrow{l} s' \} \]
- \( \text{pred}^l(X, \rho) \): Gives the set of \( l \)-predecessor classes:
  \[ \text{pred}^l(X, \rho) = \{ Y \in \rho \mid \exists s' \in Y . s' \xleftarrow{l} s \} \]
- \( \text{split}(X, \rho) \): Minimally splits \( X \) into subclasses that are stable with respect to \( \rho \). Thus, \( \text{split}(X, \rho) \) is the coarsest partition of \( X \) such that \( \forall Y \in \rho . \forall l \in \text{Lab} . X \cap \text{pred}^l(Y, \rho) \in \{ \emptyset, X \} \).

The general model minimization algorithm can be seen as a combination of two types of inferences:

- backward inference of stability information, and
- forward inference of reachability information

The interaction between these inferences comes from the requirement that

- reachable classes have to be stabilized, and
- stable classes can be used to infer reachability information.

The algorithm picks an arbitrary reachable and potentially instable class, and — if the class appears to be stable — infers reachability of successor classes, or — if the class was not stable — enforces stability by splitting the class into stable subclasses.
Starting from the initial partition, the algorithm iteratively refines instable reachable classes until all reachable classes are stable. Instable reachable classes are selected (line 3) and split into stable subclasses (line 4). The split function can be seen as stabilizing a part of the state space, moving towards a completely stable state space. If the split operation left the class intact, then the class was already stable. Since we know already that the class was reachable, it can be added to $\sigma$ (line 6). Also, we then know that successors of this class are also reachable. These are therefore added to $\alpha$ (line 7). In case the split operation actually resulted in a split, we have a set of subclasses of which we do not yet know which ones are reachable. Classes that are split have to be removed from $\alpha$ (line 9). For these classes we do not yet know if these are reachable, except in case such classes contain initial states. Therefore, those classes are added to $\alpha$ in line 10. Classes which had transitions to the class that was split, may become instable because of the splitting. Such classes are therefore removed from the set of stable classes $\sigma$ (line 11). Finally, line 12 updates the partition to reflect the new split.

The algorithm starts from an initial partition $\rho_0$ that respects all relevant atomic properties. If the greatest bisimulation induces a finite partition, the algorithm will terminate. A more elaborate discussion of the algorithm, as well as a correctness proof can be found in [24].

The minimal model generation algorithm is a general algorithm, allowing many different implementations. The most important implementation abstractions in the general algorithm are listed below.

- The fact that the state space is generated from an implicit description of the system, for example in the form of an XTG system is not addressed. To fully exploit the benefits of the algorithm, this generation would have to be done on-the-fly.

- Classes are handled as sets of states and "abstract transitions" between classes are handled in terms of the transitions of the underlying timed transition system. Sets of states and transitions will usually be very large and often infinite, and can therefore normally not be represented explicitly, but have to be represented symbolically. As was explained, the availability of such a representation is essential for an implementation of any symbolic algorithm.

- The nature of the stabilization operation split is left unspecified. Obviously, stabilization is not performed in terms of explicit states and transitions, but in terms of symbolic states and implicitly formulated transitions (i.e. XTG edges). For timed systems, stabilization involves also dealing with the progress of time between symbolic states.

The algorithm presented here is purely aimed at construction of a minimal model. We want to use the algorithm to verify properties of the state space to which it is applied. This means that some property evaluation approach has to be integrated, which is one of the issues that is addressed in the coming chapters.

The minimal model generation algorithm is not optimal because stabilization is done in a rather crude manner. A class is always stabilized with respect to the complete state space. As we will see, stabilization can alternatively be done in a more fine-grained manner, by stabilizing with respect to one class at the time. This avoids refinement of classes that are not reachable.
3.3 Abstract models for timed systems

To apply partition refinement in a real-time context, the notion of bisimulation must be extended. This section therefore introduces time-abstracting bisimulation, as well as abstract models that are generated by it.

3.3.1 Time-abstracting bisimulation

If we would apply the above discussed algorithm directly to timed transition systems, nothing would be gained. A finite model would not be obtained since most states would be not be considered equivalent. They would be distinguished by the different delay values on their outgoing time transitions. However, the exact values of these delays are often not relevant for our verification problem. From a TCTL verification perspective, runs are evaluated in terms of the states they visit. The delay values on time transitions are most of the time not relevant.

Therefore, a new transition relation is defined which abstracts away from the exact delay values. It turns out that bisimulation relations based on these so-called time-abstracting transition relations result in abstract models that are in many cases finite.

The following definition formally defines time-abstracting bisimulation, a notion introduced in [105].

**Definition 3.3 (time-abstracting bisimulation [105]).** Given a set of discrete transition labels $E$, a timed transition system $\langle S, S_0, T \rangle$ and a set of corresponding atomic properties $AP$, a time-abstracting transition relation $T_{ta} \in S \times (E \cup \{\tau\}) \times S$ is defined on $S$ in the following way:

$$
\begin{align*}
    s \xrightarrow{e_{T_{ta}}} s' & \iff s \xrightarrow{e} s' \\
    s \xrightarrow{\tau_{T_{ta}}} s' & \iff \exists \delta \in \mathbb{R} \geq 0 . s \xrightarrow{\delta} s'
\end{align*}
$$

We say that a relation is a time-abstracting bisimulation on a timed transition system $\langle S, S_0, T \rangle$ and $AP$ if it is a bisimulation on the labelled transition system $\langle S, S_0, T_{ta} \rangle$ and $AP$. □

This equivalence relation is either called time-abstracting bisimulation [111], strong time-abstracting bisimulation [106], or tai-bisimulation [105]. For more elaborate discussions on time-abstracting bisimulations see [106, 76]. Note that $\tau$ is used here only as a symbol to represent abstracted time transitions. It does not refer to often found usage of this symbol.

The fundamental idea behind time-abstracting bisimulation is that although quantitative time influences the behaviour of a system, it does not need to be relevant to the observer of the system. In fact, it assumes that the observer may be sensitive for the ordering of actions (i.e. qualitative time) but that the observer is not able to measure time. Time-abstracting bisimulation can directly be used for reachability analysis (see section 3.6). However, time-abstracting bisimulation is not compatible with TCTL verification since TCTL properties are able to measure time (by means of property clocks). Chapter 4 deals with this.

3.3.2 Abstract models and abstract paths

The following definition formalizes the notion of abstract models generated by time-abstracting bisimulation. Note that the resulting abstract models are labelled transition
Theorem 3.1 (abstract paths and concrete runs). Given a set of states $S$, the partition induced by an equivalence relation $B$ on $S$ is a partition $S'$ into equivalence classes of $B$ (i.e. $\forall s_1, s_2 \in S \cdot (\exists a \in S'. \{s_1, s_2\} \subseteq a \iff \langle s_1, s_2 \rangle \in B)$).

Given a timed transition system $M = \langle S, S_0, T \rangle$, and corresponding discrete transition label set $E$, an abstract model generated by the time-abstracting bisimulation $B$ is a labelled transition system $M' = \langle S', S'_0, T' \rangle$ such that

- $S'$ is the partition of $S$ induced by $B$.
- $S'_0 = \{a \in S' | a \cap S_0 \neq \emptyset\}$.
- $T' \subseteq S' \times (E \cup \{\tau\}) \times S'$, such that for any $a_1, a_2 \in S'$, $e \in E$,

\[
\begin{align*}
(a_1 & \xrightarrow{e/T} a_2 \iff \exists s_1 \in a_1, s_2 \in a_2 . s_1 \xrightarrow{e} s_2) \quad \text{and} \\
(a_1 & \xrightarrow{\tau/T} a_2 \iff \exists s_1 \in a_1, s_2 \in a_2 . \exists \delta \geq 0 . (s_1 \xrightarrow{\delta} s_2 \quad \text{and} \quad \forall s_3 \in S . \forall d . (0 < d < \delta \quad \text{and} \quad (s_1 \xrightarrow{d} s_3) \Rightarrow s_3 \in a_1 \cup a_2))
\end{align*}
\]

We say that $a$ is the abstract state of $s$, denoted $a = \text{abs}(s)$, if $s \in a$.

The definition of the abstract model’s transition relation deserves some attention. It would have been more straightforward to define the case for abstraction of time transition as

\[
a_1 \xrightarrow{\tau/T} a_2 \iff \exists s_1 \in a_1, s_2 \in a_2 . \exists \delta \geq 0 . s_1 \xrightarrow{\delta/T} s_2
\]

in which case there would have been a time transition in the abstract model whenever there are corresponding transitions in the original model. However, in our definition, $\tau$ transitions are left out if they can be obtained from a transitive closure over all $\tau$ transitions. Suppose there are three abstract states $a_1, a_2$ and $a_3$, and that there exists $\tau$ transitions $a_1 \xrightarrow{\tau/T} a_2$ and $a_2 \xrightarrow{\tau/T} a_3$. In that case the edge $a_1 \xrightarrow{\tau/T} a_3$ is left out because it is ”covered” by the other two transitions.

There are two reasons for this particular definition of abstract transitions. The first is a matter of optimization. This simplification consists of leaving out $\tau$ transitions that are redundant from an analysis point of view. The second reason is more fundamental. Defining abstract models in this fashion is essential when using the abstract model for TCTL verification. If one would allow abstract transitions to skip over abstract states (like the abstract transition $a_1 \xrightarrow{\tau/T} a_3$ of the above example), this would introduce abstract paths (see definition below) that do not represent proper abstractions of the concrete runs in the original model. For example, an $EGp$ property that is not satisfied in the concrete model could then be satisfied in the abstract model because an abstract path might exist that leaps over a set of states that do not satisfy $p$.

Runs of a timed transition system have corresponding paths in the abstraction of the timed transition system, and conversely. The following theorem gives some properties of these relations, which will be needed later.

Theorem 3.1 (abstract paths and concrete runs). Let $M$ be a timed transition system and $M'$ be the labelled transition system of a corresponding abstract model, as in definition 3.4. Then

1. Given a path $\psi : a_0 \longrightarrow a_1 \longrightarrow \cdots$ in $M'$ and a state $s_0$ from $M$ for which $s_0 \in a_0$, there exists a run $\pi : s_0 \longrightarrow s_1 \longrightarrow \cdots$, such that

   \[a) \ \forall i \geq 0 . \text{abs}(s_i) = a_i.\]
Proof. 

1. The concrete run is constructed straightforwardly, since \( s_0 \in a_0 \) and by the definition of abstract models, for each \( s_i \in a_i \) there must be an \( s_{i+1} \in a_{i+1} \) such that \( s_i \rightarrow s_{i+1} \) (remember that \( s \in a \iff \text{abs}(s) = a \)). By induction, this proves (a). For (b) we have to show that for each position \( (i, t) \) on \( \pi \), \( \text{abs}(s_i[+t]) \) is on \( \psi \). If \( s_i \xrightarrow{e} s_{i+1} \), then \( (i, 0) \) is the only valid position for \( i \), and thus \( \text{abs}(s_i[+0]) = \text{abs}(s_i) = a_i \). For \( s_i \xrightarrow{\delta} s_{i+1} \) we know from the definition of abstract models that \( \forall d . \ (0 < d < \delta \text{ and } s_i \xrightarrow{d} s_i[+d]) \Rightarrow s_i[+d] \in a_i \cup a_{i+1} \), which means \( \text{abs}(s_i[+d]) \in \{a_1, a_2\} \), for any position \( (i, d) \). Together, this means that (b) holds. For (c), \( (i_1, t_1) \ll (i_2, t_2) \) means that either \( i_1 < i_2 \), or \( i_1 = i_2 \) and \( t_1 \leq t_2 \). If \( i_1 < i_2 \) then \( \text{abs}(s_{i_1}[+t_1]) \in \{a_{i_1}, a_{i_1+1}\} \) (since \( \text{abs}(s_{i_1}) = a_1 \) and \( \text{abs}(s_{i_1+1}) = a_{i_1+1} \)) and \( \text{abs}(s_{i_2}[+t_2]) \in \{a_{i_2}, a_{i_2+1}\} \), in which case (c) holds. Alternatively, if \( i_1 = i_2 \) and \( t_1 \leq t_2 \) then from the properties of timed transition systems we know that \( s_i[+t_1] \xrightarrow{t_2-t_1} s_i[+t_2] \). From the definition of abstract models then follows that \( \text{abs}(s_i[+t_1]) \xrightarrow{t_2-t_1} \text{abs}(s_i[+t_2]) \), which means that (c) holds.

2. We first show that for each transition \( s_i \xrightarrow{lab_i} s_{i+1} \), we can construct the path fragment \( \text{abs}(s_i) = a_{i_0}^{lab_i} a_{i_1}^{lab_i} \cdots a_{i_n} = \text{abs}(s_{i+1}) \) of states from the abstract model. If \( lab_i \in E \) then from the definition of abstract models follows that \( \text{abs}(s_i) \xrightarrow{lab} \text{abs}(s_{i+1}) \). Alternatively, for \( lab_i = \delta \), let \( A_i = \{ \text{abs}(s''') \mid \exists d . \ (0 \leq d \leq \delta \text{ and } s_i \xrightarrow{d} s''') \} \) be the set of abstract states that the transition \( s_i \xrightarrow{\delta} s_{i+1} \) “passes through”. Then there is a sequence \( 0 = d_0 < d_1 < \cdots < d_n = \delta \) such that for each \( a \in A_i \), there is exactly one \( d_j \) such that \( s_i \xrightarrow{d_j} s' \) with \( s' \in a \). From the properties of timed transition systems then follows that \( s_i[+d_0] \xrightarrow{d_1} \cdots d_j \xrightarrow{d_j} s_i[+d_j+1] \), for each \( j \in \{0, \ldots, n\} \). From the definition of abstract models then follows that \( \text{abs}(s_i[+d_0]) \xrightarrow{d_1} \cdots \text{abs}(s_i[+d_j+1]) \). This means that we can build a path fragment \( \text{abs}(s_i) \xrightarrow{d_1} \text{abs}(s_i[+d_1]) \xrightarrow{d_2} \cdots \text{abs}(s_i[+d_n]) = \text{abs}(s_{i+1}) \). Thus we can build the path fragment from \( \text{abs}(s_i) \) to \( \text{abs}(s_{i+1}) \) for each transition \( s_i \xrightarrow{lab_i} s_{i+1} \). The abstract path can now be formed by concatenating these fragments for each \( s_i \).

Now, property (a) is readily proved. If \( s_i \xrightarrow{e} s_{i+1} \) then it follows immediately that \( \text{abs}(s_i[+0]) = \text{abs}(s_i) \) is on the abstract path. If \( s_i \xrightarrow{\delta} s_{i+1} \) then it follows immediately
from the definition of $A_i$ that for all positions $(i, t)$ with $t \leq \delta$ that $\text{abs}(s_i[+t]) \in A_i$ and thus that it is on the path. Also, property (b) and (c) follow immediately from the way the abstract path is constructed.\hfill \Box

Figure 3.2: Illustrating abstract paths and concrete runs

Figure 3.2 illustrates the notion of abstract paths and concrete runs, especially with respect to time transitions. It shows the value state space of some location for some XTG that has two clocks $c$ and $d$ shown on the horizontal and vertical axis respectively. It has four abstract states $a_1, a_2, a_4, a_3$ induced by the four constraints $c < 1$, $1 \leq c < 2$, $2 \leq c < 3$, and $c \geq 3$, respectively. Small circles represent concrete states, solid arrows denote concrete transitions and dotted arrows denote transitions of the abstract model. $s_1, s_2, s_3, s_4$ are four states that form a fragment $\cdots s_1 \stackrel{a_1}{\longrightarrow} s_2 \stackrel{\delta_2}{\longrightarrow} s_3 \stackrel{\delta_3}{\longrightarrow} s_4 \cdots$ of a run $\pi$. The path fragment $\cdots a_1 \stackrel{\tau}{\longrightarrow} a_2 \stackrel{\tau}{\longrightarrow} a_3 \stackrel{\tau}{\longrightarrow} a_4 \cdots$ is a part of the abstract path $\psi$ corresponding to $\pi$. Finally, $\cdots s'_1 \stackrel{\delta'_1}{\longrightarrow} s'_2 \stackrel{\delta'_2}{\longrightarrow} s'_3 \stackrel{0}{\longrightarrow} s'_4 \stackrel{\delta'_3}{\longrightarrow} s'_4 \cdots$ is a fragment of a concrete run $\pi'$ corresponding to $\psi$, starting from $s'_1$.

### 3.4 Region graphs and stability

This section defines region graphs as a vehicle for reasoning about symbolic state spaces. Note that the implementation of these symbolic state spaces is discussed in chapter 5. The abstract models defined above form a theoretical concept that allows us to reason about TCTL verification — as will be done in chapter 4. Region graphs are directly related to an XTG specification and do not have to be induced by a bisimulation equivalence. We will show that once a region graph satisfies certain criteria, it does define a proper abstract model as defined above.

#### 3.4.1 Region graphs

In the context of XTG, abstract states are viewed as combinations of locations and sets of valuations, referred to as regions.
**Definition 3.5 (regions).** A region is the combination of a single location and a zone. Thus, given an XTG with locations $L$ and variables $V$, a zone is an element of $\mathcal{P}(\text{Env}_V)$, and a region is an element of $L \times \mathcal{P}(\text{Env}_V)$.

To indicate zones, we will often use the letter $Z$, optionally with primes and subscripts. In the same way, $r$ is used to indicate regions. If $r = \langle l, Z \rangle$, we write $\langle l', \rho \rangle \in r$ if $l = l'$ and $\rho \in Z$, indicating that a state is included in a region. An analogous writing is $\langle l, Z \rangle \subseteq \langle l', Z' \rangle$ to indicate that $l = l'$ and $Z \subseteq Z'$. Other set operators are overloaded in the same manner.

Furthermore, we extend the satisfaction relation to regions in the following way:

$$r = \langle l, Z \rangle \models \phi \iff \forall \langle l, \rho \rangle \in Z. \langle l, \rho \rangle \models \phi$$

We will also be needing the notion of respect.

**Definition 3.6 (respect).** A region $r$ respects a TCTL property $\phi$ if either all states in $r$ satisfy $\phi$, or all states in $r$ dissatisfy $\phi$.

The definition below simplifies notation for reasoning about time edges in region graphs. $s \rightarrow[r] - \rightarrow s'$ expresses that there is a time transition from $s$ to $s'$ that passes only through states of $r$.

**Definition 3.7 (time edges in region graphs).** Let $s, s' \in S$ be two states of a timed transition system $\langle S, S_0, T \rangle$. Then given a region $r$, $s \rightarrow[r] - \rightarrow s'$ is defined as follows:

$$s \rightarrow[r] - \rightarrow s' \iff \exists \delta \geq 0. s \rightarrow[s]^T s' \land \forall \delta'. 0 \leq \delta' \leq \delta \Rightarrow \forall s'' \in S. (s \rightarrow[s]^T s'' \Rightarrow s'' \in r)$$

Usually, the $T$ is omitted.

A region graph defines a partition of the states of a timed transition system underlying an XTG. Vertices of the region graph correspond to classes of the partition. The single constraint that is put on such partitions is that all states in a particular class have to correspond to the same location. As a consequence, the vertices of a region graph are regions.

**Definition 3.8 (region graph).** Let $M = \langle S, S_0, T \rangle$ be a timed transition system corresponding to an XTG $\langle V, V_C, V_F, \rho_0, L, l_0, I, E, U \rangle$. Then a region graph corresponding to $M$ is a graph $\mathcal{RG} = \langle R, R_0, E_R \rangle$, with

- $R \subseteq L \times \mathcal{P}(\text{Env}_V)$, such that $\bigcup R = L \times \text{Env}_V$ and $\forall r_1, r_2 \in R \cdot r_1 \cap r_2 = \emptyset$.
- $R_0 = \{ r \in R \mid \exists s \in r \cdot s \in S_0 \}$ defines the set of initial regions.
- $E_R \subseteq R \times (E \cup \{ \tau \}) \times R$ is given by the following definition:

$$r \xrightarrow{e} E_R r' \iff \exists s \in r \cdot s \xrightarrow{e} s' \cdot s \xrightarrow{\tau} E_R r' \iff \exists s \in r \cdot s \xrightarrow{\tau} s' \cdot s \xrightarrow{[r \cup r']} - \rightarrow s'$$

We use $\text{req}(s)$ to denote the region to which $s$ belongs (which must exist because $R$ is a partition of $S$).

From the above definition follows that a region graph is completely defined by a partition of the state space $S$. The set of edges is induced by that partition. Like the timed transition system it is based on, a region graph has two types of edges: discrete edges (labelled with an XTG edge $e$) and time edges (labelled with $\tau$). There is a discrete edge between two
regions, if there are corresponding discrete transitions between states of the regions. There is a time edge between two regions if there are corresponding time transitions that do not pass through different intermediate regions.

Note that the term region graph is not consistently used in literature. It is also used to denote a more specific form of region graph that is induced by a specific equivalence relation called region equivalence [2]. We use the term region graph in a more general manner, imposing no equivalence relation (like it is used in for example [4]).

We will be needing the term urgent region, as defined below.

**Definition 3.9 (urgent region).** A region \( r \) of a region graph is denoted as an urgent region if there is a state in \( r \) in which an urgent edge of the XTG is enabled. \( \Box \)

### 3.4.2 Stability for region graphs

A region graph is based on an arbitrary partition of the state space. We are interested in those region graphs that define models that can be used for verification. We saw already that the stability concept can be used to characterize state sets as bisimulation equivalence classes. For region graphs the concept of stability is defined as follows.

**Definition 3.10 (stability for region graphs).** Let \( RG = \langle R, R_0, E_R \rangle \) be a region graph corresponding to a timed transition system \( M = \langle S, S_0, T \rangle \). Then

- \( r \xrightarrow{e} \in \mathcal{E}_R r' \) is stable if and only if \( \forall s \in r \cdot \exists s' \in r' \cdot s \xrightarrow{e} T s' \).
- \( r \xrightarrow{-} \in \mathcal{E}_R r' \) is stable if and only if \( \forall s \in r \cdot \exists s' \in r' \cdot s \xrightarrow{[r \cup r']} T s' \)

A region is stable with respect to an outgoing edge if that edge is stable. A region is stable if it is stable with respect to all its outgoing edges. Finally, a region graph is stable if all its reachable regions are stable. \( \square \)

Using this definition, the theorem below defines the conditions under which a region graph defines an abstract model with respect to time-abstracting bisimulation.

**Theorem 3.2 (stability and time-abstracting bisimulation).** Let \( M = \langle S, S_0, T \rangle \) be the timed transition system corresponding to an XTG \( X = \langle \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, E, \cdot \rangle \). Let \( RG = \langle R, R_0, E_R \rangle \) be a region graph corresponding to \( M \), and let \( AP \) denote the set of atomic properties for \( M \). If \( RG \) is stable and respects all atomic properties in \( AP \), then the partition of \( S \) defined by \( R \) corresponds to a time-abstracting bisimulation. \( \square \)

**Proof.** We have to prove that the partition associated with a stable region graph, defines a time-abstracting bisimulation equivalence, thus that any two states in any stable region are time-abstracting bisimulation equivalent. From definitions 3.3 and 3.1 it then follows that we have to prove that for an arbitrary region \( r \) and for any two arbitrary states \( s_1 \) and \( s_2 \) in \( r \) that:

- \( \forall p \in AP \cdot s_1 \in p \iff s_2 \in p \)
- \( \forall s'_1 \in S \cdot \forall e \in E \cdot s_1 \xrightarrow{e} T s'_1 \Rightarrow \exists s'_2 \in S \cdot s_2 \xrightarrow{e} T s'_2 \) and \( \text{reg}(s'_1) = \text{reg}(s'_2) \)
- \( \forall s'_1 \in S \cdot \forall \delta \geq 0 \cdot s_1 \xrightarrow{\delta} T s'_1 \Rightarrow \exists s'_2 \in S \cdot \exists \delta' \geq 0 \cdot s_2 \xrightarrow{\delta'} T s'_2 \) and \( \text{reg}(s'_1) = \text{reg}(s'_2) \)

The first item follows from the fact that \( R \) respects all atomic properties. For the second item, assume that \( s_1 \xrightarrow{e} T s'_1 \). Then from definition 3.8 it follows that \( \text{reg}(s'_1) = \text{reg}(s'_2) \) and then from definition 3.10 it follows that \( \exists s'_2 \in S \cdot s_2 \xrightarrow{e} T s'_2 \) and \( \text{reg}(s'_1) = \text{reg}(s'_2) \).

For the third item, assume that \( s_1 \xrightarrow{\delta} T s'_1 \). Then from definition 3.8 and timed transition system properties (definition 2.11) it follows that there must be a sequence \( r_1 = \)
Chapter 3 Partition Refinement

reg\(s_1\) \(\rightarrow\) \(E_R\) \(r_2\) \(\cdots\) \(\rightarrow\) \(E_R\) \(r_n\) = \(\text{reg}(s'_1)\). Then from definition 3.10 it follows that there must be a sequence \(s_2 \overset{\delta_1}{\rightarrow} r s_{2,1} \cdots \overset{\delta_n}{\rightarrow} r s_{2,n}\) with \(s_{2,n} \in \text{reg}(s'_1)\). Then from timed transition system properties it follows that for some \(\delta\), \(s_2 \overset{\delta}{\rightarrow} r s_{2,n}\). This proves the third item since \(s_{2,n} \in \text{reg}(s'_1)\) means that \(\text{reg}(s'_1) = \text{reg}(s_{2,n})\).

**Theorem 3.3 (region graphs and abstract models).** Let \(RG = \langle R, R_0, E_R \rangle\) be a region graph, let \(M = \langle S, S_0, T \rangle\) be the timed transition system to which \(RG\) corresponds, and let \(AP\) be a set of atomic properties for \(M\). If \(RG\) is stable and respects all atomic properties in \(AP\), then \(RG\) defines an abstract model \(\langle R, R_0, E_R \rangle\).

**Proof.** From theorem 3.2 follows that \(R\) is induced by a time-abstracting bisimulation. From the definitions for abstract models and region graphs it follows immediately that \(R\) is a correct transition relation (since for both the transition relation is defined in the same way).

**3.4.3 Characterizing stability**

Stability was only defined in terms of uninterpreted timed transition systems (definition 3.10). To be practically useful, a more concrete definition of stability is needed, in terms of the zones that define the regions.

There are two aspects to stability. First, for a region to be stable, in all states in that region exactly the same XTG edges must be enabled. This means that if a region graph edge is to be stable, the source region has to respect the guard of corresponding XTG edge. Also, one has to take into account invariants attached to destination locations, since these also influence the enabling of edges. If some states satisfy the guard of an outgoing XTG edge, but applying the edge’s update to the state would lead to a state that does not satisfy the invariant, then the outgoing edge is not enabled.

In terms of the algorithm this aspect of stability — which we will refer to as enabling stability — is a relative static quality that can be enforced through some initial splitting. After that, if a region is split, then enabling stability is preserved.

The second aspect of stability requires that all states in a region have transitions to the same regions. It is dependent of the splitting of regions to which edges exist. This aspect of stability introduces the need for propagation of splits. Figure 3.3 illustrates this. On the left side of that figure a region is split into three subregions (indicated by the dashed lines). As a consequence of that, the other region becomes unstable because it has transitions to the region that was split. Therefore, it has to be split into three subregions that are again stable with all outgoing transitions. We will refer to this aspect of stability as propagation stability. In terms of refinement of the region graph, propagation instability causes the repeated propagated splitting, while the enforcement of enabling instability (together with splits needed for verifying a specific property) initiates such chains of splits.

To be able to characterize propagation stability, the following two functions are defined.

**Definition 3.11 (pre\(_u\) and pre\(_r\)).** Let \(Z_1\) and \(Z_2\) be two zones. Let \(u\) be an update. Then \(\text{pre}_u\) is defined as follows.

\[
\text{pre}_u(Z_1, Z_2) = \{ \rho \in Z_1 \mid \rho[u] \in Z_2 \}
\]

Furthermore, we define \(\text{pre}_r\) as follows.

\[
\text{pre}_r(Z_1, Z_2) = \{ \rho \in Z_1 \mid \exists \delta \geq 0. \rho[+\delta] \in Z_2 \land \forall \delta'. 0 < \delta' < \delta \Rightarrow \rho[+\delta'] \in Z_1 \cup Z_2 \}
\]
pre_u(Z_1, Z_2) returns the subset of valuations in Z_1 which lead to a valuation in Z_2 when the update u is applied to it. pre_τ(Z_1, Z_2) returns the subset of valuations in Z_1 in which a valuation of Z_2 can be reached through the elapsing of time, without entering any other zones besides Z_1 and Z_2.

The following theorem formalizes the conditions for stability on region graphs.

**Theorem 3.4 (stability of region graphs).** Let M = ⟨S, S_0, T⟩ be a timed transition system corresponding to an XTG ⟨V, V_C, V_P, ρ_0, L, l_0, I, E, U⟩ and let RG = ⟨R, R_0, E_R⟩ be a corresponding region graph for which

\[
∀ \langle l_1, Z_1 \rangle \xrightarrow{(l_1, u, l_2)} \langle l_2, Z_2 \rangle ∈ E_R \Rightarrow Z_1 ⊆ \text{pre}_u([g], [I(l_2)])
\]  

(3.1)

Then

1. For any \langle l_1, Z_1 \rangle \xrightarrow{e} \langle l_2, Z_2 \rangle ∈ E_R with e = \langle l_1, g, u, l_2 \rangle,

   Z_1 = \text{pre}_u(Z_1, Z_2) ⇔ \langle l_1, Z_1 \rangle \xrightarrow{e} \langle l_2, Z_2 \rangle is stable

2. For any \langle l, Z_1 \rangle \xrightarrow{τ} \langle l, Z_2 \rangle ∈ E_R,

   Z_1 = \text{pre}_τ(Z_1, Z_2) ⇔ \langle l, Z_1 \rangle \xrightarrow{τ} \langle l, Z_2 \rangle is stable

**Proof.**

1. Z_1 = \text{pre}_u(Z_1, Z_2) means that ∀ ρ_1 ∈ Z_1 . ∃ ρ_2 ∈ Z_2 . ρ_1[u] = ρ_2. Since from equation 3.1 it follows that Z_1 ⊆ [g], this is equivalent to

   ∀ ρ_1 ∈ Z_1 . ∃ ρ_2 ∈ Z_2 . (ρ_1 ∈ [g] ∧ ρ_1[u] = ρ_2)

   which is according to XTG semantics equivalent to

   ∀ \langle l_1, ρ_1 \rangle ∈ \langle l_1, Z_1 \rangle . ∃ \langle l_2, ρ_2 \rangle ∈ \langle l_2, Z_2 \rangle . \langle l_1, ρ_1 \rangle \xrightarrow{e} \langle l_2, ρ_2 \rangle

   in other words, the edge is stable.

2. Let S_{urg} = \{ s ∈ S | ∃ s′ ∈ S . ∃ e ∈ U . s \xrightarrow{e} s′ \} denote the set of states in which at least one urgent edge is enabled. Then from equation 3.1 it follows that

   ∀ \langle l, Z \rangle ∈ R . \langle l, Z \rangle ∩ S_{urg} ∈ \{ \langle l, Z \rangle, ∅ \}

   (3.2)

Thus for each urgent region, all states in that region have at least one outgoing transition corresponding to an urgent edge. Thus S_{urg} captures the regions in which time can not elapse because one or more urgent edges are enabled.
Consider a time edge \( \langle l, Z_1 \rangle \xrightarrow{t} \langle l, Z_2 \rangle \) of the region graph. From the definition of region graphs it follows that there exists a state in \( \langle l, \rho \rangle \in \langle l, Z_1 \rangle \) from which a time transition departs. From the semantics of XTG then it follows that \( \langle l, \rho \rangle \notin S_{urg} \), and thus, using equation 3.2, \( \langle l, Z_1 \rangle \cap S_{urg} = \emptyset \). Also, from the definition of XTG and region graphs it follows that \( Z_1 \cup Z_2 \subseteq [I(l)] \).

Now for \( Z_2 \), either \( \langle l, Z_2 \rangle \cap S_{urg} = \emptyset \) or \( \langle l, Z_2 \rangle \subseteq S_{urg} \) (equation 3.2). These cases are considered separately, starting with the first. First assume that \( \langle l, Z_2 \rangle \cap S_{urg} = \emptyset \).

\(^{\footnote{By giving necessary conditions for region graph edges to be stable, the theorem defines the requirements for a region graph to be stable. The condition in equation 3.1 ensures enabling stability. This is done by requiring that for each edge, the zone corresponding to the region from which the edge departs, respects the guard of the associated XTG edge, and that the zone of valuations that result after applying the update, respects the invariant of the destination location. The conditions \( Z_1 = pre_u(Z_1, Z_2) \) and \( Z_1 = pre_r(Z_1, Z_2) \) enforce propagation stability.}}\)
3.5 An example

This section gives a taste of what TCTL model checking based on partition refinement looks like. For a very small TCTL-XTG verification problem, a verification scenario is sketched, illustrating the kind of operations on a symbolic state space that are needed. The example system and property are depicted in figure 3.4. Remember that a blob at the beginning of an edge indicates that the XTG edge is urgent.

\[
\begin{align*}
&c := 0 \\
&d := 0 \\
&d := 0 \\
&[d > 2] \\
&d := c + 3
\end{align*}
\]

\[AF \, d > 6 ?\]

Figure 3.4: An example verification problem

Figure 3.5 shows the sequence of splits that would occur, when dealing with the verification problem in our partition-refinement based approach. It shows five snapshots. For each snapshot it shows for both locations how the value space is split into zones. In the figure, a zone implicitly represents a region since it either belongs to location \(l_0\) (on the left side) or \(l_1\) (on the right side).

Exploration starts at the initial location \(l_0\). First, the initial region corresponding to \(\langle l_0, [\text{TRUE}] \rangle\) is split to respect the invariant of \(l_0\), \(d < 4\). The region \(\langle l_0, [d \geq 4] \rangle\) is marked as being inaccessible, since the states it represents do not satisfy the invariant.

To be able to evaluate our TCTL property, reachable regions are split to ensure that atomic properties are respected. In our case, we have to split according to \(d > 6\). However this split is not needed since the region \(\langle l_0, [d < 4] \rangle\) already respects the constraint \(d \leq 6\). Furthermore, we have to ensure that the region respects the guards of the outgoing edges (enabling stability). This results in a split according to \(d > 2\). Also, a time edge is added between the two newly created regions, reflecting the fact that each state in \(\langle l_0, [d \leq 2] \rangle\) has a time edge to a state in \(\langle l_0, [2 < d < 4] \rangle\). Note that no edge to \(\langle l_0, [d > 4] \rangle\) because that region is not accessible. The result is shown in figure 3.5 (a).

Now that the regions belonging to \(l_0\) respect invariants, atomic properties and the guards of outgoing edges, the outgoing edge to \(l_1\) can be explored. The first step is to ensure that reachable regions associated with \(l_1\) also respect invariants, atomic properties and guards. In this case we only have to split the initial region \(\langle l_1, [\text{TRUE}] \rangle\) according to the atomic property \(d > 6\). Note that no time edge is added between the two subregions, because in the regions of \(l_1\) no time is allowed to pass. This is the consequence of the fact that in all of its states an urgent edge is enabled. The resulting regions are shown in snapshot (b) of figure 3.5. There, it is also shown that we can assign the region \(\langle l_1, [d > 6] \rangle\) the value \(T\) (for \textit{true}) indicating that it satisfies \(AF \, d > 6\) (which follows directly from the fact that it satisfies \(d > 6\)).

In snapshot (b), the edge from \(l_0\) to \(l_1\) is not yet drawn, because it is not stable. The region in which the edge is enabled \(\langle l_0, [2 < d < 4] \rangle\) has to be split in order make it stable with respect to the edge. This means that we have to separate the states that have transition that lead to states of \(\langle l_1, [d > 6] \rangle\) from those that do not. This is a straightforward computation that takes into account the update on the edge. As a result \(\langle l_0, [2 < d < 4] \rangle\) is split by \(c > 3\), resulting in two subregions — \(\langle l_0, [c > 3 \land 2 < d < 4] \rangle\).
Figure 3.5: Example splitting scenario

and \( \langle l_1, [c \leq 3 \land 2 < d < 4] \rangle \) — that are stable with respect to edges to \( l_1 \).

As a result of the new split, a new time edge appears between the two new regions, causing the source region of this edge \( \langle l_1, [c \leq 3 \land 2 < d < 4] \rangle \) to become instable. To remedy this, a split by \( d - c < 1 \) is necessary. Together this results in the situation depicted in snapshot (c).

We can now assign the region \( \langle l_0, [2 < d < 4 \land c > 3 \land d - c > 1] \rangle \) the value \( T \),
because it is stable, all its outgoing edges are explored, and all destinations of these edges satisfy $AF \, d > 6$. The exact rules for such decisions are discussed in the coming chapters.

Again there is instability, caused by the time edge from $\langle l_0, d \leq 2 \rangle$. This region has to be split into subregions that are stable with respect to outgoing time edges. The results the situation shown in snapshot (d). Now all regions of $l_0$ are again stable. Note that only reachable edges are shown.

The last step for this example consists of exploring the edge from $\langle l_1, d \leq 6 \rangle$ to $l_0$. This does not result in additional splitting, since because of the updates on this edge all edges go to the region containing the initial state. Note that we do not explore the edge from $\langle l_1, d > 6 \rangle$ to $l_0$. This would be useless, since $\langle l_1, d > 6 \rangle$ is already decided. We can now decide that all reachable regions, including the initial region, satisfy $AF \, d > 6$. See diagram (e) of figure 3.5.

3.6 An algorithm for parametric reachability analysis

Chapter 6 will define a TCTL model checking algorithm based on the general minimal model generation algorithm discussed in section 3.2. Here, a first step is taken by presenting an algorithm for the simpler problem of reachability analysis. Also, it deals with several — but not all — of the aspects that were abstracted from in the general algorithm, and can be seen as an intermediate step towards the TCTL model checking algorithm presented later, in the sense that it is aimed at a simpler verification problem and that still several refinements are possible.

3.6.1 Parametric reachability analysis

The concept of parametric reachability analysis is necessarily defined in terms of XTG models. For parametric reachability analysis the goal is to characterize the solution set $S_{sol} \subseteq S_0$ in terms of parameter values. Thus what is needed is a characterization of the set $Q$ of initial parameter valuations that, together with the initial values for the other variables, defines the set of initial states from which a state satisfying $p$ can be reached.

Definition 3.12 (parametric reachability analysis). Let $G = \langle V, \cdot, V_p, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot \rangle$ be an XTG, and let $M = \langle S, S_0, T \rangle$ be the timed transition system associated with $G$. Furthermore, let $p \in \text{Expr}_V$ be a predicate over the variables (including parameters) of $G$. Then a parametric reachability problem is defined as a pair $\langle G, p \rangle$.

The solution set of a parametric reachability problem is the set

$$Q = \{ \nu \in Env_{V_p} \mid \exists \langle l, \rho \rangle \in S \cdot \rho \in [p]_V \land \langle l_0, \rho_0 + \nu \rangle \leadsto_T \langle l, \rho \rangle \}$$

A solution of a parametric reachability problem is an expression $sol \in \text{Expr}_{V_p}$ such that $[sol]_{V_p} = Q$. □

Here, the set of states for which reachability is to be tested is given by means of a predicate on the value state. Alternatively, it could have been defined by a set of locations. This would not significantly change the approach.

It is not hard to prove that parametric reachability is preserved by time-abstracting bisimulation. In other words, given a stable region graph $\langle R, R_0, E_R \rangle$ that respects $p$, the solution set $Q$ is equal to $\bigcup R_{sol}$, where

$$R_{sol} = \{ r \in R_0 \mid \exists r' \in R \cdot \exists \langle l, \rho \rangle \in r' \cdot (\rho \in [p]_V \land r \leadsto_{E_R} r') \}$$
We will not give a proof here since the more general case of TCTL verification is handled in chapter 6.

3.6.2 Refining the general algorithm

Before presenting the algorithm, some attention is paid to how it refines the general minimal model generation algorithm. First of all, we have a concrete input language. Rather than abstracting from the implicit description that generates state spaces, we now want to take into account that the abstract state space is generated on-the-fly from an XTG model. Thus, input of the algorithm is a parametric reachability problem \( \langle G, p \rangle \).

The algorithm will operate on a partial state space, which reflects the fact that most of the time only a part of the XTG has been explored. Additional data structures will be needed to administrate the relation between the partial region graph and the XTG it is based on. For example, for each explored region we have to maintain the set of outgoing unexplored XTG edges. But it also means that for example information concerning inaccessible parts of the state space and urgency has to be stored.

The bookkeeping of regions and edges has to be made explicit. Remember that in general algorithm, edges where induced from the underlying transition system. Now, the algorithm will has explicitly store a representation for regions and region graph edges. Note that at this point we do not provide a solution for efficient representation and manipulation of the state space. This is the topic of chapter 5, the algorithm here assumes that it exists.

In the general algorithm splitting is purely based on instability. In the refined algorithm, to allow exploration of the state space from an XTG specification, it will be necessary to perform certain splits that are not directly induced by instability. One could say that these splits implement the initial partition of the general algorithm. This concerns four aspects.

- First of all, we will perform splits to separate states that are inaccessible (i.e. excluded by invariants), thus ensuring that these states are not taken into account.
- Second, we will always split to ensure that all guards of all edges are respected. This is needed anyway to ensure enabling stability, but there is a reason to do this beforehand. We have to distinguish states which have enabled outgoing urgent edges from those that do not, because the former cannot have outgoing time transitions. To be able to stabilize a region, we have to know whether or not its states have outgoing urgent edges.
- Third, we have to ensure that regions respect the reachability predicate \( p \) to allow evaluation of the reachability property.
- Finally, we want to enforce that different locations cannot occur in one region, i.e. the partition has to respect the control structure expressed by the location component of the state. This proves to be convenient for many reasons, one of them being the symbolic state space representation discussed later.

In terms of the general algorithm one could say that we have an initial partition \( \rho_0 \) such that it respects invariants, respects guards, respects the reachability predicate, and respects the distinction between different locations of regions.

As was already suggested in section 3.2.2, stabilization can be performed in a more fine grained manner, based on stabilization of regions with respect to single instable edges, rather than with respect to a complete partition. This improvement is included in the algorithm presented here. As a consequence, it is avoided that irrelevant stabilization is performed.
The general algorithm arbitrarily picks reachable classes to be processed next. The refined algorithm is aimed at as much as possible maintaining the partial partition stable with respect to a set of already explored edges. Every time a new edge is explored, instability is introduced. Before exploring any other edge, the algorithm first attempts to remove the newly introduced instability. This results in two nested iterations, the outer one exploring new edges, and the inner one propagating the instability caused by the newly explored edge backwards through the partial region graph.

To use the algorithm for verification purposes, it has to be equipped with a mechanism that, based on the constructed abstract models, decides the verification problem. Such a mechanism should be able to operate on-the-fly, thus in our case, operate on partial region graphs. For parametric reachability, this mechanism is relatively simple. Each region \( r \) is given a value \( \text{value}(r) \) which indicates (1) whether or not reachability is already decided for that region and if so, (2) whether or not the region is known to satisfy reachability. This means that for a region \( r \), \( \text{value}(r) \) can have three values:

- \( U \): no decision was taken for this region,
- \( T \): a region satisfying \( p \) is reachable from \( r \), or
- \( F \): no region satisfying \( p \) is reachable from \( r \).

Two values would already have been sufficient (\( U \) and \( T \)), but using three values results in a clearer presentation and matches to the approach that will be used in the TCTL algorithm.

The algorithm uses a simplified interpretation of region graphs that is allowed for reachability analysis. Time edges that can be formed by the transitive arrangement of other time edges are normally not represented in region graphs. For reachability analysis, we do not include this constraint, which results in a simpler algorithm description. For reachability analysis this is not a problem, but for TCTL verification, which is discussed later, the correct region graph definition has to be used.

3.6.3 The algorithm

We will use \( \text{Reg} = L \times \text{Env}_V \) to denote the domain of all possible regions for the timed structure \( M = (S, S_0, T) \) induced by an XTG \( G = \langle V, V_C, V_P, \rho_0, L, l_0, I, E, U \rangle \). Also, we will use \( \langle l, Z_{in} \rangle = \langle l, \text{Env}_V \rangle \) to denote the region that covers all states corresponding to a location \( l \). \( R \) is the set of regions that defines the current partial partition. In the algorithm a region \( r \in R \) may become split, which means that in \( R \), \( r \) is replaced by its subregions. However, in our representation of partial region graphs, edges may still refer to \( r \), being unaware that it is split. To be able to deal with this in a simple way, we will use \( R^* = \{ r \in \text{Reg} \mid \exists R' \subseteq R \cdot \bigcup R' = r \} \) to denote the set of unions of regions in \( R \), such that the members of a union all correspond to a single location.

The following variables are used to keep track of the current partial partition:

- \( L_{exp} \subseteq L \): the subset of explored locations. Note that a location being explored does not mean that all of its outgoing edges have been explored.
- \( R \subseteq \text{Reg} \): the current set of regions of the region graph. It defines a partition of the part of the state space for which the locations have been explored. Thus, it is a refinement of \( \{ \langle l, Z_{in} \rangle \mid l \in L_{exp} \} \).
- \( R_0 \subseteq R \): the subset of initial regions.

- \( \text{sub} : R^* \rightarrow \mathcal{P}(R) \): holds the set of subregions from \( R \) into which a region is split. Thus \( \text{sub}(r) = \{ r' \in R \mid r' \subseteq r \} \). Note that \( \forall r \in R \cdot \text{sub}(r) = r \).
\( \mathcal{S} \subseteq R^* \): the set of regions that still need further exploration. Note that \( \mathcal{S} \) may contain regions that have been split.

Edges of the partial region graph are kept in three different structures:

- \( \text{stab} : R \rightarrow R^* \times (E \cup \tau) \): holds the incoming edges of a region \( r \) that are known to be stable. If \( \langle r', \tau \rangle \in \text{stab}(r) \) then there is a stable time edge from \( r' \) to \( r \). If \( \langle r', e \rangle \in \text{stab}(r) \) then there is a stable discrete edge from \( r' \) to \( r \) labelled with \( e \). Note that \( r' \) is allowed to be a split region.

- \( \text{instab} : R \rightarrow R^* \times (E \cup \tau) \): associates with each region in \( R \) the set of (potentially) unstable outgoing edges. If \( \langle r', \tau \rangle \in \text{instab}(r) \), then there is a stable time edge from \( r \) to \( r' \), where \( r' \) is a region that could be split. Likewise, \( \langle r', e \rangle \in \text{instab}(r) \) denotes a stable discrete edge to a potentially split region.

- \( \text{unexp} : R \rightarrow E \): associates with each region of the abstract model the edges from \( E \) that are enabled in \( r \), but which are not explored yet.

Only \( \text{stab} \) and \( \text{instab} \) hold edges of the current partial region graph. Stable edges are stored with their destination (in \( \text{stab} \)) as these edges will only be needed in case for destination regions new information becomes available (instability or decisions). This can then be propagated back to the source region.

Instable edges are stored with their origin because in order to stabilize the origin, its instable outgoing edges have to be known. Edges are added to \( \text{instab} \) when the source of the edge is not known to be reachable. If the destination of such an edge is split, this does not have to be propagated back to the source region. The reason for storing edges in \( \text{instab} \) is that currently unreachable source regions may become reachable at a later stage, in which case the edge becomes again relevant.

So an explored edge is stored either at its destination or at its source, depending on whether reachability of the edge is confirmed. As a consequence, an edge \( \langle r', e \rangle \in \text{instab}(r) \) represents a stable edge for which \( r' \) can be split into subregions, causing \( r \) to be instable. An edge \( \langle r, e \rangle \in \text{stab}(r') \) also represents a stable edge \( r \xrightarrow{e} r' \), but in this case, \( r' \) is not split and therefore \( r \) is stable with respect to the outgoing edge \( e \). However, it may be the case that \( r \) is split into subregions.

For unexplored edges, we only know the XTG edge, which is stored in \( \text{unexp} \). Thus \( \text{instab}(r) \) and \( \text{unexp}(r) \) together hold the set of instable edges, the first those that have been generated already from the XTG, the second the implicit ones that still have to generated from the XTG description. Figure 3.6 illustrates the different cases.

\[
\begin{align*}
\langle r, e \rangle & \in \text{stab}(r') & r \xrightarrow{e} r' \text{ is stable} \\
\langle r', e \rangle & \in \text{instab}(r) & r \xrightarrow{e} r' \text{ is stable} \\
e & \in \text{unexp}(r) & e \text{ \ is a XTG edge}
\end{align*}
\]

Figure 3.6: Illustrating the storage of edge

Finally, we have to keep track of some basic qualities of regions.

- \( \text{reachable} : R \rightarrow \mathbb{B} \): indicates whether or not at least some of the states in a region are known to be reachable from an initial region.

- \( \text{noaccess} : R \rightarrow \mathbb{B} \): indicates whether or not the states in a region are inaccessible as a result of an invariant.
An algorithm for parametric reachability analysis

Figure 3.7: main loop of the parametric reachability analysis algorithm

Figures 3.7, 3.8, 3.9, 3.11 and 3.12 present a pseudocode description of the algorithm. The state space is explored starting from the initial states by iteratively selecting either instable edges (from $I_{sy}$) or yet unexplored edges (from $unexp$). New splits that occur during this exploration are propagated backwards through the partition (over edges stored in $stab$), minimizing instability in the reachable part of the state space. Also reachability information is kept and propagated forwards through the state space.

The algorithm simultaneously performs four main activities:
1. Exploring new edges and, as a consequence, new locations;
2. Stabilizing the partial region graph;
3. Maintaining reachability information;
4. Deciding values of regions.

In the main loop (figure 3.7), these four aspects can be recognized. Iteratively, a region $r$ is taken from the current set $S$ of reachable regions that are under investigation (line 5). During the time it was on $S$, $r$ could have become split. In that case, $r$ is replaced on $S$ by its reachable subregions (lines 6–7). If $r$ does not have instable nor unexplored outgoing edges, then it is removed from $S$ because in that case there is nothing left to be done for this region (lines 8–9). Regions that have been decided already can also be removed from $S$. 

$urgent : R \rightarrow \mathbb{B}$: indicates whether or not the states in a region have at least one enabled outgoing urgent edge.

1. $L_{exp} := \emptyset$
2. $initialize \ (l_0)$
3. $S := \{(l_0, Z_{in})\}$
4. while $S \neq \emptyset$ and $\exists r \in R_0 \ s.t. \ value(r) = \mathcal{U}$ do
5. choose some $r \in S$
6. if $sub(r) \neq \{r\}$ then
7. $S := (S \setminus \{r\}) \cup \{r' \in sub(r) \mid reachable(r')\}$
8. else if $(instab(r) = \emptyset \ and \ unexp(r) = \emptyset) \ or \ value(r) \neq \mathcal{U}$ then
9. $S := S \setminus \{r\}$
10. else
11. if $instab(r) \neq \emptyset$ then
12. choose some $\langle nr, e \rangle \in instab(r)$; $instab(r) := instab(r) \setminus \{\langle nr, e \rangle\}$
13. else
14. choose some $e = \langle l, g, u, l' \rangle \in unexp(r)$; $unexp(r) := unexp(r) \setminus \{e\}$
15. if $l' \notin L_{exp}$ then
16. $initialize \ (l')$
17. $nr := \langle l', Z_{in} \rangle$
18. $S := S \cup sub(nr)$
19. $stabilize \ (r, e, nr)$
20. for each $nr' \in sub(nr) \ s.t. \ value(nr') = \mathcal{T}$ do
21. $propagate.value \ (nr')$
22. for each $r \in R_0 \ s.t. \ value(r) = \mathcal{U}$ do
23. $value(r) := \mathcal{F}$
If \( r \) has explored but instable edges then such an edge is selected (line 12), otherwise, an new, unexplored edge is selected (line 14). In the latter case, if the destination location of the edge was not yet explored already, a new region is added and initialized for that location (lines 15 and 16).

Now (line 18) we have a new edge, which is probably instable. Therefore, the \textit{stabilize} procedure (described later) is called to fix the instability and propagate stability information further back in the graph (line 19). Also, (the subregions of) the newly reached region are added to \( S \) (line 18), ensuring that it is considered in a future iteration.

The \textit{initialize} procedure, decides reachability for those subregions of \( nr \) that satisfy \( p \). If for one of these regions, reachability was decided, then this information is propagated back to its predecessor regions by the \textit{propagate_value} procedure (lines 20 and 21).

\begin{verbatim}
initialize (l):
24. let \( r = (l, Z_{in}) \)
25. \( R := R \cup \{r\}; L_{exp} := L_{exp} \cup \{l\} \)
26. \( \text{reachable}(r) := \text{false}; \text{value}(r) := U; \text{urgent}(r) := \text{false}; \text{noaccess}(r) := \text{false} \)
27. if \( l = l_0 \) then \( R_0 := R_0 \cup \{r\}; \text{reachable}(r) := \text{true} \)
28. \( \text{constraint_split}(r, I(l)) \)
29. for each \( r' \in \text{sub}(r) \) s.t. \( r' \neq I(l) \) do
30. \( \text{noaccess}(r') := \text{true} \)
31. \( \text{constraint_split}(r, p) \)
32. for each \( r' \in \text{sub}(r) \) s.t. not \( \text{noaccess}(r') \) and \( r' \models p \) do
33. \( \text{value}(r') := T \)
34. for each \( (ll, g, u, ll') \in E \) s.t. \( ll = l \) do
35. \( \text{constraint_split}(r, g) \)
36. for each \( r' \in \text{sub}(r) \) do
37. \( \text{if } r' \models g \text{ then} \)
38. \( \text{unexp}(r') := \text{unexp}(r') \cup \{(ll, g, u, ll')\} \)
39. \( \text{if } \text{urgent}((ll, g, u, ll')) \text{ then } \text{urgent}(r') := \text{true} \)
40. for each \( r' \in \text{sub}(r) \) s.t. not \( \text{urgent}(r') \) and not \( \text{noaccess}(r') \) do
41. \( \text{stabilize}(r', \tau, r) \)
\end{verbatim}

Figure 3.8: \textit{initialize}

Figure 3.8 shows the initialize procedure. When incorporating a new edge, the algorithm will often encounter locations that were not yet visited (line 16). The \textit{initialize} procedure initializes the regions for such locations. A new region \( (l, Z_{in}) \) is created which represents all possible states associated with the location \( l \). Using the \textit{constraint_split} procedure, these newly created regions are immediately split to create subregions that respect the invariant of \( l \) (line 28), the reachability predicate \( p \) (line 31) and the guards of all outgoing edges of this location (line 35). Splitting according to \( p \) is needed to be able to do reachability analysis. Splitting according to invariants is needed to avoid that states that are excluded by an invariant are taken into account. Regions representing such states are therefore marked as \textit{noaccess}. Splitting with respect to guards enforces the enabling aspect of stability, as discussed in section 3.4.3. Edges that are enabled in a region \( r \) are added to \textit{unexp}(\( r \)).

Note that in principle it is not necessary to directly split newly explored regions on the guards of all outgoing edges. This introduces unnecessary splitting, because it will often occur that regions are split which turn out to be unreachable. Note however, that it is
essential that it is known whether or not a region has one or more enabled urgent edges, before any stabilization is performed. If a region has at least one outgoing urgent edge, then \( \text{urgent}(r) \) is set.

In line 41 the appropriate time edges are added to those newly created regions that do not have outgoing urgent edges. This is done by stabilizing these regions with respect to a time edge from and to itself.

\[
\begin{align*}
stabilize (r, \text{lab}, nr): \\
42. & \text{if } \text{sub}(r) \neq \{r\} \text{ then} \\
43. & \quad \text{for each } r' \in \text{sub}(r) \text{ do} \\
44. & \quad \quad \text{stabilize } (r', \text{lab}, nr) \\
45. & \text{else} \\
46. & \quad \text{if } \text{value}(r) = \mathcal{U} \text{ then} \\
47. & \quad \quad \text{if } \text{reachable}(r) \text{ then} \\
48. & \quad \quad \quad \text{let } H = \text{stab}(r) \\
49. & \quad \quad \quad \text{propagate}\_\text{split}(r, \text{lab}, nr) \\
50. & \quad \quad \quad \text{if } \text{sub}(r) \neq \{r\} \text{ then} \\
51. & \quad \quad \quad \quad \text{for all } \langle pr, \text{lab}' \rangle \in H \text{ do} \\
52. & \quad \quad \quad \quad \quad \text{stabilize } (pr, \text{lab}', r) \\
53. & \quad \quad \quad \text{for each } nr' \in \text{sub}(nr) \text{ do} \\
54. & \quad \quad \quad \quad \text{for each } \langle r', \text{lab}'' \rangle \in \text{stab}(nr') \text{ do} \\
55. & \quad \quad \quad \quad \quad \text{if } \text{reachable}(r') \text{ do} \\
56. & \quad \quad \quad \quad \quad \quad \text{reachable}(nr') := \text{true} \\
57. & \quad \quad \quad \quad \quad \quad \text{else} \\
58. & \quad \quad \quad \quad \quad \quad \quad \text{stab}(nr') := \text{stab}(nr') \setminus \{\langle r', \text{lab}'' \rangle\} \\
59. & \quad \quad \quad \quad \quad \quad \quad \quad \text{instab}(r') := \text{instab}(r') \cup \{\langle nr', \text{lab}'' \rangle\} \\
60. & \quad \quad \quad \quad \quad \quad \text{else} \\
61. & \quad \quad \quad \quad \quad \quad \quad \text{instab}(r) := \text{instab}(r) \cup \{\langle nr, \text{lab} \rangle\}
\end{align*}
\]

Figure 3.9: stabilize

The stabilize procedure (figure 3.9) is responsible for stabilizing a region according to an outgoing instable edge. By doing so, it may introduce new instability, which is handled by recursive calls to the procedure — thus implementing the backward propagation of splits. The procedure is also responsible for maintaining the reachable attribute.

The central operation is the propagate\_split in line 49, which stabilizes \( r \) with respect to a lab-edge to \( nr \) by splitting \( r \) into subregions that are stable with respect to lab-edges to subregions of \( nr \). It adds all edges between subregions of \( r \) and \( nr \) to the appropriate stab attribute of subregions of \( nr \). This splitting operation may cause stable predecessors of \( r \) to become instable, because states corresponding to these predecessor regions may now have successors in different regions. Therefore, if \( r \) is split, stabilization is subsequently performed for all reachable predecessors of \( r \) (lines 51 and 52). In this manner splits are propagated backwards through the graph. The stabilize operation also keeps track of reachability properties (lines 53 – 59). Unreachable edges are removed from instab and added to the appropriate stab attribute. Lines 42 – 44 ensure that stabilization is always done on regions of the current partition, i.e. regions that have not been split already.

The stabilization operation is an integrated procedure for inference of stability information and reachability information. When a region is split because of stabilization, it immediately deals with possible newly introduced instability, rather than marking these
regions for investigation later. This propagation will continue until regions are reached for which no splitting is required. From these regions, reachability information can be propagated forwards, all the way to where the splitting sequence was started. Figure 3.10 illustrates this. In that figure, the ellipse identifies the set of initial states and filled rectangles denote regions that are known to be reachable. The bottom region is split (diagram (b)) and its subregions are no longer marked as reachable, because it is not yet known which of these regions are (at least one of them is). Diagram (c) shows the result of propagating the split back to the middle region. This region is split and new edges are introduced. Propagating further, also the top region is split (diagram (d)). The left-hand subregion contains initial states and is therefore marked as reachable (diagram (e)). As a result, as shown in diagram (f), also the left-hand subregion of the middle region can be marked as reachable, since we know that the edge between the two subregions is stable. Finally, the same reasoning applies for the bottom region (diagram (g)). A dashed arrow indicates an edge from an unreachable region, which is stored in $\text{instab}$, rather than in $\text{stab}$.

\[
\text{propagate\_value}\ (r): \quad \begin{align*}
62. & \quad \text{for each } (pr, \text{lab}) \in \text{stab}(r) \text{ do} \\
63. & \quad \text{for each } pr' \in \text{sub}(pr) \text{ do} \\
64. & \quad \text{if } \text{value}(pr') = \mathcal{U} \text{ then} \\
65. & \quad \quad \text{value}(pr') := \mathcal{T} \\
66. & \quad \quad \text{propagate\_value}\ (pr')
\end{align*}
\]

Figure 3.11: propagate\_value

In the model checking process, values of regions are decided on-the-fly. Initially all values are set to $\mathcal{U}$ (undecided). If a region is encountered that satisfies $p$, its value can immediately be decided to $\mathcal{T}$ (true). This value is then reported to the stable predecessors which as a result are also set to true. This mechanism thus causes values to be propagated backwards through the graph. It is implemented in the \textit{propagate\_values} operation. Once the reachable part of the state space has been explored completely, all remaining regions can be set to false. Thus, in the parametric reachability algorithm, the value of a region

\[
\text{for each } (pr, \text{lab}) \in \text{stab}(r) \text{ do} \\
\text{for each } pr' \in \text{sub}(pr) \text{ do} \\
\text{if } \text{value}(pr') = \mathcal{U} \text{ then} \\
\quad \text{value}(pr') := \mathcal{T} \\
\quad \text{propagate\_value}\ (pr')
\]

Figure 3.11: propagate\_value
An algorithm for parametric reachability analysis

is decided in one of three ways:

1. based on the characteristics of the region: if the region satisfies \( p \), its value can be set to \( \mathcal{T} \) (line 33).
2. based on the value of its stable immediate successors: if the value of a successor region is set to \( \mathcal{T} \), the value of the region can also be set to \( \mathcal{T} \) (line 65).
3. based on global characteristics of the graph: if the exploration is finished all undecided regions are set to \( \mathcal{F} \) (line 23).

Once all initial regions have become decided to \( \mathcal{T} \), the algorithm terminates (line 4).

Alternatively, if \( \mathcal{S} \) becomes empty, then a complete stable region graph has been built, in which case the algorithm also terminates. Then all remaining undecided initial regions can be decided to \( \mathcal{F} \). Note that decision making for reachability is rather simple. The reason for choosing this somewhat involved decision procedure is that it serves as a simple form of the decision procedure for TCTL, discussed later in this thesis.

Until now, we have left two operations unspecified, namely, \texttt{constraint\_split} and \texttt{propagate\_split}. Their implementation very much depends on how the symbolic state space representation is realized. Figure 3.12 gives their (implicit) definitions. \texttt{constraint\_split} simply splits a region to subregions that respect a constraint. This is needed to ensure enabling stability and that \( p \) as well as invariants are respected. It uses the \texttt{refine} procedure, shown in the same figure, to perform the actual splitting, according to \( P \).

\begin{verbatim}
constraint_split (r, c):
  67. let \( P \) be the coarsest partition of \( r \) that respects \( c \)
  68. refine \((r, P)\)

propagate_split \((r, lab, nr)\):
  69. let \( P \) be the coarsest partition of \( r \) such that
      \( \forall \langle l, Z \rangle \in P \ . \forall \langle l', Z' \rangle \in \text{sub}(nr) . \pre_{lab}(Z, Z') \in \{Z, \emptyset\} \)
  70. refine \((r, P)\)
  71. for each \( \langle l, Z \rangle \in \text{sub}(r), \langle l', Z' \rangle \in \text{sub}(nr) \) s.t. not \texttt{noaccess}(\langle l', Z'\rangle) do
      72. if \( \pre_{lab}(Z, Z') = Z \) then
      73. \( \text{stab}(\langle l', Z'\rangle) := \text{stab}(\langle l', Z'\rangle) \cup \{\langle l, Z \rangle, \text{lab}\} \)
  74. \( R := R \cup P \)
  75. for each \( r' \in P \) do
      76. \text{value}(r') := \mathcal{U}; \text{reachable}(r') := \text{false}; \text{noaccess}(r') := \text{false}
      77. \( \text{stab}(r') := \text{stab}(r); \text{unexp}(r') := \text{unexp}(r); \text{instab}(r') := \text{instab}(r) \)
  78. if \( r \in R_0 \) then
      79. \( R_0 := R_0 \setminus \{r\} \)
  80. for each \( r' \in \text{sub}(r) \) s.t. \( \exists s \in r' . s \in S_0 \) do
      81. \( R_0 := R_0 \cup \{r'\} \); \text{reachable}(r') := \text{true}
  82. \( R := R \setminus \{r\} \)

Figure 3.12: The two splitting procedures
\end{verbatim}

The \texttt{propagate\_split} procedure enforces propagation stability as was discussed in section 3.4.3. The required split is (implicitly) defined in line 69 as the minimal splitting \( r \) that results in subregions that are stable with respect to edges to \( nr \). Lines 71 – 73 are responsible for adding appropriate edges. Note that at this point all edges are added to
3.6.4 Correctness

Below an informal outline of the correctness proof is given.

1. The correct set of edges is represented:

\[
\forall r, r' \in R. \forall s \in r, s' \in r', \text{lab} \in (E \cup \{\tau\}) \ . \ (s \xrightarrow{\text{lab}} s') \Rightarrow \\
\exists \ r \ . \langle rr, e \rangle \in \text{stab}(r') \ \lor \\
\exists \ r' \ . \langle rr', e \rangle \in \text{instab}(r) \ \lor \\
\text{lab} \in \text{unexp}(r)
\]

Thus for each region in \( r \), an outgoing edge is either not yet explored, or it is stored in either \( \text{stab} \) or \( \text{instab} \). Initially, all enabled discrete edges of a region \( r \) are added to \( \text{unexp}(r) \) (line 38). Line 41 takes care of the introduction of time edges. The \text{stabilize} procedure and the splitting procedures ensure that edges are added correctly to either \( \text{stab} \) or \( \text{instab} \) such that above equation holds.

2. Stability of the edges in \( \text{stab} \) and \( \text{instab} \). For each \( r \in R \), for each \( \langle r', \text{lab} \rangle \in \text{stab}(r) \), \( r \) is stable with respect to \( r' \xrightarrow{\text{lab}} \). For each \( \langle r', \text{lab} \rangle \in \text{instab}(r) \), \( r' \) is stable with respect to \( r' \xrightarrow{\text{lab}} r \). Enabling stability is ensured by the \text{initialize} procedure (lines 28 and 35) while ensuring propagation stability is the sole purpose of the \text{stabilize} procedure.

3. For each reachable and undecided region \( r \) that has unexplored or instable outgoing edges, \( r \) or a region of which \( r \) is a subregion, must be on \( S \). Thus \( \forall r \in R \ . \((\text{instab}(r) \neq \emptyset \lor \text{unexp}(r) \neq \emptyset) \land \text{reachable}(r) \land \text{value}(r) = U \Rightarrow \exists r' \in S \ . \ r \in\text{sub}(r')\)\). Initially, all newly created regions are put on \( S \) (line 18). Furthermore, only unreachable or undecided regions and regions having no instable or unexplored edges are removed from \( S \) (lines 6 – 9).

4. Correct completion of exploration. \( S = \emptyset \Rightarrow \forall r \in R \ . \((\text{reachable}(r) \land \text{value}(r) = U) \Rightarrow (\text{instab}(r) = \emptyset \land \text{unexp}(r) = \emptyset)) \). This follows directly from property 3.

5. Correctness of reachable attribute: \( \forall r \in R \ . \text{reachable}(r) \Rightarrow \exists s \in r \ . \exists s' \in S_0 \ . s' \leadsto s \). Either a state is reachable because it is initial (line 81) or it has an incoming edge from a reachable region (line 56).

6. \( \forall r \in R \ , r \text{ respects } p \). This is ensured in line 31.

7. Correctness of \( T \) decisions: \( \forall r \in R \ . \text{value}(r) = T \Rightarrow \forall s \in r \ . \exists s' \in [p] \ . s \leadsto s' \). \( T \) decisions in line 33 are obviously correct (using property 6). Furthermore, the correctness of \( T \) decisions in the \text{propagate_value} function can be shown using stability of edges (property 2).

8. Correctness of \( F \) decisions: \( S = \emptyset \Rightarrow \forall r \in R \ . \((\text{reachable}(r) \land \text{value}(r) = U) \Rightarrow \forall s \in r \ . \exists s' \in [p] \ . s \leadsto s' \). This can be shown by an induction argument using properties 2, 1 and 4. As a consequence, \( F \) decisions in line 23 are correct.
9. Let $P_f$ be the partition of $S$ that respects $p$, all invariants of all locations and the guards of all edges. If there exists a time-abstracting bisimulation w.r.t. $P_f$ that induces a finite partition then let $P_{min}$ be coarsest partition induced by such a time-abstracting bisimulation. Then at all times $P_{min}$ is a refinement of $R$. This property follows from the fact that $\text{constrained\_split}$ and $\text{propagate\_split}$ will never perform unnecessary splits.

10. If a partition $P_{min}$ exists, then the algorithm terminates. This follows from the previous property, and the fact that if the algorithm runs forever, it must keep on performing splits.

### 3.7 Discussion

Our approach is based on the minimal model generation algorithm of [24]. In [77] an alternative partition refinement algorithm is presented, but in essence these algorithms are very similar. In [77], splitting is done with respect to outgoing edges, rather than with respect to the whole partition. As we discussed already, the refined algorithm of section 3.6 also applies this more optimal splitting approach.

Another difference between the algorithm of [77] and ours is that the former gives priority to forward inference of reachability information. The minimal model generation focuses on backward inference of stability. Only from stable classes forward reachability is inferred. The algorithm of [77] maintains for each reachable class, a representative state, which is used to infer reachability from unstable classes. It is not clear if this algorithm performs better. It seems likely that this would depend on the structure of the verification problem. In [49] a symbolic model checking implementation based on [77] is presented.

In [42, 43] partition refinement techniques for CTL* (a generalization of CTL) properties are described. These were not implemented. An early partition refinement algorithm can be found in [89]. Unlike algorithms mentioned above, this algorithm does not take into account reachability.

It is not easy to compare partition refinement approaches with other symbolic model checking approaches, because there are many factors that might influence such a comparison. For example, it would depend on how the general partition refinement algorithms are concretely instantiated, how the initial partition is chosen, and what the verification goal is. Also it would depend on how the symbolic state space is represented. The result of a comparison will probably depend on the state space representation that is chosen. Finally, in practice the performance of a specific approach will also depend on the problem structure. The relative efficiency of a model checking procedure depends on the verification problem that is to be dealt with.

In [50], a comparison is made of different partition refinement algorithms in the context of reachability checking (in their terms, checking of invariant properties). There, it is shown that partition refinement algorithms do not perform better than backward reachability algorithms. This result crucially depends on the choice for the initial partition. There, the initial partition is taken to be $\{[p], [\neg p]\}$. Indeed, for that case partition refinement approaches become degraded to backward reachability analysis. By doing so, one eliminates the local character of partition refinement techniques, because knowledge on reachability is not used properly. This also explains their conclusion that in their context the algorithm of [89] — which ignores reachability — perform no worse than the new partition refinement techniques. An initial partition that also separates initial states would give different results.
In [4], a first application of partition refinement in model checking real-time systems is reported. There the minimal model generation algorithm is taken and straightforwardly instantiated to deal with real-time systems. Also, the algorithm of [77] has also been extended to a real-time systems context [109]. Here two important improvements are made, which would also apply to the approach of [4]. First, it stabilizes with respect to single edges rather than the whole partition. Second, it avoids the difference operation on zones, which is in general very costly. As we will see in chapter 5, our approach also ensures convexity, using a different approach.

In [3] different approaches for partition refinement in real-time model checking are compared. The main conclusion is that the two partition refinement approaches perform better than a symbolic forward reachability approach, although the basis for comparison is quite small.

Our refinement of the minimal model generation algorithm is partly based on ideas from the real-time partition refinement algorithm of [98, 97]. In particular, this concerns the idea of propagating of splits and values. By focusing on stabilizing regions as soon as possible, verification becomes potentially more efficient, since decisions can only be taken on the basis of stable regions. This idea is implemented by our approach to backward propagation of splits, which originates from [98].

All approaches mentioned above focus on the original timed automaton formalism in which clocks are the only data type, and the only allowed update on clocks is the reset operator. Our focus is more broad, dealing with a less restricted modelling language. For example, this means that our approach also deals with the consequences of urgent edges. Also the approaches mentioned above differ in the type of verification problems that are handled. In chapter 6 we will discuss the combination of partition refinement with types of verification problems.

For a certain subset of XTG, finite bisimulations can be proven to exist. However, the fact that a finite model exists in no way guarantees that the verification problem can be solved, since the size of the resulting models may still inhibit their evaluation. On the other hand, the class of XTG systems for which the existence of a finite model cannot be guaranteed, has many members for which verification can be done in terms of a finite abstract model.

In terms of the three key problems identified in section 1.1.6, this chapter described a state space exploration technique. The coming chapter will define a state space evaluation approach for TCTL verification. This will be done in a generic manner based on the abstract models identified in section 3.3. After that, chapter 5 will present a solution for the final key problem: the symbolic state space representation. This representation will be optimized for our type of exploration algorithm. It starts from the fact that the exploration algorithm is based on partition refinement. Therefore, that chapter takes into account the results of sections 3.4 and 3.6.
Chapter 4

Evaluating TCTL properties

The previous chapter defined an algorithm for solving the parametric reachability problem based on abstract models generated by time-abstracting bisimulations. This chapter investigates how to decide TCTL properties on such abstract models, however without describing concrete algorithms.

In terms of the three key issues in defining a symbolic model checker (see section 1.1.6), this chapter defines a generic solution for the evaluation of state spaces for verification of TCTL properties. It is therefore based on abstract models (section 3.3) rather than region graphs. Chapter 6 will instantiate this generic solution for the state space exploration approach based on partition refinement, discussed in chapter 3.

4.1 Introduction

For reachability analysis it was very convenient that reachability properties were retained by time-abstracting bisimulation — reachability properties "carry over" from timed transition systems to abstract models generated by time-abstracting bisimulations. A first idea would therefore be to build — like for reachability analysis — a region graph, but use different, more sophisticated, decision procedures to cope with TCTL properties. It turns out that such an approach is not possible when straightforwardly using region graphs.

In the context of non-real-time model checking, bisimulation abstraction retains the satisfaction of CTL properties [29]. This means that a CTL property of some model can be checked by evaluating that property on the abstracted model. However, time-abstracting bisimulation does not directly retain the satisfaction of TCTL properties. Among other things, divergence does not carry over to abstract models, since transitions in abstract models do not have delays associated with them. TCTL is intended to deal with dense models, while the models resulting from abstraction are discrete, and hopefully finite. A natural solution to this problem seems to be to use CTL, the non-real-time analogue of TCTL, for properties on abstract models, and translate TCTL properties to corresponding CTL properties.

We will show that, by using CTL for properties of abstract models, time-abstracting bisimulation can be used for TCTL verification. However, we cannot directly formulate CTL properties on abstract models that characterize exactly those states that satisfy a TCTL property on the original model. Some aspects of the TCTL language cannot directly be dealt with by CTL. To deal with this, we have to extend our models with some
additional structure.

Given a timed transition system generated by an XTG system $X$ and a corresponding TCTL property $\phi$, we can construct an extended XTG system $X'$ and a CTL property $\psi$ such that the verification of $\phi$ on $X$ can be reduced to the verification of $\psi$ on an abstract model of $X'$. Note that the extension from $X$ to $X'$ must be safe in the sense $X$ and $X'$ should be indistinguishable by our TCTL property. Figure 4.1 illustrates the approach. The left side shows the straightforward approach as it can be used for CTL verification (and the verification of reachability). The right side illustrates the approach we adopt for TCTL.

![Figure 4.1: Abstraction approaches](image)

This strategy introduces two subproblems. First, we have to formulate the problem of deciding TCTL properties on timed transition systems in terms of deciding CTL properties on finite abstract models of these timed transition systems. To do so, we have to deal with TCTL concepts that do not carry over to CTL. As will be shown, this requires the extension of the XTG under investigation with some additional structure which does not affect the relevant behaviour of the system. For these extended systems the reduction of TCTL verification problems to CTL verification problems on abstract models can be defined and proved correct. This can be found in section 4.2.

The second subproblem is the evaluation of CTL properties on abstract models. While many algorithms already exist for dealing with CTL verification on transition systems, we will take a general view on this problem. Given a partially explored model, we are looking for decision rules for CTL verification that are optimal, i.e. that decide the verification problem as soon as possible. Thus, we are looking for generic decision criteria that are independent of any exploration approach. This is particularly important for a partition-refinement based symbolic model checking approach. First of all, compared to explicit model checking, in symbolic model checking redundant explorations of the state space have far more impact on the model checker's performance. Secondly, for partition refinement model checking, different choices for an evaluation approach are still possible — unlike for example reachability-based model checking. Section 4.3 presents a general decision approach for CTL.

### 4.2 Abstract models for TCTL verification

Suppose we have an XTG $X$ and a TCTL property $\phi$. Then the question addressed in this section is how to safely extend $X$ to $X'$ such that we can formulate a CTL property $\psi$ such that verifying $\psi$ on an abstract model of $X'$ can be used to decide the satisfaction of $\phi$ on $X$. More formally, let $s$ be a state in the timed transition system $M$ of $X$, let
s' be the "corresponding" state in the timed transition system M' of X', and let F be a fairness property for X (and X'). Also, let B denote the labelled transition system of the abstract model of M' and F' a fairness property for the abstract model of X' that corresponds to F. Then
\[ s \models_{M}^F \phi \iff s' \models_{M'}^F \phi \quad \text{and} \quad s' \models_{M'}^F \phi \iff \text{abs}(s') \models_{B}^F \psi \]
Thus, firstly, the extension of X leads to a new XTG system X' that satisfies the same properties, and secondly, \( \psi \) is a property for the abstract model of X' that characterizes the states of X' that satisfy \( \phi \).

It is useful to briefly discuss the differences between CTL and TCTL in the light of abstract models. The most obvious conceptual difference between CTL and TCTL is that the first is interpreted over discrete models, while the second is interpreted over dense models. This is reflected in the fact that TCTL does not have next-time operators and a somewhat different semantics for \( \phi_1 U \phi_2 \). It turns out that on abstract models the \( U \) operator in CTL retains the meaning of the \( U \) operator in TCTL in the original model.

There are two other TCTL concepts that do not immediately carry over to CTL. First, TCTL is interpreted over divergent runs, while for CTL, divergence is not an issue because it is not interpreted over a dense state space. However, somehow the divergence constraint on concrete models also has to be reflected in the abstract models, because otherwise our abstraction would also take into account non-divergent behaviour. This issue is discussed in section 4.2.2.

Secondly, to be able to express timing constraints, TCTL has property variables. Although one could add property variables to CTL, this would not help, because such property variables would have no meaning on abstract models. So a more involved solution is needed. Section 4.2.3 deals with this.

### 4.2.1 Normalizing properties

To avoid useless complexity, some trivial restrictions are put on TCTL formulas. First, we assume that boolean operators only occur within the scope of temporal operators. This means that properties that have boolean operators at the outermost level, like for example \( AG p \land AF q \), are not taken into account. This does not limit the usefulness of TCTL, since the properties that are excluded can easily be split into smaller properties that can be separately verified (in our example \( AG p \) and \( AF q \)). The only reason for this restriction is that it simplifies the descriptions. Secondly, we make the trivial assumption that each property contains at least one temporal operator. Finally, we assume that for updates that do not occur in the scope of a temporal operator, the right-hand side of the assignments can only be literal values. Thus, for properties that start with an update, such updates may only assign literal values. This only excludes obscure properties that can also be modelled in an alternative, admissible way. Thus a property \( x:=z. AF \phi \) where \( z \) is a system variable, is not allowed. By replacing \( z \) by the initial value of \( x \), an equivalent but admissible property is obtained.

In section 4.2.3 we will see that each update in the property introduces additional complexity in the state space. It is therefore essential to minimize the number of updates in the property, which can be done by collecting updates together as much as possible. This results in formulas in which updates only occur right "before" temporal operators. Together with the above mentioned restrictions, this results in the following restricted syntax for TCTL formulas.
Definition 4.1 (normalized TCTL properties). A normalized TCTL property is a TCTL property for which formulas are generated by the following syntax:

\[
\phi ::= u'. \phi^a | \phi^a \\
\phi^a ::= \phi^b \ AU \ \phi^b | \phi^b \ EU \ \phi^b \\
\phi^b ::= \phi^b \lor \phi^b | \neg \phi^b | u. \phi^a | \phi^a | p | le
\]

where \( u \) and \( le \) are as defined in definition 2.28, and where \( u' \) is like \( u \) except that the right-hand sides of the assignments in \( u \) can only be literal values.

To see that the above syntax, except for the restrictions above, does not limit the expressiveness of TCTL, first of all note that two subsequent updates can be combined into one. Let \( e[u] \) denote the expression \( e' \) obtained from \( e \) by replacing for each \( (v,e) \in u \), each occurrence of \( v \) by \( e \). Then \( u_1, u_2, \phi = u. \phi \) where

\[
u = u_2 \cup \{(v, e[u_2]) | (v, e) \in u_1\}
\]

For any sensible property this comes down to \( u_1, u_2, \phi = (u_1 \cup u_2). \phi \). As a consequence, consecutive updates can always be avoided.

Furthermore, property updates distribute over boolean operators, that is

\[
u. (\phi_1 \lor \phi_2) = u. \phi_1 \lor u. \phi_2 \text{ and } u. \neg \phi = \neg u. \phi
\]

which can be easily seen from the semantics of TCTL. Thus, updates can be pushed through the boolean operators. Also updates before atomic properties can be eliminated:

\[
u. le = le \text{ and } u. p = p[u]
\]

Using these properties, a TCTL property can be rewritten such that all updates end up right. "before" a temporal operator, resulting in formulas obeying the syntax of definition 4.1. For example, for the following (nonsense) property \( x:=0.y:=x.(x < 1 \ AU \ z:=1.(y = 0 \lor AF \ z = 0))) \) the normalized version becomes \( \{x:=0, y:=0\}.(x < 1 \ AU \ (y = 0 \lor z:=1. \ AF \ z = 0))) \)

To be able to deal with initial updates, the following operations on properties are needed.

Definition 4.2 (identifying subproperties). \( \phi' \) is said to be a strict subproperty of \( \phi \) if it occurs in the scope of \( \phi \). subprop(\( \phi \)) denotes the set of subproperties of \( \phi \), which is the union of \( \{\phi\} \) and the set of all strict subproperties of \( \phi \). subprop_temp(\( \phi \)) \( \subseteq \) subprop(\( \phi \)) denotes the set of subproperties of which the outer operator is a temporal operator. subprop_upd(\( \phi \)) \( \subseteq \) subprop(\( \phi \)) denotes the set of subproperties of which the outer operator is an update operator. subprop_atom(\( \phi \)) \( \subseteq \) subprop(\( \phi \)) denotes the set of atomic subproperties of \( \phi \).

Definition 4.3 (stripping properties). Let init(\( \phi \)) denote the set of initial updates of a normalized property \( \phi \) and strip(\( \phi \)) denote the stripped form of \( \phi \), defined as follows:

\[
\begin{align*}
\text{init}(u, \phi') &= u \quad &\text{strip}(u, \phi') &= \phi' \\
\text{init}(\phi_1 \ AU \ \phi_2) &= \emptyset \quad &\text{strip}(\phi_1 \ AU \ \phi_2) &= \phi_1 \ AU \ \phi_2 \\
\text{init}(\phi_1 \ EU \ \phi_2) &= \emptyset \quad &\text{strip}(\phi_1 \ EU \ \phi_2) &= \phi_1 \ EU \ \phi_2
\end{align*}
\]

Thus, init(\( \phi \)) and strip(\( \phi \)) separate initial updates from the rest of the property. As a result init(\( \phi \)). strip(\( \phi \)) is equivalent to \( \phi \).
4.2.2 Divergence

Recall that a run was defined to be divergent (definition 2.14) if for an arbitrary delay \( d \) greater than 0, for all states \( s \) along the run, there is a future state \( s' \) such that on the sub-run from \( s \) to \( s' \) at least \( d \) time-units elapse. The question now is how to check for divergence, or more precisely, how to limit the interpretation of temporal operators to only divergent runs. The basic idea for doing so is that the divergence constraint of TCTL can be expressed as a fairness property. It is easy to see that a run is divergent if for any \( d > 0 \), infinitely often \( d \) time-units can pass. However, to specify this as a fairness condition, we have to recognize the passing of \( d \) time-units.

\[ c := 0 \]
\[ l_0 \]
\[ c \leq 2 \]
\[ c := 1 \]
\[ l_1 \]
\[ c \leq 2 \]
\[ l_2 \]
\[ AF \ g@l_2 \ ? \]

Figure 4.2: Illustrating divergence

It turns out that abstract models as defined in section 3.3 do not inherently allow the recognition of divergence. This is illustrated by the verification problem shown in figure 4.2, which is almost equal to the one depicted in the top of figure 2.4. The difference is that here an update on the edge from the second to the first location is added. This causes the property to become false, while for the version without the update the property is satisfied. This is explained by the fact that now a divergent run exists that cycles through the first two locations. However, there are abstract models that could be associated with both verification problems. In fact, the smallest abstract model for the first problem is exactly the same as the one for the second problem. This is illustrated in figure 4.3, which shows this minimal abstract model. The fact that the update causes the run to become divergent, thus changing the value of the property, is not reflected in the abstract model.

\[ \langle l_0, [c \leq 2] \rangle \]
\[ \langle l_1, [c \leq 2] \rangle \]
\[ \langle l_2, [\text{true}] \rangle \]

Figure 4.3: Abstract model for the XTG in figure 4.2

This means that abstract models generated by time-abstracting bisimulations cannot be used directly to detect divergence. For standard timed automata, which is a less expressive formalism than ours, this is not a problem ([4], [62]). In timed automata, the only manipulations on clocks are clocks resets, while constraints can only be bounds on clock values or bounds on clock differences. In that case a run is divergent if for every clock \( c \), it is either reset infinitely many times, or eventually always \( c > v_c \), where is \( v_c \) is the largest constant \( c \) is ever compared to ([2]). This can be expressed as a fairness condition. In our case, clock assignments are arbitrary expressions in which clocks as well as non-clock variables can be used. A similar condition for our model would not be possible.

Instead, we have to extend the system with some additional structure that allows the recognition of divergence as a fairness property. A small graph \( G_{\text{div}} \) is added to the XTG
system, which ensures that the progress of time can be recognized by a fairness condition. In other words, the additional graph allows the formulation of a fairness condition $F_{\text{div}}$ that is only satisfied for runs on which infinitely often $d$ time-units can pass, for an arbitrary $d > 0$. If $X$ is the XTG system that is to be verified, then a new XTG system resulting from the extension of $X$ with $G_{\text{div}}$ (see definition 2.22) will be used as the new starting point for the verification. The new fairness property becomes $F \cup F_{\text{div}}$ where $F$ is the original fairness condition. The extension of $X$ with $G_{\text{div}}$ does not change the relevant behaviour of $X$, it just adds some transitions that cannot change the value of any TCTL property over $X$.

$$\text{Figure 4.4: The graph } G_{\text{div}} \text{ for recognizing divergence}$$

A divergence graph and a matching fairness condition can be formulated in different ways. Here one solution is presented, referred to as $G_{\text{div}}$ and $F_{\text{div}}$, but others can be thought of. Figure 4.4 presents $G_{\text{div}}$, in which $dc$ is a clock and $v_D$ is an arbitrary positive constant. The important characteristic of $G_{\text{div}}$ is that it is forced to perform a transition every $v_D$ time-units. $F_{\text{div}}$ is defined as $\{dc = 0, dc = v_D\}$, although other definitions also work. As a consequence of this fairness condition a run in the extended system is $F_{\text{div}}$-fair if and only if the edge of $G_{\text{div}}$ is taken infinitely often. The theorems below formalize the use of $G_{\text{div}}$ and $F_{\text{div}}$ for recognizing divergence.

**Theorem 4.1 (divergence as a fairness property).** Let $X$ be an XTG and $X'$ be the extension of $X$ with $G_{\text{div}}$, where $G_{\text{div}}$ is defined as in figure 4.4. Let $M'$ denote the timed transition system corresponding to $X'$. Then a run in $M'$ is divergent if and only if it is fair with respect to $F_{\text{div}} = \{dc = 0, dc = v_D\}$. $\square$

**Proof.** Suppose a run $s_0 \rightarrow s_1 \rightarrow \cdots$ in $M'$ is divergent. Then according to the definition of divergence this means that for each $d > 0$, there is an $s_i$ along this run such that from $s_0$ to $s_i$ at least $d$ time-units have elapsed. The divergence graph necessarily traverses its only edge every time exactly $v_D$ time-units have passed. This means that for any delay $d$, the edge must be traversed $d/v_D$ times. As a consequence, a run is divergent if and only if the edge of $G_{\text{div}}$ is traversed infinitely often.

If the edge of $G_{\text{div}}$ is traversed infinitely often then both $dc = 0$ as well $dc = v_D$ are infinitely often true, since $dc = 0$ is a necessary postcondition of the edge, while $dc = v_D$ is a necessary precondition of the edge. Also, in the other direction, if both $dc = 0$ as well $dc = v_D$ are infinitely often true, then the edge of $G_{\text{div}}$ must be traversed infinitely often because the only way from $dc = v_D$ to $dc = 0$ is by traversing the edge. Therefore, a run traverses the edge of $G_{\text{div}}$ infinitely often if and only if it is $F_{\text{div}}$-fair. Combining the two results we get that a run is divergent if and only if it is $F_{\text{div}}$-fair. $\square$

We still have to prove that the additional graph does not interfere with our verification problem. The following theorem states that adding $G_{\text{div}}$ to an XTG system results in a new system that is equivalent with respect to our property.
Theorem 4.2 (safety of adding \(G_{\text{div}}\)). Let \(X\) be an XTG and \(X'\) be the extension of \(X\) with \(G_{\text{div}}\), where \(G_{\text{div}}\) is defined as in figure 4.4. Let \(F\) be a fairness property for \(X\) and let \(F' = F \cup F_{\text{div}}\). Let \(M\) and \(M'\) denote the timed transition system corresponding to \(X\) and \(X'\), respectively. Then for each TCTL property \(\phi\) for \(X\),

\[
\langle l, \rho \rangle \models^F_M \phi \iff \langle l + l^*, \rho + \nu \rangle \models^{F'}_{M'} \phi
\]

(4.1)

where \(l^*\) denotes the location of \(G_{\text{div}}\) and \(\nu\) is an arbitrary valuation for \(\{dv\}\) such that \(0 \leq \nu(dv) \leq v_D\).

Proof. The proof is done by induction on the structure of \(\phi\). This means that we have to prove that

\[
\langle l, \rho \rangle \models^{F}_M \phi \iff \langle l + l^*, \rho + \nu \rangle \models^{F'}_{M'} \phi
\]

(4.2)

holds, assuming that it holds for all subproperties of \(\phi\) (\(\xi\) being an arbitrary valuation for the property variables).

- The induction case is formed by the atomic properties. The atomic properties only refer to the components of states that originate from \(X\) and thus for an atomic property \(p\), \(\langle l, \rho \rangle \models^{F}_M \phi \iff \langle l + l^*, \rho + \nu \rangle \models^{F'}_{M'} \phi\).

- \(\phi_1 \cup \phi_2\):
  It has to be proved that for any state \(\langle l, \rho \rangle\) in \(M\) and \(\langle l + l^*, \rho + \nu \rangle\) in \(M'\),

\[
\exists \pi \in \Pi^F_M (\langle l, \rho \rangle) . \pi \models^{F}_{M, \xi} \phi_1 \cup \phi_2 \iff \exists \pi' \in \Pi^{F'}_{M'} (\langle l + l^*, \rho + \nu \rangle) . \pi' \models^{F'}_{M', \xi} \phi_1 \cup \phi_2
\]

(4.3)

given that equation 4.2 already holds for \(\phi_1\) and \(\phi_2\).

First, note that from the definition of \(G_{\text{div}}\) we know that

- \(G_{\text{div}}\) does not influence the enabling of edges that originate from \(X\), since it only accesses local data and it does not engage in synchronizations.
- \(G_{\text{div}}\) does not change the value of any variables accessed by edges from \(X\), since it only accesses its local variable.
- \(G_{\text{div}}\) cannot block the progress of time, since in \(G_{\text{div}}\) time can always pass, except when \(dc = v_D\) in which case the edge of \(G_{\text{div}}\) can always be executed, leading to a situation in which time can again elapse.

We start with the \(\implies\) direction of equation 4.3. Given the run \(\pi\), a run \(\pi\) can be constructed from \(\pi\) in which some of the \(\delta\) transitions in \(\pi\) are split into smaller \(\delta\) transitions such that for each integer \(k > 0\), if there exists a position \(\langle i, t \rangle\) on \(\pi\) such that \(\Delta_\pi(\langle i, t \rangle) = k \cdot v_D\), then there is a zero time transition \(s \xrightarrow{0} s\), where \(s = \Xi_\pi(\langle i, t \rangle)\).

As a consequence, if \(k \cdot v_D\) can elapse on \(\pi\) then there is transition \(s \xrightarrow{0} s\) on \(\pi\), where at \(s\) exactly \(k \cdot v_D\) time units have passed. Since \(\pi\) is derived from \(\pi\) by only splitting time transitions into smaller ones, if follows from the semantics of TCTL that \(\pi\) and \(\pi\) are equivalent with respect to \(\phi_1 \cup \phi_2\).

Now the run \(\pi'\) can be constructed from \(\pi\), by taking edges originating from \(X\) exactly at the same moment as in \(\pi\), but replacing each zero-time transition that was introduced above, by a discrete transition originating from the edge of \(G_{\text{div}}\). Thus

- a discrete transition \(\langle l, \rho \rangle \xrightarrow{(l,g,u,l')} \langle l', \rho' \rangle\) in \(\pi\) leads to a discrete transition \(\langle l + l^*, \rho + \nu \rangle \xrightarrow{(l + l^*, g,u,l' + l')} \langle l' + l^*, \rho' + \nu \rangle\) in \(\pi'\).
A transition \((l, \rho) \xrightarrow{0} (l, \rho)\) in \(\pi\) that was introduced above leads to a discrete transition \((l + l^*, \rho + \nu) \xrightarrow{(l + l^*, \sigma_{\nu} \cdot \pi, l + l^*)} (l + l^*, \rho + \nu[u^*])\) in \(\pi'\) (where \(g^*\) and \(u^*\) denote the guard and update of the \(G_{\text{div}}\) edge, respectively);

- all other time transitions \((l, \rho) \xrightarrow{\delta}(l, \rho[+\delta])\) in \(\overline{\pi}\), lead to a transition \((l + l^*, \rho + \nu) \xrightarrow{\delta}(l + l^*, \rho[+\delta] + \nu[+\delta])\) in \(\pi'\).

It is easy to prove that \(\pi'\) is a correct run of \(M'\) taking into account that

- edges originating from \(X\) are in no way influenced by the behaviour of \(G_{\text{div}}\);
- the edge of \(G_{\text{div}}\) is executed whenever \(k \cdot v_D\) has elapsed \((k > 0)\).

Also, it follows that \(\overline{\pi}\) and \(\pi'\) have exactly the same set of valid positions and that for each valid position \(\langle i, t \rangle\)

\[
\Xi_{\pi}(i, t) = \langle l, \rho \rangle \iff \exists \nu . \Xi_{\pi'}(i, t) = \langle l + l^*, \rho + \nu \rangle \tag{4.4}
\]

Thus at each position the state component originating from \(X\) is exactly the same in both runs. Furthermore, for each valid position \((i, t)\),

\[
\Delta_{\pi}(i, t) = \Delta_{\pi'}(i, t) \tag{4.5}
\]

Fairness predicates \(f_i \in F\) only refer to the part of the state that originates from \(X\). This means that a fairness predicate is infinitely often satisfied along \(\pi\) if and only if it is infinitely often satisfied along \(\pi'\). As a consequence \(\pi'\) is F-fair. Because of theorem 4.1, \(\pi'\) is also \(F_{\text{div}}\)-fair, and thus \(F\)-fair. Furthermore, \(\pi'\) is divergent because \(\pi\) is divergent and \(G_{\text{div}}\) cannot block the progress of time. Remains to prove that \(\pi' \models_{M', \xi} \phi_1 U \phi_2\), given that \(\overline{\pi} \models_{M, \xi} \phi_1 U \phi_2\). This follows directly from equations 4.4, 4.5 and the induction hypothesis.

For the other direction we can apply the same approach. Given a run \(\pi' \in \overline{\Pi}_{M'}^F\) that satisfies \(\phi_1 U \phi_2\), we can construct a run \(\pi \in \overline{\Pi}_{M}^F\) by executing discrete transitions and time transitions at exactly the same time as on \(\pi'\), but replacing the discrete transitions from \(G_{\text{div}}\) with zero time transitions. Using that we can prove the \(\leftrightarrow\) case, similar to the \(\Rightarrow\) case. Details are omitted here.

- \(\phi_1 AU \phi_2\)

This can be proved in the same way as for \(\phi_1 EU \phi_2\), now proving that

\[
\exists \pi \in \overline{\Pi}_{M}^F(\langle l, \rho \rangle) . \pi \not\models_{M, \xi} \phi_1 U \phi_2 \iff
\exists \pi' \in \overline{\Pi}_{M'}^F(\langle l + l^*, \rho + \nu \rangle) . \pi \not\models_{M', \xi} \phi_1 U \phi_2
\]

The details are not shown here.

- For the boolean operators and the property update, the proofs are immediate \(\square\)

The theorems above allows us to interpret TCTL properties over the set of all fair runs, instead of the set of divergent runs, by pushing the check for divergence into the fairness condition. In this way, we disposed of the divergence aspect of TCTL for which there is no counterpart when evaluating CTL properties on abstract models.

### 4.2.3 Property variables

Unlike TCTL, CTL does not have property variables. We could easily extend CTL with property variables, but this would not solve our problem, since the heart of the matter is
that there is no way to interpret expressions on property variables over abstract models. In abstract models, the exact timing of the original explicit model is lost, while the clock variables in properties still refer to concrete values. But also non-clock property variables are a problem. The fundamental issue is that one cannot interpret atomic properties that refer to property variables on an abstract system model, because concrete property variables values cannot be related to the state space of the abstract model. In fact, we would want the property variables to be also included into the abstraction.

Property variables are not essentially different from system variables. Therefore, it seems logical to somehow add the property variables to the system model. When viewing property variables as a part of the state space, then the value of a property \( u \phi \) in some state can be interpreted as the value of \( \phi \) in an adapted state, namely the one in which at the current state, the update \( u \) is performed. More precisely, suppose we would augment the original state space \( M \) with the property variables, resulting in a new state space \( M' \). Instead of states \( \langle l, \rho \rangle \), \( M' \) would have states \( \langle l, \rho + \xi \rangle \) where \( \xi \) is a valuation for the property variables. Then \( \langle l, \rho + \xi \rangle \models_{M'} u \phi \iff \langle l, \rho + \xi[u] \rangle \models \phi \).

One could view the augmented state space as a set of subspaces, each subspace being an extension of the original state space, but with different values for the property variables. Transitions with property updates form the links between the subspaces. Such an augmented state space can be created by adding a small automaton to the system that has edges that realize the transitions from one subspace to another.

As an example, suppose we have to verify a TCTL property \( AG u. AF p \) on an XTG \( X \). We extend \( X \) to an XTG \( X' \) by adding the automaton \( u g \) that is depicted in figure 4.5. The global state space that results from this extension can be seen as a pair of linked subspaces in which for each state \( s \) in the first state space there is a state \( s' \) in the second state space which is the result of applying the update \( u \) in \( s \). This means that \( s \models u. AF p \iff s' \models AF p \). The first state space, associated with \( l_0 \), is used for the \( AG \) property, while the second state space is used for the \( AF \) property. Both subspaces are "extensions" of the original state space of \( X \). Figure 4.6 illustrates the state space of our example.
This means that the property update in the property has to be replaced by a transition to a new subspace. This can be done using the EX operator of CTL. Suppose we extend TCTL with an EX operator. Then for our example we could transform the TCTL property \( \phi = AG u . AF p \) to the property \( \phi' = AG (u \otimes l_0 \Rightarrow EX (u \otimes l_1 \land AF p)) \). This enforces that the AG property is evaluated in the subspace associated with \( l_0 \), while the AF property is evaluated in the \( l_1 \) subspace. The new property specifies that for each state in the original state space (i.e. \( AG \)), there is a successor state which results from applying update \( u \), in which \( AF p \) is satisfied.

The newly obtained property can now be evaluated in the state space of the system that results after extending the original system with \( u \otimes \), resulting in an equivalent verification problem, now without property updates. This leads to an approach in which the original system is extended with additional structure and a new property that does not contain updates, such that the resulting verification problem can be used to solve the original verification problem.

An intermediate formalism is used to specify properties like \( \phi' \) in the example above. This formalism, called D-CTL can be seen as an intermediate formalism between CTL and TCTL. In fact, D-CTL is what one would get if one would interpret CTL formulas over timed transition systems instead of labelled transition systems. A D-CTL formula is generated by the same syntax as CTL formula. The semantics of a D-CTL is very similar to that of CTL, except that — due to the denseness of the model — \( \phi_1 U \phi_2 \) is interpreted over runs rather than paths, like in TCTL. On the other hand, D-CTL can be seen as a variation of TCTL in which property variables and the divergence condition are removed, and a next-state operator is added.

**Definition 4.4 (syntax and semantics of D-CTL).** D-CTL formulas for an XTG system \( X \) are generated by the following syntax:

\[
\phi ::= b \mid le \mid \phi \lor \phi \mid \neg \phi \mid \phi AU \phi \mid \phi EU \phi \mid EX \phi
\]

where \( b \) is a boolean value expression and \( le \) is a location expression.

Let \( M = (S, S_0, T) \) be the timed transition system corresponding to an XTG system \( X \), and \( F \) be a fairness condition for \( X \). Then the semantics of D-CTL is a satisfaction relation \( \models_F M \), defined as follows.

\[
(\langle l, \rho \rangle) \models_F M b \iff \forall V. [b](\rho) = \text{true}
\]

\[
(\langle l, \rho \rangle) \models_F M le \iff \mathcal{L}(le)(l) = \text{true}
\]

\[
(\langle l, \rho \rangle) \models_F M \phi_1 \lor \phi_2 \iff (\langle l, \rho \rangle) \models_F M \phi_1 \text{ or } (\langle l, \rho \rangle) \models_F M \phi_2
\]

\[
(\langle l, \rho \rangle) \models_F M \neg \phi_1 \iff \text{not } (\langle l, \rho \rangle) \models_F M \phi_1
\]

\[
(\langle l, \rho \rangle) \models_F M \phi_1 AU \phi_2 \iff \forall \pi \in \Pi_F M (\langle l, \rho \rangle) \cdot \pi \models_F M \phi_1 U \phi_2
\]

\[
(\langle l, \rho \rangle) \models_F M \phi_1 EU \phi_2 \iff \exists \pi \in \Pi_F M (\langle l, \rho \rangle) \cdot \pi \models_F M \phi_1 U \phi_2
\]

\[
(\langle l, \rho \rangle) \models_F M EX \phi \iff \exists \langle l', \rho' \rangle \in \text{succ}_T (\langle l, \rho \rangle) \cdot \langle l', \rho' \rangle \models_F M \phi
\]

where

\[
\pi \models_F M \phi_1 U \phi_2 \iff \text{there is a position } (i, t) \text{ such that } \Xi_{\pi} (i, t) \models_F M \phi_2 \text{ and for all } \langle i', t' \rangle \ll (i, t), \Xi_{\pi} (i, t') \models_F M \phi_1 \lor \phi_2
\]

Now we are ready to define the above discussed transformations. Definition 4.5 defines the property variable graph that is to be added. Definition 4.6 defines the transformation of the TCTL property to the required D-CTL property.

**Definition 4.5 (property variable graph).** Let \( \langle PV, PV_C, PV_P, \phi \rangle \) be a property specification. The property variable graph corresponding to this property specification is an XTG \( \langle V, V_C, V_P, \rho_0, L, l_0, I, E, U \rangle \) where
Abstract models for TCTL verification

- \( V = PV, V_C = PV_C, V_P = PV_P \)
- \( \xi_0(v) = \begin{cases} e & \text{if } \langle v, e \rangle \in \text{initU}(\phi) \\ \text{any value} & \text{otherwise} \end{cases} \)
- \( L = \{ l_\psi \mid \psi \in (\text{subprop}_\text{upd}(\text{stripU}(\phi)) \cup \text{stripU}(\phi)) \} \)
- \( l_0 = l_\phi' \) where \( \phi' = \text{stripU}(\phi) \)
- \( \forall l \in L . \ I(l) = \text{true} \)
- \( \langle l_\psi_1, \text{true}, u, l_\psi_2 \rangle \in E \) if and only if \( \text{subprop}_\text{upd}(\psi_1) = \text{subprop}_\text{upd}(\psi_2) \cup \{u.\psi_2\} \)
- \( \forall e \in E . \ U(e) = \text{false} \)

Furthermore, let \( \text{loc}(\phi') \) be a function that assigns to each \( \phi' \in \text{subprop}(\phi) \) the corresponding location in the property variable graph:

\[
\text{loc}(\phi') = \begin{cases} 
  l_{\phi''} & \text{if } l_{\phi''} \in L \land \phi' \in \text{subprop}(\phi'') \land \text{subprop}_\text{upd}(\phi') = \text{subprop}_\text{upd}(\phi'') \\
  l_{\text{stripU}(\phi)} & \text{otherwise}
\end{cases}
\]

Thus the property variable graph has one location for each property update and an initial location representing the complete property. Each location except the initial location has one incoming edge which is labelled with the update of the update subproperty after which the location is named. Thus, it is always the case that for each subproperty \( u.\phi' \) of \( \phi \), there is an edge \( l_{\text{loc}(u.\phi')} \xrightarrow{u} l_{\text{loc}(\phi')} \). Figure 4.7 shows the property variable graph corresponding to the property \( \phi = \text{EF} d = 0 . \text{AG}(d < 10 \Rightarrow c := 0 . \text{AF}(p \land c < 5)) \). In that figure \( \phi' = d := 0 . \text{AG}(d < 10 \Rightarrow c := 0 . \text{AF}(p \land c < 5)) \) and \( \phi'' = c := 0 . \text{AF}(p \land c < 5) \) are subproperties of \( \phi \).

\[\text{Figure 4.7: An example property variable graph}\]

The transformation from TCTL to D-CTL is given by a function \( \Gamma \), which is defined below.

**Definition 4.6 (From TCTL to D-CTL).** Given a TCTL property \( \phi, \psi = \Gamma(\phi) \) is inductively defined as follows:

\[
\begin{align*}
\Gamma(\phi_1 \text{EU} \phi_2) &= \Gamma(\phi_1) \text{EU} (\text{G}_\text{upd}@\text{loc}(\phi_2) \land \Gamma(\phi_2)) \\
\Gamma(\phi_1 \text{AU} \phi_2) &= \Gamma(\phi_1) \text{AU} (\neg\text{G}_\text{upd}@\text{loc}(\phi_2) \lor \Gamma(\phi_2)) \\
\Gamma(\phi_1 \lor \phi_2) &= \Gamma(\phi_1) \lor \Gamma(\phi_2) \\
\Gamma(\neg \phi_1) &= \neg \Gamma(\phi_1) \\
\Gamma(u.\phi_1) &= \text{EX} (\text{G}_\text{upd}@\text{loc}(\phi_1) \land \Gamma(\phi_1)) \\
\Gamma(p) &= p \\
\Gamma(l_e) &= l_e
\end{align*}
\]

The following theorem formalizes the correctness of the transformations.

**Theorem 4.3 (Removing property variables).** Let \( X \) be an XTG system, let \( F \) be a fairness property for \( X \) such that all fair runs are divergent and let \( \phi \) be a TCTL property.
for $X$. Furthermore, let $G = (V, \cdot, -\cdot, \cdot, \cdot, E, -)$ be the global graph of $X$, and let $M = (S, S_0, T)$ be the timed transition system induced by $G$. Let $ug = (\cdot, -\cdot, \xi_0, \{l^*\}, -\cdot, E^*$, $-\cdot$) be the property variable graph corresponding to $\phi$. Let $X$ be the extension of $X$ with $ug$. Furthermore, let $\overline{G} = (\overline{V}, \cdot, -\cdot, \cdot, -\cdot, \overline{E}, -\cdot)$ be the global graph of $X$, and let $\overline{M} = (\overline{S}, \overline{S}_0, \overline{T})$ be the timed transition system induced by $\overline{G}$. Then for any state $(l, \phi) \in S$,

$$\langle l, \rho \rangle \models^F_M \phi \iff \langle l + l_0, \rho + \xi_0 \rangle \models^F_{\overline{M}} \Gamma(stripU(\phi))$$

**Proof.** We first prove two intermediate properties. Let $\langle l, \rho \rangle$ and $\langle l', \rho' \rangle$ be states of $M$ and for arbitrary $l^*$ and $\xi$, let $\langle l + l^*, \rho + \xi \rangle$ be a state of $\overline{M}$ ($\xi$ being a valuation for the sole variable of $ug$). Then for any $e = \langle l, g, u, l' \rangle \in E$ and $\overline{e} = \langle l + l^*, g, u, l' + l^* \rangle \in \overline{E}$ and any $\delta > 0$

$$\langle l, \rho \rangle \xrightarrow{e}_T \langle l', \rho' \rangle \iff \langle l + l^*, \rho + \xi \rangle \xrightarrow{\overline{e}} (l' + l^*, \rho' + \xi)$$

(4.6)

$$\langle l, \rho \rangle \xrightarrow{\delta}_T \langle l', \rho' \rangle \iff \langle l + l^*, \rho + \xi \rangle \xrightarrow{\delta} (l' + l^*, \rho' + \xi[+\delta])$$

(4.7)

We start with equation 4.6, proving the two directions separately, starting with $\Rightarrow$. If $e \in E$, then according to the semantics of parallel composition there is a corresponding $\overline{e} \in \overline{E}$. If $e$ is enabled in $(l, \rho)$ then $\overline{e}$ must enabled in $(l + l^*, \rho + \xi)$ because $e$ and $\overline{e}$ share the same guard $g$ which is both cases evaluated against $\rho$. Also, $e$ and $\overline{e}$ share the same update $u$, which only refers to variables in $V$ and only assigns values to variables in $V$. Therefore (since $\rho$ is the valuation for $V$) executing $e$ and $\overline{e}$ in $(l, \rho)$ and $(l + l^*, \rho + \xi)$, respectively, leads to the same transformation from $\rho$ to $\rho'$. Also, $\xi$ is not affected by the update of $\overline{e}$. Finally $l'$ and $l' + l^*$ share the same invariant, because $ug$ has no location invariants. Thus if $(l', \rho')$ is an admissible location, then so is $(l' + l^*, \rho' + \xi)$. This completes the $\Rightarrow$ case of 4.6. For $\Leftarrow$, $\overline{e} \in \overline{E}$ cannot originate from the update graph since an update graph does not have edges with equal source and destination location. This means that it must originate from $X$. Therefore, from the semantics of parallel composition it follows that there has to be a corresponding edge $e \in E$ in the global graph of $X$. Using the same arguments as for the first direction, $e$ must be enabled if $\overline{e}$ is enabled and the execution of $e$ leads to the same transformation on $\rho$, while leaving $\xi$ untouched. This proves the $\Leftarrow$ case and thus equation 4.6.

We continue with equation 4.7, starting with the $\Rightarrow$ direction. If $\langle l, \rho \rangle \xrightarrow{\delta}_T (l', \rho')$ then from the semantics of $XTG$ it follows that $l = l'$ and $\rho' = \rho[+\delta]$. It also means that for any $d$ with $0 \leq d < \delta$, in $\rho[+d]$ the invariant of $l$ is not violated and no urgent edges are enabled. Because all invariants of $ug$ are set to $true$, it also follows that for all $d$ with $0 \leq d < \delta$, the invariant of $l + l^*$ is not violated in $(\rho + \xi)[+d]$. Furthermore, because $ug$ does not contain urgent edges, and it cannot influence guards of edges from $X$ nor synchronize with edges from $X$, it also follows that no urgent edges can be enabled in $(l + l^*, \rho + \xi)[+d]$, for each $d$ with $0 \leq d < \delta$. From the semantics it then follows that also in $(l + l^*, \rho + \xi)$ a $\delta$ transition can be taken. This means that $(l + l^*, \rho + \xi) \xrightarrow{\delta} (l + l^*, \rho[+d] + \xi) = (l' + l^*, \rho'[+d] + \xi[+d])$. The other direction follows almost directly. If $(l + l^*, \rho + \xi) \xrightarrow{\delta} (l' + l^*, \rho' + \xi[+\delta])$, then it follows that for each $d$ with $0 \leq d < \delta$, the invariant of $l$ is not violated in $\rho[+d]$ and no urgent edges are enabled in $\rho[+d]$, and therefore $\langle l, \rho \rangle \xrightarrow{\delta}_T (l', \rho')$. This proves equation 4.7.

Using equations 4.6 and 4.7 it can now be proved that given states $(l_0, \rho_0) \in S$ and
\[ \langle l_0 + l^*, \rho_0 + \xi \rangle \in \mathcal{S}, \]
\[ \exists \pi = \langle l_0, \rho_0 \rangle \rightarrow_T \langle l_1, \rho_1 \rangle \rightarrow_T \cdots \in \Pi_M^F \iff \]
\[ \exists \pi = \langle l_0 + l^*, \rho_0 + \xi[+\Delta\tau(0,0)] \rangle \rightarrow_T \langle l_1 + l^*, \rho_1 + \xi[+\Delta\tau(0,0)] \rangle \rightarrow_T \cdots \in \Pi_M^F \tag{4.8} \]

By induction on the length of runs it is first proved that equation 4.8 holds when disregarding fairness. Suppose that corresponding prefixes \( \langle l_0, \rho_0 \rangle \rightarrow_T \cdots \langle l_i, \rho_i \rangle \) and \( \langle l_0 + l^*, \rho_0 + \xi[+\Delta\tau(0,0)] \rangle \rightarrow_T \cdots \langle l_i + l^*, \rho_i + \xi[+\Delta\tau(i,0)] \rangle \) already exist. Then for discrete transitions:

\[ \langle l_i, \rho_i \rangle \xrightarrow{e} \langle l_{i+1}, \rho_{i+1} \rangle \iff \text{[equation 4.6 and the induction hypothesis]} \]
\[ \langle l_i + l^*, \rho_i + \xi[+\Delta\tau(i,0)] \rangle \xrightarrow{e} \langle l_{i+1} + l^*, \rho_{i+1} + \xi[+\Delta\tau(i,0)] \rangle \]
\[ \iff [\Delta\tau(i+1,0) = \Delta\tau(i,0) \text{ because no time elapses}] \]
\[ \langle l_i + l^*, \rho_i + \xi[+\Delta\tau(i,0)] \rangle \xrightarrow{e} \langle l_{i+1} + l^*, \rho_{i+1} + \xi[+\Delta\tau(i+1,0)] \rangle \]

For \( \delta \) transitions:

\[ \langle l_i, \rho_i \rangle \xrightarrow{\delta} \langle l_{i+1}, \rho_{i+1} \rangle \iff \text{[equation 4.7 and the induction hypothesis]} \]
\[ \langle l_i + l^*, \rho_i + \xi[+\Delta\tau(i,0)] \rangle \xrightarrow{\delta} \langle l_{i+1} + l^*, \rho_{i+1} + \xi[+(\Delta\tau(i,0) + \delta)] \rangle \]
\[ \iff [\Delta\tau(i+1,0) = \Delta\tau(i,0) + \delta] \text{ since \( \delta \) time elapses in transition } i + 1] \]
\[ \langle l_i + l^*, \rho_i + \xi[+\Delta\tau(i,0)] \rangle \xrightarrow{\delta} \langle l_{i+1} + l^*, \rho_{i+1} + \xi[+\Delta\tau(i+1,0)] \rangle \]

which completes the proof of equation 4.8, disregarding fairness. The preservation of fairness between \( \pi \) and \( \tau \) follows from the fact that for any fairness predicate \( f \), for each \( i \geq 0 \), \( \langle l_i, \rho_i \rangle \) satisfies \( f \) if and only if \( \langle l_i + l^*, \rho_i + \xi[+\Delta\tau(i,0)] \rangle \) satisfies \( f \), because \( f \) can only refer to the state aspect they have in common (namely \( l_i \) and \( \rho_i \)). This proves equation 4.8 since for any fair run in one of the two models, there is a corresponding fair run in the other model.

Using this result, we can now prove by induction on the structure of \( \phi \) that for each \( \phi' \in \text{subprop}(\phi) \) and for any pair of states \( \langle l, \rho \rangle \) and \( \langle l + \text{loc}(\phi'), \rho + \xi \rangle \) in \( M \) and \( \overline{M} \), respectively, that

\[ \langle l, \rho \rangle \models_{M, \xi} \phi' \iff \langle l + \text{loc}(\phi'), \rho + \xi \rangle \models_{\overline{M}} \Gamma(\phi') \tag{4.9} \]

The base cases concerns the atomic properties. There are two cases.

- **Boolean value expressions.**
  \[ \langle l, \rho \rangle \models_{M, \xi} b \iff \text{[TCTL semantics]} \]
  \[ \forall \nu \forall [b][\rho(\rho + \xi) = \text{true} \iff \text{[D-CTL semantics]}] \]
  \[ \langle l + \text{loc}(b), \rho + \xi \rangle \models_{\overline{M}} b \iff \text{[definition of } \Gamma \text{]} \]
  \[ \langle l + \text{loc}(b), \rho + \xi \rangle \models_{\overline{M}} \Gamma(b) \]

- **Location expressions.**
  \[ \langle l, \rho \rangle \models_{M, \xi} g_x @ l_y \iff \text{[TCTL semantics]} \]
  \[ \mathcal{L}_x [g_x @ l_y](l) = \text{true} \iff \text{[D-CTL semantics]} \]
  \[ \langle l + \text{loc}(g_x @ l_y), \rho + \xi \rangle \models_{\overline{M}} g_x @ l_x \iff \text{[definition of } \Gamma \text{]} \]
  \[ \langle l + \text{loc}(g_x @ l_y), \rho + \xi \rangle \models_{\overline{M}} \Gamma(g_x @ l_x) \]

In addition, this means that equation 4.9 also holds for fairness predicates, since these are defined as atomic properties.

The induction case:
• $\phi_1 \lor \phi_2$:
  \[
  (l, \rho) \models^F_{M, \xi} \phi_1 \lor \phi_2 \iff \text{[TCTL semantics]}
  \]
  \[
  (l, \rho) \models^F_{M, \xi} \phi_1 \text{ or } (l, \rho) \models^F_{M, \xi} \phi_2 \iff \text{[induction hypothesis]}
  \]
  \[
  (l + \text{loc}(\phi_1), \rho + \xi) \models^F_{M, \xi} \Gamma(\phi_1) \text{ or } (l + \text{loc}(\phi_2), \rho + \xi) \models^F_{M, \xi} \Gamma(\phi_2)
  \]

• $\phi_1 \text{ EU } \phi_2$:
  \[
  \exists \pi = (l_0, \rho_0) \longrightarrow^l (l_1, \rho_1) \longrightarrow^l \cdots \in \vec{\Pi}_M^F \iff
  \exists \pi = (l_0 + \text{loc}(\phi_1 \text{ EU } \phi_2), \rho_0 + \xi[+\Delta(0,0)]) \longrightarrow^l (l_1 + \text{loc}(\phi_1 \text{ EU } \phi_2), \rho_1 + \xi[+\Delta(1,0)])
  \]

Here we also used the fact that $\pi \in \Pi_M^F \iff \pi \in \vec{\Pi}_M^F$ because all fair runs are divergent. Now from the proof of equation 4.8 it can be seen that both runs must have the same set of valid positions. We can then show that for any position $(i, t)$ on $\pi$ and $\pi'$ and for each $\phi \in \{\phi_1, \phi_2\}$, the state in $\pi$ at $(i, t)$ satisfies $\phi$ if and only if the state in $\pi'$ at $(i, t)$ satisfies $\phi$.

Using this result it follows from the semantics of CTL and D-CTL that,

\[
\pi \models^F_{M, \xi} \phi_1 \text{ U } \phi_2 \iff \pi' \models^F_{M, \xi} \Gamma(\phi_1) \text{ U } \Gamma(\phi_2)
\]

Now since in each state on $\pi'$, $\text{ug}@\text{loc}(\phi_1 \text{ EU } \phi_2)$ is satisfied, and since $\text{loc}(\phi_1 \text{ EU } \phi_2) = \text{loc}(\phi_2)$, it follows for any $l_0$, $\rho_0$ and $\xi$, that

\[
(l_0, \rho_0) \models^F_{M, \xi} \phi_1 \text{ EU } \phi_2 \iff (l_0 + \text{loc}(\phi_1 \text{ EU } \phi_2), \rho_0 + \xi) \models^F_{M, \xi} \Gamma(\phi_1) \text{ EU } (\Gamma(\phi_2) \land \text{ug}@\text{loc}(\phi_2))
\]

• $\phi_1 \text{ AU } \phi_2$:
  This case is analogous to $\phi_1 \text{ EU } \phi_2$. Now it has to be proved that for $\pi$ and $\pi'$,

\[
\pi \not\models^F_{M, \xi} \phi_1 \text{ U } \phi_2 \iff \pi' \not\models^F_{M, \xi} \Gamma(\phi_1) \text{ U } \Gamma(\phi_2)
\]

This is not shown here.

• $u, \phi_1$:
  \[
  (l, \rho) \models^F_{M, \xi} u, \phi_1 \iff \text{[TCTL semantics]}
  \]
  \[
  (l, \rho) \models^F_{M, \xi[\nu]} \phi_1 \iff \text{[induction hypothesis]}
  \]
  \[
  (l + \text{loc}(\phi_1), \rho + \xi[\nu]) \models^F_{M, \xi[\nu]} \Gamma(\phi_1) \iff \text{[semantics of u@loc(\phi_1)]}
  \]
  \[
  (l + \text{loc}(\phi_1), \rho + \xi[\nu]) \models^F_{M, \xi[\nu]} \text{ug}@\text{loc}(\phi_1) \land \Gamma(\phi_1)
  \]
  \[
  \iff \text{[structure of the update graph, see discussion below]}
  \]
  \[
  (l + \text{loc}(u, \phi_1), \rho + \xi) \models^F_{M, \xi[\nu]} \text{EX} (\text{ug}@\text{loc}(\phi_1) \land \Gamma(\phi_1)) \iff \text{[definition of \Gamma]}
  \]
  \[
  (l + \text{loc}(u, \phi_1), \rho + \xi) \models^F_{M, \xi[\nu]} \Gamma(\phi_1)
  \]
Abstract models for TCTL verification

The step from the fourth to the fifth equation deserves some explanation. From the definition of the property update graph it follows that it must have an unguarded edge \( e \) from \( loc(u, \phi_1) \) to \( loc(\phi_1) \) with an update equal to \( u \). As a consequence, there is a transition \((l + loc(u, \phi_1), \rho + \xi) \xrightarrow{e} (l + loc(\phi_1), \rho + \xi[u])\) in \( T \), which explains the introduction of the EX.

This concludes the proof of equation 4.9. Now we can prove:

\[
\langle l, \rho \rangle \models^F_M \phi \iff \langle l, \rho \rangle \models^F_{M, l_0} \text{ stripU}(\phi) \text{ where } \xi_0(v) = \begin{cases} e & \text{if } \langle v, e \rangle \in \text{initU}(\phi) \\ \text{any value} & \text{otherwise} \end{cases}
\]

\[
\langle l + \text{loc}(\text{stripU}(\phi)), \rho + \xi_0 \rangle \models^F_M \Gamma(\text{stripU}(\phi)) \iff \langle l + l_0, \rho + \xi_0 \rangle \models^F_M \Gamma(\text{stripU}(\phi))
\]

which completes the proof.

4.2.4 Bisimulation abstraction on D-CTL

From sections 4.2.2 and 4.2.3 we see that a TCTL verification problem can be transformed into a D-CTL verification problem, by dealing with divergence and property variables. The reason for this transformation is that D-CTL is more compatible with time-abstracting bisimulation abstraction. To be precise, D-CTL properties "carry over" to CTL properties on abstract models based on time-abstracting bisimulation.

The underlying reason is that states that are equivalent with respect to some time-abstracting bisimulation, satisfy the same D-CTL formulas. As was discussed already, the same is not true for TCTL properties.

**Proposition 4.4 (time-abstracting bisimulation and D-CTL).** Consider the timed transition system \( M = \langle S, S_0, T \rangle \). For any D-CTL formula \( \phi \) and any two states \( s, s' \in S \), if \( s \) and \( s' \) are time-abstracting bisimulation equivalent then

\[
s \models^F_M \phi \iff s' \models^F_M \phi
\]

We will not prove this proposition here, since it is not directly needed. We are primarily interested in building abstract models based on bisimulation, in particular, reducing verification problems on timed transition systems to verification problems on more abstract labelled transition systems. In our case, a D-CTL property on a timed transition system is transformed to a CTL property on the labelled transition system that results from the abstraction. In fact, the resulting CTL property is syntactically exactly the same, except that in our version of CTL atomic properties are abstractly defined as sets of states instead of predicates.

**Definition 4.7 (from D-CTL to CTL).** Given a D-CTL property \( \phi \) for a model \( M = \langle S, S_0, T \rangle \) and an abstract model \( B = \langle A, A_0, T_A \rangle \) for \( M \), let \( A(\phi) \) denote the CTL property in which each atomic property \( p \) in \( \phi \) is replaced by \( A(p) = \{ a \in A \mid \exists s \in a . \ s \models_M p \} \). Also, given a fairness predicate \( F = \{ f_1, \ldots, f_n \} \), we define \( A(F) = \{ A(f_1), \ldots, A(f_2) \} \) where \( A(f_i) = \{ a \in A \mid \exists s \in a . \ s \models_M f_i \} \).

Thus, the conversion from \( \phi \) to \( A(\phi) \) replaces predicates by sets of abstract states of which the concrete states satisfy the predicate.
Chapter 4 Evaluating TCTL properties

Theorem 4.5 (verification of abstract models). Let $M = \langle S, S_0, T \rangle$ be a timed transition system corresponding to an XTG system $X$. Let $\phi$ be a D-CTL formula for $X$ and $F = \{ f_1, \ldots, f_n \}$ be a fairness property for $X$. Let $B = \langle A, A_0, T_A \rangle$ be a labelled transition system that is an abstract model for $M$, such that $B$ respects the atomic properties of $\phi$ and the fairness predicates in $F$. Then for any $s \in S$

$$s \models_M^F \phi \iff \text{abs}(s) \models_B^{\phi} A(\phi)$$

(4.11) □

Proof. We first prove two pairs of supporting properties. Assume that equation 4.11 holds for $\phi_1$ and $\phi_2$, i.e. for any $s \in S$,

$$s \models_M^F \phi_1 \iff \text{abs}(s) \models_B^{\phi_1} \phi_1$$

and

$$s \models_M^F \phi_2 \iff \text{abs}(s) \models_B^{\phi_2} \phi_2$$

(4.12)

1. Let $\pi = s_0 \rightarrow s_1 \rightarrow \cdots$ be a run in $M$, and $\psi = \text{abs}(s) = a_0 \rightarrow a_1 \rightarrow \cdots$ be the abstract path in $B$ corresponding to $\pi$ (theorem 3.1). Then we prove that (given equation 4.12)

$$\pi \text{ is } F\text{-fair} \Rightarrow \psi \text{ is } A(F)\text{-fair}$$

(4.13)

$$\pi \models_M^{F} \phi_1 U \phi_2 \Rightarrow \psi \models_B^{A(F)} \phi_1 U \phi_2$$

(4.14)

We start with 4.13, assuming that $\pi$ is $F$-fair. As the abstract model respects the fairness predicates, we know for each state $s_i$ on $\pi$ and each fairness predicate $f_j \in F$, that $s_i \models f_j \Rightarrow \text{abs}(s_i) \models A(f_j)$. From theorem 3.1 we know that for each state $s_i$, $\text{abs}(s_i)$ is on $\psi$. Thus if $f_j$ is infinitely often satisfied along $\pi$, then $A(f_j)$ is infinitely often satisfied along $\psi$. As a consequence, $\psi$ is $A(F)$-fair.

Continuing with 4.14, $\pi \models_M^{F} \phi_1 U \phi_2$ means that there exists a position $\langle i, t \rangle$ on $\pi$ such that $s_i[t+] \models_M^F \phi_1$ and for all $\langle i', t' \rangle \ll \langle i, t \rangle$, $s_{i'}[t'+] \models_M^F \phi_1 \lor \phi_2$. Then according to theorem 3.1, $\text{abs}(s_i[t+]) = a_i$ is a state on $\psi$. From the assumption (equation 4.12) it follows that $a_i \models_B^{A(F)} \phi_2$. Now let $a_k$ be the first state on $\psi$ such that $a_k \models_B^{A(F)} \phi_2$ (which exists because $a_i$ exists). From theorem 3.1 we know that for each $a_m$ on $\psi$ with $m < k$, there exists a position $\langle j, u \rangle$ on $\pi$ such that $a_m = \text{abs}(s_j[u+])$ and $\langle j, u \rangle \ll \langle i, t \rangle$. Since we know that $s_j[u+] \models \phi_1 \lor \phi_2$ it follows from the assumption in equation 4.12 that $a_m \models_B^{A(F)} \phi_1 \lor \phi_2$. Because $a_k$ was defined to be first state on $\psi$ to satisfy $A(\phi_2)$ it follows that $\forall m \in \{0, \ldots, k-1\}, a_m \models_B^{A(F)} \phi_1$. Together with the fact that $a_k \models_B^{A(F)} \phi_2$ it follows that $\psi \models_B^{A(F)} \phi_2$.

2. Let $\pi = s_0 \rightarrow a_1 \rightarrow \cdots$ be a path in $B$ and let $\pi = s_0 \delta_0 \rightarrow s_1 \delta_1 \rightarrow \cdots$ be the concrete run corresponding to $\pi$ starting at $s_0$ (that exists according to theorem 3.1). Then we prove that

$$\psi \models_B^{A(F)} \phi_1 U \phi_2 \Rightarrow \pi \text{ is } F\text{-fair}$$

(4.15)

$$\psi \models_B^{A(F)} \phi_1 U \phi_2 \Rightarrow \pi \models_M^{F} \phi_1 U \phi_2$$

(4.16)

From theorem 3.1 we now that for each state $a_i$ on $\psi$ there is a state $s_i \in a_i$ on $\pi$. From the definition of $\phi$ we know that for any $A(f_j) \in A(F)$, $a_i \models A(f_j) \Rightarrow s_i \models f_j$. As a consequence, if $A(f)$ is infinitely often satisfied on $\psi$, then $f$ must be infinitely often satisfied on $\pi$. Thus if $\psi$ is fair then also $\pi$ is fair, proving equation 4.15.

Continuing with equation 4.16, if $\psi \models_B^{A(F)} \phi_1 U \phi_2$, then there exists an $a_k$ on $\psi$, for which $a_k \models_B^{A(F)} \phi_2$ and for all $\forall m \in \{0, \ldots, k-1\}, a_m \models_B^{A(F)} \phi_1$. From theorem 3.1 we know that for all $i$, $\text{abs}(s_i) = a_i$. Therefore, we know from our assumption
in equation 4.12 that $s_k \models_F \phi_2$. Remains to prove that $\forall (j, t) \ll \langle k, 0 \rangle \cdot s_j[+t] \models_F \phi_1 \lor \phi_2$. From theorem 3.1 it follows that for any $(j, t) \ll \langle k, 0 \rangle$, $abs(s_j[+t]) = a_m$ is a state on $\psi$, with $m \leq k$. For $m = k$ — because $a_k \models_B[A(F)] A(\phi_2)$ and the assumption of equation 4.12 — it follows that $s_j[+t] \models_F \phi_2$. For $m \in \{0, \ldots, k - 1\}$, we know that $a_m \models_B[A(F)] A(\phi_1)$ and then from the assumption of equation 4.12 it follows that $s_j[+t] \models_F \phi_1$. Summarizing, there exists a $\langle k, 0 \rangle$ on $\pi$ such that $s_k[+0] \models_F \phi_2$ and for all $(j, t) \ll \langle k, 0 \rangle$, $s_j[+t] \models_F \phi_1 \lor \phi_2$, in other words, $\pi \models_F \phi_1 U \phi_2$.

The proof is now fairly straightforward by induction on the structure of $\phi$.

• The base case concerns atomic properties $p$.

\[ s \models_M p \iff [A(p) = \{a \in A \mid \exists s \in a \cdot s \models_M p\}] \]

\[ abs(s) \in A(p) \iff [\text{CTL semantics}] abs(s) \models_B[A(F)] A(p) \]

• $\phi_1 \lor \phi_2$:

\[ s \models_M \phi_1 \lor \phi_2 \iff s \models_M \phi_1 \text{ or } s \models_M \phi_2 \iff [\text{induction hypothesis}] \]

\[ abs(s) \models_B[A(F)] A(\phi_1) \text{ or } abs(s) \models_B[A(F)] A(\phi_2) \iff abs(s) \models_B[A(F)] A(\phi_1 \lor \phi_2) \]

• $\neg \phi_1$: similar to the case above.

• $EX \phi_1$:

\[ s \models_M EX \phi_1 \iff \exists s' \in succ_T(s) \cdot s' \models \phi_1 \iff [\text{induction hypothesis}] \]

\[ \exists s' \in succ_T(s) \cdot abs(s') \models_B[A(F)] A(\phi_1) \iff [\text{definition 3.4}] \]

\[ \exists a \in succ_T(a) \models_B[A(F)] A(\phi_1) \iff abs(s) \models_B[A(F)] A(EX \phi_1) \]

• $\phi_1 E \phi_2$: We prove the two directions separately, starting with $\Rightarrow$. If $s \models_M \phi_1 E \phi_2$ then there exists a $F$-fair run $\pi$ in $M$ such that $\pi \models_M \phi_1 U \phi_2$. From equations 4.13 and 4.14 it then follows that there exists an $A(F)$-fair run $\psi$ in $B$ such that $\psi \models_B[A(F)] A(\phi_1 U \phi_2)$. This means that $abs(s) \models_B[A(F)] A(\phi_1 E \phi_2)$. Note that equations 4.13 and 4.14 can be applied because from the induction hypothesis it follows that equation 4.11 holds for $\phi_1$ and $\phi_2$. The $\Leftarrow$ case can be proved by the reasoning in the other direction, using equations 4.15 and 4.16 instead of equations 4.13 and 4.14.

• $\phi_1 A \phi_2$: We prove $s \not\models_F \phi_1 A \phi_2 \iff abs(s) \not\models_B[A(F)] A(\phi_1 A \phi_2)$. For $\Rightarrow$, suppose that $s \not\models_F \phi_1 A \phi_2$. Then D-CTL semantics require that there exists an $F$-fair run $\pi$ in $M$ such that $\pi \not\models_F \phi_1 \phi_2$. From equations 4.13 and 4.14 it then follows that there exists an $A(F)$-fair run $\psi$ in $B$ such that $\psi \not\models_B[A(F)] A(\phi_1 \phi_2)$. According to the CTL semantics this means that $abs(s) \not\models_B[A(F)] A(\phi_1 A \phi_2)$. The $\Leftarrow$ case can again be proved by reasoning in the other direction.

The proves the induction and thus the theorem. \[ \Box \]

Summarizing, this section defined how TCTL specifications carry over to CTL specifications on abstract models. This was done in three steps. First, the divergence requirement was removed from TCTL by adding some structure to the system and augmenting the fairness condition. Then, it was shown how to transform a TCTL specification to a D-CTL specification on an extended model, thus removing property variables from the specification. Finally, we related D-CTL specifications to equivalent CTL specifications on abstract models based on time-abstractive bisimulation.
4.3 Deciding fair CTL properties

In the previous sections, a mechanism was defined to transform TCTL verification problems to CTL verification problems on finite state spaces. This section concentrates on solving the latter problems.

When building a model checker, one usually does not explicitly separate the evaluation approach from the exploration algorithm. However, for our purposes it is interesting to find an optimal evaluation approach. The key property of an optimal evaluation approach is that it decides a property as soon as enough information is available. In other words, once the portion of the state space that has been explored is sufficient to decide that the property of interest is either satisfied or not satisfied in an initial state, it should immediately do so. This avoids the exploration of useless parts of the state space.

To build an efficient model checker the evaluation should be done on-the-fly. Rather than first fully exploring the state space and subsequently evaluating it, evaluation should be intertwined with the exploration, thus avoiding unnecessary explorations. To allow efficient evaluation, the relevant information that can already be derived is captured in what could be called an evaluation state. For reachability analysis, the evaluation state was embodied by the valuation value. The evaluation space is defined such that it is easy to recognize if and with which result the verification problem at hand can be decided. Decision rules then define how the evaluation space is to be updated to incorporate new information (i.e. newly explored states).

The decision criteria can be specified independently from a concrete model checking algorithm. Such decision criteria can be seen as an abstract specification of an optimal decision procedure, independent of the exploration technique that is used. Given any exploration approach for a finite state space, this allows the definition of an optimal matching evaluation approach. Note that explicitly formulating general decision rules provides a nice abstract viewpoint on the evaluation of CTL properties.

In section 3.6 already an evaluation approach was constructed for the simpler problem of parametric reachability analysis. From that solution we could postulate that an evaluation approach consists of three elements:

An evaluation state: An evaluation state holds everything known already that can contribute to solving the verification problem. For parametric reachability analysis the evaluation state was a mapping from explored nodes to a value indicating if a node was known to be reachable ($T$), unreachable ($F$), or that it is not yet known whether or not the node is reachable ($U$).

Criteria for updating the evaluation state: The evaluation state is updated whenever new information becomes available. For the parametric reachability algorithm this meant setting values of nodes to TRUE or FALSE when appropriate.

Recognizing that the verification problem is solved: This is the stop criterion of the algorithm. For the parametric reachability algorithm this meant that it stops as soon as all initial nodes are decided.

As we will show, for the verification of temporal properties, the evaluation approach is also defined in terms of these three elements.

So what do we expect from decision criteria? Obviously, they have to be correct. If a state is decided to satisfy or dissatisfy a relevant (sub)property, this should indeed be the case. Furthermore, the decision rules would have to be complete. This means that if some state $s$ satisfies (or dissatisfies) $\phi$, the decision procedure eventually will decide so. Finally, as was mentioned earlier, decision rules have to be optimal. Given a partially explored
Deciding fair CTL properties

When evaluating the satisfaction of CTL properties, a hierarchical approach seems natural. This means that for each CTL operator we define decision criteria in terms of the satisfaction of the subproperties in its operands. For all operators but the temporal operators this is very straightforward. For example, a state satisfies $EX \phi$ if one of its successors satisfies $\phi$, while a state satisfies $\phi_1 \lor \phi_2$ if that state satisfies $\phi_1$ and $\phi_2$. Obviously, the challenge lies in dealing with the temporal operators: $\phi_1 EU \phi_2$ and $\phi_1 AU \phi_2$.

Given a property $\phi$, suppose that for some states in the explored part of a state space, it is already known whether or not these states satisfy $\phi$. Also suppose that for each subproperty of $\phi$, we also know for some of the states whether or not these states satisfy that subproperty. Then what we are looking for is a procedure that, given this information, indicates that explored but yet undecided states can be decided as either satisfying $\phi$ or dissatisfying $\phi$, if possible. For such a decision procedure, an evaluation state can be used that has two sets of states for each subproperty, one (called $S_T$), containing the set of states that are already known to satisfy the property and another one (called $S_F$), containing the set of states that are already known not to satisfy the property. Decision criteria evaluate the currently explored part of the state space and adds states to either $S_T$ or $S_F$.

Such an evaluation state is analogous to the evaluation state used in reachability analysis. Like for reachability analysis, the stop criterion would be that all initial states have been decided (i.e. have been added to either $S_T$ or $S_F$). The representation in terms of sets $S_T$ and $S_F$, rather than a valuation, fits better to the abstract definition approach taken in this chapter.

The remainder of this section focuses on finding the criteria for updating the evaluation state based on the above described evaluation state, such that an optimal evaluation procedure is obtained.

4.3.1 Partial models

To be able to reason about optimality, the concept of partially explored state spaces has to be made explicit. The reason for this is the fact that the knowledge of whether or not a state is fully explored is relevant. Partially explored state spaces are formalized by so-called partial models.

**Definition 4.8 (partial model).** A partial model is a tuple $\langle S^*, S^*_0, T^*, S_U \rangle$, where $S^*$ is a set of (partially or fully) explored states, $S^*_0 \subseteq S^*$ is the subset of initial states, $T^* \subseteq S^* \times Lab \times S^*$ is the set of already explored transitions, and $S_U \subseteq S^*$ is the subset of partially explored states. Given a partial model $M^* = \langle S^*, S^*_0, T^*, S_U \rangle$, a labelled transition system $M = \langle S, S_0, T \rangle$ is a completion of $M^*$, written $M \supseteq M^*$ if

- $S^* \subseteq S$, $S^*_0 = S^* \cap S_0$
- $T^* \subseteq T$
- $\{ s \in S^* \mid \exists \langle s, l, s' \rangle \in T \setminus T^* \} \subseteq S_U$

Finally, a satisfaction relation $s \models^{\mathcal{F}}_{M^*} \phi$ on partial models is defined as follows:

$$s \models^{\mathcal{F}}_{M^*} \phi \iff \forall M \supseteq M^* . \ s \models^{\mathcal{F}}_{M} \phi$$
Thus \( S^* \) has fully explored states \( (S^* \setminus S_{U}) \) and partially explored states \( (S_U) \). Partially explored states may still have some outgoing transitions that have not yet been explored, while for all fully explored states all outgoing transitions are known. A completion of a partial model is a labelled transition system that adds states and transitions, but which does not add outgoing transitions to fully explored states.

A decision procedure is optimal if for any partial model, it is complete (with respect to the satisfaction relation for partial models). In that case, a state is decided to be in a solution set \( S_T \) or \( S_F \) if and only if it would be in the corresponding solution set of any conceivable completion of the partial model.

### 4.3.2 Deciding temporal operators using least fixed points

Decision procedures can relatively easily be constructed using least fixed point characterizations of the set of states that have to be decided. As will be shown, such characterizations exists for our problems.

**Definition 4.9 (least fixed points).** Let \( S \) be a finite set of states and \( f : \mathcal{P}(S) \to \mathcal{P}(S) \). \( f \) is said to be monotonic if for any \( X, Y \in \mathcal{P}(S) \), \( X \subseteq Y \Rightarrow f(X) \subseteq f(Y) \). \( X \) is said to be a fixed point of \( f \) if \( f(X) = X \). \( X \) is a least fixed point of \( f \) if there exists no fixed point \( Y \) of \( f \) such that \( Y \subset X \).

The theorem below is a well known result [101] that enables the use of fixed point characterizations in model checking [83].

**Theorem 4.6 (least fixed point characterizations [83, 101]).** Let \( S \) be a finite set of states and \( f : \mathcal{P}(S) \to \mathcal{P}(S) \). If \( f \) is monotonic then it has a least fixed point which is equal to \( \bigcup_{i \geq 0} f^i(\emptyset) \). \( \square \)

Suppose that solution set \( sol \) of some verification problem can be characterized as a least fixed point of a monotonic function \( f : \mathcal{P}(S) \to \mathcal{P}(S) \). Now consider a decision procedure for the solution set that uses \( C \) to hold the set of states that have already been decided to be part of the solution set. Then our decision procedure could be formulated as follows:

\[
\text{for any } s \in S \setminus C, \text{ if } s \in f(C) \text{ then } C := C \cup \{s\}
\]

where \( C \) would start out as an empty set. In other words, a state can be decided to be part of the solution set \( sol \) if it is in \( f(C) \). The correctness and completeness of this procedure follows immediately. Initially \( C \subseteq sol \) since \( C \) is initially empty. Now if we assume that at some moment \( C \subseteq sol \), then if for some state \( s \in S \setminus C, s \in f(C) \), it follows that because of monotonicity \( s \in f(sol) \) and thus \( s \in sol \). This means that the decision procedure makes only correct decisions, thus that it is always the case that \( C \subseteq sol \). As for completeness, suppose that for some \( C \subseteq sol \), all \( s \in S \setminus C, s \notin f(C) \), that is, all nodes not in \( C \) cannot be decided any more using the decision procedure. In that case it follows that \( C \) is a fixed point of \( f \), and from the correctness it follows it is the least fixed point of \( f \) which means that \( C = sol \).

As an example, consider the CTL property \( \phi_1 EU \phi_2 \), for the moment disregarding fairness. It is well-known that the set of states that satisfy \( \phi_1 EU \phi_2 \) is characterized by the least fixed point of the following function [37]:

\[
f(X) = \{s \in S \mid s \models \phi_2 \lor (s \models \phi_1 \land \exists s' \in \text{succ}(s) . s' \in X)\}
\]
which leads to the following decision rule:

\[
\text{if } s \models \phi_2 \lor (s \models \phi_1 \land \exists s' \in \text{succ}(s) . s' \in C) \text{ then } C := C \cup \{s\}
\]

Thus, a state can be decided to satisfy \( \phi_1 \mathbin{EU} \phi_2 \) if it is known that either it satisfies \( \phi_2 \) or it satisfies \( \phi_1 \) and at least one of its successors is known to satisfy \( \phi_1 \mathbin{EU} \phi_2 \).

Often in model checking, fixed point characterizations of temporal operators have been used as a basis for algorithms. Here the fixed points are not used to build an algorithm, but to provide a general characterization of the evaluation aspect of such an algorithm. A higher level of abstraction is chosen, by specifying general criteria that are correct and optimal.

Note that for our example, to achieve an optimal decision procedure, a decision rule is also needed to decide states that dissatisfy \( \phi_1 \mathbin{EU} \phi_2 \). Also, note that greatest fixed points are not useful for this purpose. They cannot be used to derive decision procedures. While a function used in a least fixed point defines how to extend a set \( C \subseteq \text{sol} \) with additional states, the function used in a greatest fixed point defines how to narrow a set \( C \supseteq \text{sol} \). This means that intermediate sets \( C \) are not conclusive.

The remainder of this chapter concentrates on formulating least fixed point characterizations for the temporal operators of fair TCTL.

### 4.3.3 Deciding properties in non-fair CTL

As an introduction to the verification of fair CTL properties we discuss the somewhat simpler case of non-fair CTL properties. After that, we extend to fair CTL in the subsequent subsections. Compared to the example above, some additional complexity has to be dealt with, namely the fact that we also have to characterize the set of states that do not satisfy a property \( \phi \).

We will be needing the notion of strongly connected subgraphs, and later in this chapter, strongly connected components.

**Definition 4.10 (strongly connected subgraphs).** Given a finite graph, a strongly connected subgraph (SCS) in that graph is a subset \( C \) of vertices, such that for each pair of nodes \( n \) and \( n' \) in \( C \), \( n' \) can be reached from \( n \) by only passing through vertices in \( C \).

An SCS \( C \) is a strongly connected component (SCC) if no larger SCS containing \( C \) exists.

SCS’s correspond to cycles in the graph. An SCC is an SCS for which all cycles from any state in that SCS are captured within the SCS. The set of SCC’s of a graph defines a partition of the graph, which is the finest partition of that graph such that all cycles are captured within single SCC’s. Also the set of SCC’s can be seen as defining an acyclic quotient graph in which the SCC’s are partially ordered by means of the reachability relation.

Section 4.3.2 already gave a solution for deciding \( \phi_1 \mathbin{EU} \phi_2 \) in the absence of fairness properties. From the least fixed point given there, the following decision criterion for deciding the set of states that satisfy \( \phi_1 \mathbin{EU} \phi_2 \) was derived:

\[
\text{if } s \models \phi_2 \lor (s \models \phi_1 \land \exists s' \in \text{succ}(s) . s' \in S_T) \text{ then } S_T := S_T \cup \{s\}
\]

This result can be intuitively explained. First of all, if \( s \) satisfies \( \phi_2 \), then surely \( s \) satisfies \( \phi_1 \mathbin{EU} \phi_2 \), since each path from \( s \) trivially satisfies \( \phi_1 \mathbin{U} \phi_2 \). Secondly, if \( s \) does not satisfy
Chapter 4 Evaluating TCTL properties

φ₂ then at least it has to satisfy φ₁, since the first state of a path satisfying φ₁ U φ₂ either has to satisfy φ₁ or φ₂. Furthermore, there has to be a successor of s which does satisfy φ₁ EU φ₂. This can be explained as follows. If a state satisfies φ₁, and a successor s' satisfies φ₁ EU φ₂, then a path starts from s' that satisfies φ₁ U φ₂, which can be extended to a path starting from s which also satisfies φ₁ U φ₂, which means that s also satisfies φ₁ EU φ₂.

Since we are looking for an optimal decision procedure, also conditions have to be formulated that define when a property is not satisfied in a state. A naive procedure would be to only make positive decisions until no decisions can be made any more, and then decide that all remaining states are FALSE. However, this would only allow negative decisions on complete models, not on partial models. Since we want to decide values of states as soon as possible, i.e. based on partial models, this is not sufficient.

Based on the rule for positive decisions, we can straightforwardly derive a first guess for a rule for negative decisions, namely:

\[
\text{if } s \not\models \phi_2 \land (s \not\models \phi_1 \lor \forall s' \in \text{succ}(s) . s' \in S_F) \text{ then } S_F := S_F \cup \{s\}
\]

If a state is to dissatisfy φ₁ EU φ₂, then at least it has to dissatisfy φ₂. If it also dissatisfies φ₁ then clearly s dissatisfies φ₁ EU φ₂. Also, if all successor states dissatisfy φ₁ EU φ₂ then s will also dissatisfy this property.

![Figure 4.8: Illustrating global decisions](image)

However, this does not yet result in a complete decision criterion for φ₁ EU φ₂, because cycles in the graph may cause situations in which states dissatisfy φ₁ EU φ₂, but cannot be decided to be part of S_F by the above rule. This happens in case there is a strongly connected set of states all satisfying φ₁ but not φ₂ and of which all successors that are not in this set do not satisfy φ₁ EU φ₂. These states all have at least one successor which is 'undecided'. Figure 4.8 illustrates this. In this case, we can obviously decide that all states in this set do not satisfy φ₁ EU φ₂, since paths that stay within the set will never pass through a state satisfying φ₂, while paths that leave the set will have a state not satisfying φ₁ EU φ₂ and therefore such paths do not satisfy φ₁ U φ₂.

This results in the following decision criteria for negative decisions for φ₁ EU φ₂:

- if \( \exists s \in S . s \not\models \phi_2 \land (s \not\models \phi_1 \lor \forall s' \in \text{succ}(s) . s' \in S_F) \) then \( S_F := S_F \cup \{s\} \)
- if \( \exists c \in \text{SCS} . \forall s \in c . (s \not\models \phi_2 \land \forall s' \in \text{succ}(s) \setminus c . s' \in S_F) \) then \( S_F := S_F \cup c \)

where SCS denotes the set of SCS's in the model. The first rule corresponds to our initial guess, while the second one is the new cycle-based rule. Note that these criteria are
Deciding fair CTL properties

formulated in a redundant way since some of the cases of the first criterion are already covered by the second one. So the first criterion can be written more compactly.

The set of decision rules corresponds to the following least-fixed point characterization of \([\neg \phi_1 \text{ EU } \phi_2]\) (without the redundancy).

\[
f(X) = \{s \in S \mid s \not\models \phi_2 \land s \not\models \phi_1\} \cup \bigcup \{c \in \text{SCS} \mid \forall s' \in c . s' \not\models \phi_2 \land \forall s \in \text{succ}(c) \setminus c . s \in X\}
\]

For \(\phi_1 \text{ AU } \phi_2\), similar reasoning applies. \([\phi_1 \text{ AU } \phi_2]\) is equal to the least fixed point of

\[
f(X) = \{s \in S \mid s \models \phi_2 \lor (s \models \phi_1 \land \forall s' \in \text{succ}(s) . s' \in X)\}
\]

while \([-\neg(\phi_1 \text{ AU } \phi_2)]\) is characterized by the least fixed point of

\[
f(X) = \{s \in S \mid s \not\models \phi_2 \land (s \not\models \phi_1 \lor \exists s' \in \text{succ}(s) . s' \in X)\} \cup \bigcup \{c \in \text{SCS} \mid \forall s' \in c . s' \not\models \phi_2\}
\]

In other words, a state can be decided to satisfy \(\phi_1 \text{ AU } \phi_2\) if it satisfies \(\phi_2\), or if it satisfies \(\phi_1\) and all its successors satisfy \(\phi_1 \text{ AU } \phi_2\). A state dissatisfies \(\phi_1 \text{ AU } \phi_2\) if it dissatisfies both \(\phi_1\) and \(\phi_2\), or it has a successor that dissatisfies \(\phi_1 \text{ AU } \phi_2\), or if it part of an SCS in which all states dissatisfy \(\phi_2\). The latter condition reflects the fact that given an SCS \(c\) of states dissatisfying \(\phi_2\), there can be paths that remain in \(c\) forever and therefore do not satisfy \(\phi_1 \text{ U } \phi_2\).

4.3.4 Recognizing fair paths

We now adapt these ideas to deal with the verification of fair CTL formulas, which means that we have to take into account the fact that path formulas range over fair paths only.

The approach taken for dealing with fairness is partly based on the work found in [39] and [62].

We first investigate how to recognize fair paths. We can partition the finite set of states of \(M\) into a set of maximal strongly connected components. Since paths are by definition infinite, any path will ultimately get caught in an SCC. Thus fair paths necessarily end up in SCC's on which all fairness predicates are at some point satisfied. Consequently, from a state \(s\) a fair path departs only in case an SCC on which all fairness predicates are satisfied, can be reached from it. The definition and theorem below formalize this idea.

**Definition 4.11 (fair subgraphs).** Let \(F\) be a set of fairness predicates and \(c\) be an SCS in a labelled transition system. Then \(c\) is an \(F\)-fair subgraph if

- \(\forall f \in F . \exists (\phi, r) \in c . r \models f\), and
- if \(c\) is a singleton set, then it has a self-looping transition

An SCC or SCS is \(F\)-fair if it is an \(F\)-fair subgraph. \(F\)-fair\(c\)\) is sometimes written to denote that \(c\) is \(F\)-fair.

The following theorem is a variation of lemma 4.1 found in [39].

**Theorem 4.7 (fair paths and fair components).** Let \(M\) be a labelled transition system and let \(F\) be a fairness constraint. Let \(s\) be state in \(M\). There exists a \(F\)-fair path in \(M\) starting from \(s\) if and only there is an \(F\)-fair SCS \(c\) in \(M\) and there is a state in \(s' \in c\) that can be reached from \(s\).
4.3.5 Deciding temporal operators in fair CTL

Using the characterization of fairness discussed above, we can adapt the decision criteria for CTL. For *EU* properties, the introduction of a fairness constraint will result in stronger conditions in the sense that less states will satisfy $\phi_1 EU \phi_2$, since fairness reduces the number of paths that satisfy a path formula $\phi_1 U \phi_2$.

In non-fair CTL, an *EU* property is satisfied in a state $s$ if a path exists that passes through a state $s'$ satisfying $\phi_2$ on which all states before $s'$ satisfy $\phi_1$. If fairness constraints apply this is not always the case. If $s'$ satisfies $\phi_2$, the path starting from $s$ will only satisfy $\phi_1 U \phi_2$ if it is a fair path. According to theorem 4.7 this means that the path has to end up in a fair SCS. In other words, there has to be a path to a state $s'$ that satisfies $\phi_2$ on which all states before $s'$ satisfy $\phi_1$ and from which a path departs to a fair SCS.

Using a mixture of non-fair CTL and fair CTL (using superscripts to distinguish the two) this can be written as $\phi_1 EU^{\text{non-fair}} (\phi_2 \land EF^{\text{fair}, \text{true}})$. $EF^{\text{fair}, \text{true}}$ expresses that there exists a fair path. We will use $E\text{true}$ as a shorthand for $EF^{\text{fair}, \text{true}}$. For the moment, suppose a decision procedure is available to decide $E\text{true} = EF^{\text{fair}, \text{true}}$. Then the decision criteria for $\phi_1 EU \phi_2$ can be written as follows (SCS denoting the set of strongly connected subgraphs):

- if $\exists s \in S \cdot (s \models E\text{true} \land s \models \phi_2) \lor (s \models \phi_1 \land \exists s' \in \text{succ}(s) \cdot s' \in S_T)$
  then $S_T := S_T \cup \{s\}$
- if $\exists s \in S \cdot s \not\models E\text{true} \lor (s \not\models \phi_2 \land s \not\models \phi_1)$
  then $S_F := S_F \cup \{s\}$
- if $\exists c \in \text{SCS} \cdot \forall s \in c \cdot s \not\models \phi_2 \land \forall s' \in \text{succ}(s) \setminus c \cdot s' \in S_F$
  then $S_F := S_F \cup c$

Until now, only (complete) labelled transition systems were considered. As described earlier, rules are needed to make decisions on partial models. This requires only a minor adaption. The only aspect in the decision rules that does not carry over to partial models is the occurrence of universal quantifications over successor states. These are only valid
in case all outgoing edges have been explored already. This can be ensured by requiring that in that case states are fully explored. For the above decision criteria this means that the third rule is changed to

- if \( \exists c \in SCS \land s \in c \land s \not\models \phi_2 \land s \not\in SU \land \forall s' \in suc(s) \setminus c \land s' \in S_F \)
  then \( S_F := S_F \cup c \)

while the other rules remain unchanged.

In the theorems below least fixed point characterizations for each of the temporal operators (and their negations) are defined, as well as for \( Etrue = true \land EU = true \). In the proof of the theorems we will use the following properties. Given a path \( \pi : s_0 \rightarrow s_1 \rightarrow \cdots \) and a state \( s_i \) on \( \pi \), the following holds:

\[
\psi \text{ is } F\text{-fair} \iff \psi(i) \text{ is } F\text{-fair} \quad (4.17)
\]

\[
\forall j \in \{0, \ldots, i - 1\} . \ s_i \models_{M} \phi_1 \land \psi(i) \models_{M} \phi_1 U \phi_2 \Rightarrow \psi \models_{M} \phi_1 U \phi_2 \quad (4.18)
\]

\[
\forall j \in \{0, \ldots, i - 1\} . \ s_i \not\models_{M} \phi_2 \land \psi(i) \not\models_{M} \phi_1 U \phi_2 \Rightarrow \psi \not\models_{M} \phi_1 U \phi_2 \quad (4.19)
\]

The first property follows directly from that fact that if a predicate is infinitely often satisfied along a path, then truncating a path by removing a finite set of states or appending a sequence of states will result in a path on which that predicate is still infinitely often satisfied. The other two properties follow directly from the semantics of CTL.

**Theorem 4.8 (characterizing \( \phi_1 \) \( EU \) \( \phi_2 \)).** Let \( M^* = (S^*, S_0^*, T^*, SU) \) be a partial model and \( F \) a fairness property. Then \((SCS\) denoting the set of strongly connected subgraphs in \( S^*\))

1. the set of states \( \{ s \in S^* \mid s \models_{M^*} \phi_1 EU \phi_2 \} \) is equivalent to the least fixed point of

\[
f_1(X) = \{ s \in S^* \mid s \models_{M^*} Etrue \land (s \models_{M^*} \phi_2 \land (s \models_{M^*} \phi_1 \land \exists s' \in suc(s) \cdot s' \in X)) \}\]

2. the set of states \( \{ s \in S^* \mid s \not\models_{M^*} \phi_1 EU \phi_2 \} \) is equivalent to the least fixed point of

\[
f_2(X) = \{ s \in S^* \mid s \not\models_{M^*} Etrue \lor (s \not\models_{M^*} \phi_2 \land s \not\models_{M^*} \phi_1) \lor (\exists c \in SCS . (s \in c \land \forall s' \in c . s' \not\models_{M^*} \phi_2 \land s' \not\in SU \land \forall s'' \in suc(s') \cdot c \land s'' \in X)) \}\]

**Proof.** First it is proved that \( f_1 \) and \( f_2 \) are both monotonic. For \( f_1 \), suppose \( X \subseteq Y \). Then it follows directly that \( Pred_T(X) \subseteq Pred_T(Y) \), which allows us to reason as follows (writing \( \models \) to denote \( \models_{M^*} \)).

\[
s \in f_1(X) \Rightarrow s \models_{M^*} Etrue \land (s \models_{M^*} \phi_2 \lor (s \models_{M^*} \phi_1 \land \exists s' \in suc(s) \cdot s' \in X)) \Rightarrow [Pred_T(X) \subseteq Pred_T(Y)]
\]

\[
s \models_{M^*} Etrue \land (s \models_{M^*} \phi_2 \lor (s \models_{M^*} \phi_1 \land \exists s' \in suc(s) \cdot s' \in Y)) \Rightarrow s \in f_1(Y)
\]

which means that \( f_1(X) \subseteq f_1(Y) \) and thus that \( f_1 \) is monotonic. The case for \( f_2 \) is similar:

\[
s \in f_2(X) \Rightarrow s \not\models_{M^*} Etrue \lor (s \not\models_{M^*} \phi_2 \land s \not\models_{M^*} \phi_1) \lor \exists c \in SCS . (s \in c \land \forall s' \in c . s' \not\models_{M^*} \phi_2 \land s' \not\in SU \land \forall s'' \in suc(s') \cdot c \land s'' \in X) \Rightarrow [Pred_T(X) \subseteq Pred_T(Y)]
\]

\[
s \not\models_{M^*} Etrue \lor (s \not\models_{M^*} \phi_2 \land s \not\models_{M^*} \phi_1) \lor \exists c \in SCS . (s \in c \land \forall s' \in c . s' \not\models_{M^*} \phi_2 \land s' \not\in SU \land \forall s'' \in suc(s') \cdot c \land s'' \in Y) \Rightarrow s \in f_2(Y)
\]
Because of theorem 4.6 and the above proved monotonicity, there exist least fixed points
\[ LFP_1 = \bigcup_{i \geq 0} f_1^i(0) \] and \[ LFP_2 = \bigcup_{i \geq 0} f_2^i(0) \] of \( f_1 \) and \( f_2 \), respectively. It remains to prove that
\[
\forall s \in S^* \ . \ (s \in LFP_1 \iff s \models_{M^*} \phi_1 EU \phi_2) \tag{4.20}
\]
\[
\forall s \in S^* \ . \ (s \in LFP_2 \iff s \models_{M^*} \phi_1 EU \phi_2) \tag{4.21}
\]
This is done by first proving four properties.

1. \([\forall M \models M^* \ . \forall s \in LFP_1 \ . \models_{M^*} \phi_1 EU \phi_2]\)

We prove that for any \( M \models M^* \), \( \forall i \geq 1 \ . \ s \in f_1^i(0) \Rightarrow s \models_{M^*} \phi_1 EU \phi_2 \), which gives the desired result, since \( LFP_1 = \bigcup_{i \geq 0} f_1^i(0) \). Consider a state \( s \) in a partial model \( M^* \) and an arbitrary model \( M \) such that \( M \models M^* \). We write \( \models \) to denote \( \models_{M^*} \). Note that (since \( M \models M^* \)) for \( \phi \in \{ \phi_1, \phi_2, Etrue \} \), \( s \models_{M^*} \phi \) implies \( s \models_{M} \phi \).

It follows from the definition of \( f_1 \) and the semantics of \( EU \) that if \( s \in f_1(0) \) then \( s \models Etrue \land s \models \phi_2 \), which means that \( s \models \phi_1 EU \phi_2 \). It remains to prove that for any \( n \geq 1 \), \( \forall s \in f_1^{n+1}(0) \ . \ s \models \phi_1 EU \phi_2 \) assuming that \( \forall s \in f_1^n(0) \ . \ s \models \phi_1 EU \phi_2 \) already holds. For an arbitrary \( s \in f_1^{n+1}(0) \), it follows from the definition of \( f_1 \) that
\[
(s \models Etrue \land s \models \phi_2) \lor (s \models \phi_1 \land \exists s' \in succ_T \cdot s'.(s' \in f_1^n(0))) \tag{4.22}
\]

Considering the left-hand side of the disjunction, \( s \models Etrue \land s \models \phi_2 \) means that there is a fair path from \( s \) of which the first state satisfies \( \phi_2 \). This means that \( s \models \phi_1 EU \phi_2 \). If the right-hand side of the disjunction is satisfied then from the assumption it follows that \( s \) has a successor \( s' \) that satisfies \( \phi_1 EU \phi_2 \). Then there must be a path \( \psi : s = s_0 \rightarrow s' = s_1 \rightarrow s_2 \rightarrow \cdots \in M^* \) (and thus \( M \)) such that \( \psi(s') \) is fair and satisfies \( \phi_1 U \phi_2 \). From properties 4.17 and 4.18 and the fact that \( s \models \phi_1 \) then follows that \( \psi \) is fair and satisfies \( \phi_1 U \phi_2 \), and thus that \( s \models \phi_1 EU \phi_2 \).

2. \([\forall M \models M^* \ . \forall s \in LFP_2 \ . \not\models_{M^*} \phi_1 EU \phi_2]\)

Consider a state \( s \) in a partial model \( M^* \) and an arbitrary model \( M \models M^* \). We write \( \not\models \) to denote \( \not\models_{M^*} \). For \( s \in f_2(0) \), \n
\[
s \not\models Etrue \lor (s \not\models \phi_2 \land s \not\models \phi_1) \lor \\
(\exists c \in SCS \ . \ (s \in c \land \forall s' \in c \ . \ s' \not\models \phi_2 \land s' \notin S_U \land \text{succ}_T \cdot (s') \setminus c = \emptyset))
\]

Considering the first two elements of the outer disjunction, if \( (s \not\models Etrue \lor (s \not\models \phi_2 \land s \not\models \phi_1)) \) then clearly \( s \not\models \phi_1 EU \phi_2 \). For the third element of the disjunction, any path in \( M \) from \( s \) is stuck within an SCS in which \( \phi_2 \) is never satisfied (since all regions in this SCS are fully explored). Therefore also in this case \( s \not\models \phi_1 EU \phi_2 \). Remains to prove that for any \( n \geq 1 \), \( \forall s \in f_2^{n+1}(0) \ . \ s \not\models \phi_1 EU \phi_2 \) assuming that \( \forall s \in f_2^n(0) \ . \ s \not\models \phi_1 EU \phi_2 \). Consider an arbitrary \( s \in f_2^{n+1}(0) \). From the definition of \( f_2 \) it follows that
\[
s \not\models Etrue \lor (s \not\models \phi_2 \land s \not\models \phi_1) \lor \\
(\exists c \in SCS \ . \ (s \in c \land \forall s' \in c \ . \ s' \not\models \phi_2 \land s' \notin S_U \land \forall s'' \in \text{succ}_T \cdot (s') \setminus c . s'' \in f_2^n(0))) \tag{4.23}
\]

If \( s \not\models Etrue \lor (s \not\models \phi_2 \land s \not\models \phi_1) \) then \( s \not\models \phi_1 EU \phi_2 \). For the third element of the disjunction, consider an \( s \) in some SCS \( c \) as defined in equation 4.23. We assume that \( s \models Etrue \) because otherwise from the left-hand side follows that \( s \not\models \phi_1 EU \phi_2 \).

Consider a fair path \( \psi : s = s_0 \rightarrow s_1 \rightarrow \cdots \in M \). If \( \psi \) remains in \( c \) forever, then all states on \( \psi \) dissatisfy \( \phi_2 \), which means that \( \psi \not\models \phi_1 U \phi_2 \). Alternatively, if \( \psi \) leaves \( c \)
at some point, it has a state $s_k$ which is the first state on $\psi$ that is not in $c$. Because all states on $c$ dissatisfy $\phi_2$ it follows that $\forall k \in \{0, \ldots, k-1\} . s_k \not\models \phi_2$. Also because all states in $c$ are not in $S_U$, $s_{k-1}$ was fully explored in $M^*$ and thus $s_k \in S^*$. Then it follows from equation 4.23 that $s_k \in f_2''(\emptyset)$, and then from the assumption it follows that $s_k \not\models \phi_1 EU \phi_2$. Together with the fact that — because of property 4.17 — $\psi(k)$ is fair, we can conclude that $\psi(k) \not\models \phi_1 U \phi_2$. Then from property 4.19 and the fact that $\forall l \in \{0, \ldots, k-1\} . s_l \not\models \phi_2$ it follows that $\psi \not\models \phi_1 U \phi_2$. Thus each fair path from $s$ dissatisfies $\phi_1 EU \phi_2$. Summarizing, $s \not\models \phi_1 EU \phi_2$, which completes this case.

3. \( \exists M \models M^* . \forall s \in \{LFP_1 \cup LFP_2\} . s \not\models_M \phi_1 EU \phi_2 \)

Let $B = S^* \setminus \{LFP_1 \cup LFP_2\}$. Because $f_1(\emptyset) \subseteq LFP_1$ and $f_2(\emptyset) \subseteq LFP_2$, it follows that

\[
\forall s \in B . s \models_M \text{Etrue} \land s \not\models_M \phi_2 \land s \models_M \phi_1 \tag{4.24}
\]

Also from the fact that $f_1(LFP_1) \cap B = \emptyset$ and the above equation, it follows that $\forall s \in \text{succ}_{CT}(B) . s \not\in LFP_1$. Since $(S^* \setminus B) \setminus LFP_1 = LFP_2$ it follows that

\[
\forall s \in \text{succ}_{CT}(B) \setminus B . s \in LFP_2 \tag{4.25}
\]

Now consider an arbitrary state $s \in B$. We first prove that from $s$, a state $s' \in S_U$ can be reached (through $T^*$) by only passing through states in $B$. This is done by contradiction, assuming that from $s$ no state in $S_U$ can be reached through only states in $B$. Let $B'$ be the set of states in $B$ that can be reached from $s$. Let $P$ be the partitioning of $B'$ into strongly connected components (with respect to edges in $B'$). Then from the nature of strongly connected components it follows that there must be a $c \in P$ such that $(\text{succ}_{CT}(c) \setminus c) \cap B' = \emptyset$. Then

\[
\forall s' \in c . s' \not\models_M \phi_2 \land s' \notin S_U \land \forall s'' \in \text{succ}_{CT}(s') \setminus c . s'' \in LFP_2 \tag{4.26}
\]

which is explained as follows: $s' \not\models_M \phi_2$ because of equation 4.24; $s' \notin S_U$ follows from the definition of $B'$ and the assumption; finally, $\forall s'' \in \text{succ}_{CT}(s') \setminus c . s'' \in LFP_2$ follows from the definition of $c$ and equation 4.25. Equation 4.26 implies that $c \subseteq f_2(LFP_2)$ which is not consistent with $c \subseteq B$. This contradiction proves our intermediate result.

This means that for each $s \in B$, there exists a sequence $s = s_0 \rightarrow_T . s_1 \rightarrow_T . \cdots . s_i$ such that $s_i \in S_U$ and $\forall j \in \{0, \ldots, i\} . s_j \in B$. Now we choose $M = (S, S_0, T) \models M^*$ such that $\forall s \in S_U . \exists s' \in \text{succ}_{CT}(s) \setminus S^* . s' \models_M \phi_1 EU \phi_2$ — i.e. each state that was not fully explored in $M^*$ is given a successor in $M$ that satisfies $\phi_1 EU \phi_2$. We have to prove that $\forall s \in B . s \not\models_M \phi_1 EU \phi_2$. We know that $s_1$ from the above sequence has a successor that satisfies $\phi_1 EU \phi_2$. As a consequence, there exists a path $s = s_0 \rightarrow_T . s_1 \rightarrow_T . \cdots . s_{i+1} \rightarrow_T . \cdots$ from $s$ such that $\psi(s_{i+1})$ is fair and $\psi(s_{i+1}) \models_M \phi_1 U \phi_2$. From equations 4.17 and 4.18 and the fact that $\forall j \in \{0, \ldots, i\} . s_j \models_M \phi_1$ it then follows that $\psi$ is fair and satisfies $\phi_1 U \phi_2$. Therefore $s \models_M \phi_1 EU \phi_2$.

4. \( \exists M \models M^* . \forall s \in \{LFP_1 \cup LFP_2\} . s \not\models_M \phi_1 EU \phi_2 \)

Let $B$ be as defined under property 3. Choose $M = (S, S_0, T) \models M^*$ such that $\forall s \in S_U . \forall s' \in \text{succ}_{CT}(s) \setminus \text{succ}_{CT}(s) . s' \not\models_M \phi_1 EU \phi_2$. It is to be proved that $\forall s \in B . s \not\models_M \phi_1 EU \phi_2$. First it is proved that

\[
\forall s \in \text{succ}_{CT}(B) \setminus B . s \not\models_M \phi_1 EU \phi_2 \tag{4.27}
\]

Consider an $s \in \text{succ}_{CT}(B) \setminus B$. If $s \in \text{succ}_{CT}(B)$, then by equation 4.25 $s \in LFP_2$ and by property 2, $s \not\models_M \phi_1 EU \phi_2$. Alternatively, if $s \not\in \text{succ}_{CT}(B)$, then $s$ must have a
predecessor in $S_U$ and then from the definition of $M$ it follows that $s \not\models_M^F \phi_1 EU \phi_2$. This proves equation 4.27.

Now consider an arbitrary fair path $\psi : s = s_0 \rightarrow s_1 \rightarrow \cdots \in M$ from a state $s \in B$. If $\psi$ remains in $B$ forever, then from 4.24 it follows that all states on $\psi$ dissatisfy $\phi_2$ and thus that $\psi \not\models_M^F \phi_1 U \phi_2$. Alternatively, if $\psi$ leaves $B$ at some point, let $s_k$ denote the first state not in $B$. Then from equation 4.27 it follows that $s_k \not\models_M^F \phi_1 EU \phi_2$, which means that (using equation 4.17) $\psi(s_k) \not\models_M^F \phi_1 U \phi_2$. From property 4.19 and the fact that $\forall 0 \leq i < k . s_i \not\models_M^F \phi_2$ (equation 4.24), it follows that $\psi \not\models_M^F \phi_1 U \phi_2$. It then follows that each fair path from $s$ dissatisfies $\phi_1 U \phi_2$ and thus that $s \not\models_M^F \phi_1 EU \phi_2$.

Together, these four properties allow us to deduce equations 4.20 and 4.21. For equation 4.20, the $\Rightarrow$ direction follows immediately from property 1: $\forall s \in LFP_1 . \forall M \models M^*. s \models_M^F \phi_1 EU \phi_2$ $\Rightarrow$ $s \models_M^F \phi_1 EU \phi_2$. As for the $\Leftarrow$ direction, for each $s \in S^*$:

$s \not\models_M^F \phi_1 EU \phi_2$ $\Rightarrow$ $s \not\models_LFP_1 \lor s \in S^* \setminus (LFP_1 \cup LFP_2)$ $\Rightarrow$ [properties 2 and 4]
$(\forall M \models M^*. s \not\models_M^F \phi_1 EU \phi_2) \lor (\exists M \models M^*. s \not\models_M^F \phi_1 EU \phi_2)$ $\Rightarrow$ [definition of $\models_{M^*}$] $s \not\models_M^F \phi_1 EU \phi_2$

Equation 4.21 follows in a similar way. $\Box$

We saw that for $EU$ properties the impact of fairness was quite limited. It only introduces an additional constraint to ensure that fair paths exist. For $AU$ properties, the introduction of fairness has more consequences.

For $AU$ properties, the $Etrue$ property plays a dual role compared to $EU$ properties. If $Etrue$ does not hold for some state $s$, then it follows immediately that $s$ satisfies $\phi_1 AU \phi_2$. This leads to a first attempt at decision rules for $\phi_1 AU \phi_2$:

- if $\exists s \in S . s \not\models Etrue \lor s \models \phi_2 \lor (s \models \phi_1 \land \forall s' \in succ(s). s' \in S_T)$
  then $S_T := S_T \cup \{s\}$
- if $\exists s \in S . s \models Etrue \land s \not\models \phi_2 \land (s \not\models \phi_1 \land \exists s' \in succ(s). s' \in S_F)$
  then $S_F := S_F \cup \{s\}$
- if $\exists c \in SCS \land \forall s \in c . s \not\models \phi_2$
  then $S_T := S_T \cup c$

However, the third rule is clearly not valid. It would only be correct for fair SCS’s, because only for those, fair paths can remain in that SCS. Because of theorem 4.7, any fair path must eventually leave a non-fair SCS. So in the third rule, we would have to add the constraint that $c$ is fair. This leaves us with non-fair SCS’s, in particular, those in which all states satisfy $\phi_1$ but dissatisfy $\phi_2$, and for which all successors outside the SCS satisfy $\phi_1 AU \phi_2$. This case would still be uncovered. All fair paths from states in such an SCS would — according to theorem 4.7 — have to leave the SCS, and thus encounter a state that satisfies $\phi_1 AU \phi_2$. From that we can infer that all states in the non-fair SCS must also satisfy $\phi_1 AU \phi_2$. For the same reasons, the first rule is not completely correct because $\{s\}$ can be a fair SCS (because of a self-looping edge). If in that case $s$ does not satisfy $\phi_2$, then it follows that $s$ does not satisfy $\phi_1 AU \phi_2$.

Figure 4.9 illustrates the global decision rules for $AU$. In the situation sketched there, the two states that form a cycle are decided to TRUE or FALSE, depending on the subgraph being non-fair or fair, respectively.

This leads to the following rules:

- if $\exists s \in S . (s \not\models Etrue \lor s \models \phi_2)$
  then $S_T := S_T \cup \{s\}$
Theorem 4.9 (characterizing $\phi_1 AU \phi_2$). Let $M = \langle S^*, S_0^*, T^*, S_U \rangle$ be a partial model and $F$ a fairness property. Then

1. the set of states $\{s \in S^* \mid s \not\in_{M^*} F, \phi_1 AU \phi_2 \}$ is equivalent to the least fixed point of

   \[
   f_1(X) = \{ s \in S^* \mid s \not\in_{M^*} F, \text{Etrue} \vee s \not\in_{M^*} F, \phi_2 \vee \\
   \exists c \in SCS . (s \in c \land \neg F\text{-fair}(c) \land \forall s' \in c . (s' \not\in F, \phi_1 \land s' \notin S_U \land \\
   \forall s'' \in \text{succ}_{T^*}(s') \land c . s'' \in X) \}
   \]

2. the set of states $\{s \in S^* \mid s \not\in_{M^*} F, \phi_1 AU \phi_2 \}$ is equivalent to the least fixed point of

   \[
   f_2(X) = \{ s \in S^* \mid (s \not\in_{M^*} F, \text{Etrue} \land s \not\in_{M^*} F, \phi_2 \land \\
   (s \not\in_{M^*} F, \phi_1 \land \exists s' \in \text{succ}_{T^*}(s) . s' \in X) \lor \\
   \exists c \in SCS . (s \in c \land F\text{-fair}(c) \land \forall s' \in c . s' \not\in_{M^*} F, \phi_2) \}
   \]

Proof. Monotonicity can be proved in the same manner as for $\phi_1 EU \phi_2$, which is not shown here. Again there exist least fixed points $LFP_1 = \bigcup_{i \geq 0} f_1(\emptyset)$ and $LFP_2 = \bigcup_{i \geq 0} f_2(\emptyset)$, of $f_1$ and $f_2$, respectively. It remains to prove that

\[
\forall s \in S^* . (s \in LFP_1 \iff s \not\in_{M^*} F, \phi_1 AU \phi_2) \quad (4.28)
\]

\[
\forall s \in S^* . (s \in LFP_2 \iff s \not\in_{M^*} F, \phi_1 AU \phi_2) \quad (4.29)
\]

which is again done in four steps.

1. $\forall M \models M^* \Rightarrow \forall s \in LFP_1 . s \not\in_{\hat{M}} F, \phi_1 AU \phi_2$

   We prove that for any $M \models M^*$, $\forall i \geq 1 . s \in f_i(\emptyset) \Rightarrow s \models F, \phi_1 AU \phi_2$ (writing $\models$ to denote $\models_{\hat{M}}$). Consider an arbitrary model $M \models M^*$ and an arbitrary state $s \in S^*$. If $s \in f_1(\emptyset)$ then

   \[
   s \not\in \text{Etrue} \vee s \models \phi_2 \vee \exists c \in SCS . (s \in c \land \neg F\text{-fair}(c) \land \\
   \forall s' \in c . (s' \models \phi_1 \land s' \notin S_U \land \text{succ}_{T^*}(s') \land c = \emptyset))
   \]
For the first two elements of the disjunction immediately follows that $s \models \phi_1 AU \phi_2$. For the third element, all paths from $s$ are stuck in a non-fair component, which means that there are no fair paths from $s$, and thus $s \not\models \phi_1 AU \phi_2$. Now assume that

$$\forall s \in f_1^n(\emptyset) . s \models \phi_1 AU \phi_2.$$  

Then $s \in f_1^{n+1}(\emptyset)$ means that

$$s \not\models Etrue \lor s \models \phi_2 \lor \exists c \in SCS . (s \in c \land \neg F\text{-}fair(c) \land \forall s' \in c . (s' \models \phi_1 \land s' \notin S_U \land \forall s'' \in succ_T(s') . s'' \in f_1^n(\emptyset))).$$  

(4.30)

For the first two elements of the disjunction it follows immediately that $s \models \phi_1 AU \phi_2$. Now consider a state $s$ that satisfies the third element of the disjunction. We can assume that at least one fair path from $s$ departs, since otherwise $s \not\models Etrue$ (which is covered by the left-hand side). Consider such a fair path $\psi : s = s_0 \rightarrow s_1 \rightarrow \cdots$ in $M$. Because $c$ is non-fair, according to theorem 4.7, $\psi$ must leave $c$ at some point. Let $s_i$ be the first state on $\psi$ that is not in $c$. Then $s_i \in S^*$ because $s_{i-1}$ — like each state in $c$ — is not in $S_U$ and thus is fully explored in $M^*$. Then from the definition of $c$ in equation 4.30 it follows that $s_i \in F_1^0(\emptyset)$ and thus — using our assumption — $s_i \models \phi_1 AU \phi_2$. This means (using equation 4.17) that $\psi(s_i) \models \phi_1 U \phi_2$. From equation 4.18 and the fact that $\forall j \in \{0, \ldots, i-1\} . s_j \models \phi_1$ (since all state in $c$ satisfy $\phi_1$) it follows that $\psi \models \phi_1 U \phi_2$. Since this reasoning is true for any fair path from $s$, it follows that $s \models \phi_1 AU \phi_2$.

2.  $\forall M \models M^* \land \forall s \in LFP_2 . s \not\models_{M^*} Etrue \land \phi_1 AU \phi_2$

Consider an arbitrary model $M \models M^*$ and an arbitrary $s \in S^*$. If $s \in f_2(\emptyset)$ then

$$(s \models Etrue \land s \not\models \phi_2 \land s \not\models \phi_1) \lor (\exists c \in SCS . s \in c \land F\text{-}fair(c) \land \forall s' \in c . s' \not\models \phi_2)$$

For the left-hand side of the disjunction it follows immediately from the semantics of $AU$ that $s \not\models \phi_1 AU \phi_2$. The right-hand side implies that there is a fair path on which $\phi_2$ is never satisfied, which means that $s \not\models \phi_1 AU \phi_2$. Now assume that $\forall s \in f_2^n(\emptyset) . s \not\models \phi_1 AU \phi_2$. If $s \in f_2^{n+1}(\emptyset)$, it follows from the definition of $f_2$ that (using that $\forall s \in f_2^n(\emptyset) . s \not\models \phi_1 AU \phi_2$)

$$(s \models Etrue \land s \not\models \phi_2 \land s \not\models \phi_1) \lor (s \models Etrue \land s \not\models \phi_2 \land \exists s' \in succ_T . s \models \phi_1 AU \phi_2) \lor (\exists s \in SCS . (s \in c \land F\text{-}fair(c) \land \forall s' \in c . s' \not\models \phi_2)$$

For the first element of the disjunction it follows immediately that $s \not\models \phi_1 AU \phi_2$. For the second element of the disjunction it follows that there exists a path $\psi : s = s_0 \rightarrow s_1 \rightarrow \cdots$ such that $\psi(1)$ is $F\text{-}fair$ and $\psi(1) \not\models \phi_1 U \phi_2$. Using also the fact that $s \not\models \phi_2$, it follows from equations 4.17 and 4.18 that $\psi$ is $F\text{-}fair$ and $\psi \not\models \phi_1 U \phi_2$, and thus that $s \not\models \phi_1 AU \phi_2$. Finally, consider the third element of the disjunction. According to theorem 4.7, if $s$ is an element of a fair SCS $c$ then there is a fair path $\psi$ from $s$ that stays in $c$. Since all states in $c$ dissatisfy $\phi_2$, it follows that $\psi \not\models \phi_1 U \phi_2$. This means that $s \not\models \phi_1 AU \phi_2$.

3.  $\exists M \models M^* . \forall s \in S^* \land (LFP_1 \cup LFP_2) . s \models_{M^*} Etrue \land \phi_1 \land s \not\models_{M^*} F\text{-}true \land \forall c \in SCS_B . \neg F\text{-}fair(c))$

Let $B = S^* \setminus (LFP_1 \cup LFP_2)$. From $f_1(\emptyset) \subseteq LFP_1$ and $f_2(\emptyset) \subseteq LFP_2$ and the definition of $B$ it follows that

$$\forall s \in B . (s \models_{M^*} Etrue \land s \models_{M^*} \phi_1 \land s \not\models_{M^*} F\text{-}true \land \forall c \in SCS_B . \neg F\text{-}fair(c))$$

(4.31)
where $SCS_B$ denotes the set of SCS’s in $B$. From this equation and the definitions of $f_2$ and $B$ it follows that for each $s \in B$, $(\exists s' \in succ_T(s) . s' \in LFP_2) \Rightarrow s \in LFP_2$ and thus because $B \cap LFP_2 = \emptyset$ that $succ_T(B) \cap LFP_2 = \emptyset$. This means that

$$succ_T(B) \setminus B \subseteq LFP_1 \tag{4.32}$$

Choose $M = \langle S, S_0, T \rangle \sqcup M^*$ such that $\forall s \in S_U . \forall s' \in succ_T(s) \setminus succ_T(s) . s' \models_{M^*} \phi_1 AU \phi_2$. We have to prove that $\forall s \in B . s \models_{M^*} \phi_1 AU \phi_2$. Consider an arbitrary state $s \in B$ and fair path $\psi : s = s_0 \rightarrow s_1 \rightarrow \cdots \in M$ from $s$. $\psi$ cannot remain in $B$ forever because in that case (according to theorem 4.7) it would have to end up in a fair SCS in $B$, which is not possible because according to equation 4.31 there are no fair SCS’s in $B$. Therefore, let $s_k$ be the first state on $\psi$ that is not in $B$. If $s_k \in S^*$, then according to equation 4.32, $s_k \in LFP_1$, and thus according to property 1, $s_k \models_{M^*} \phi_1 AU \phi_2$. Alternatively, if $s_k \notin S^*$, then $s_k$ must have been a not fully explored state in $B$, in which case from the definition of $M$ it also follows that $s_k \models_{M^*} \phi_1 AU \phi_2$. Now $s_k \models_{M^*} \phi_1 AU \phi_2$ means that $\psi(s_k) \models_{M} \phi_1 U \phi_2$. Then, from equation 4.17 and the fact that (according to equation 4.31) $\forall i \in \{0, \ldots, k - 1\} . s_i \models_{M} \phi_1$, it follows that $\psi \models_{M} \phi_1 U \phi_2$. Since this reasoning applies to any path from $s$, it follows that $s \models_{M^*} \phi_1 AU \phi_2$.

4. $\exists M \sqcup M^* . \forall s \in S^* \setminus (LFP_1 \cup LFP_2) . s \not\models_{M^*} \phi_1 AU \phi_2$

Let $B$ be defined as in property 3. Consider an arbitrary state $s \in B$. First it is proved that from $s$ a state $s' \in S_U$ can be reached (through $T^*$) by only passing through states in $B$. This is done by contradiction, assuming that no state in $S_U$ can be reached through only states in $B$. Let $B'$ be the set of states in $B$ that can be reached from $s$. Let $P$ be the partitioning of $B$ into strongly connected components (with respect to edges in $B'$). Then from the nature of strongly connected components it follows that there must be a $c \in P$ such that $(succ_T(c) \setminus c) \cap B' = \emptyset$. Then

$$\neg F \text{-fair}(c) \land \forall s \in c . s \models_{M^*} \phi_1 \land s \notin S_U \land \forall s' \in succ_T(s) \setminus c . s' \in LFP_1 \tag{4.33}$$

which is explained as follows: $\neg F \text{-fair}(c)$ because of equation 4.31; $s \models_{M^*} \phi_1$ also follows from equation 4.31; $s \notin S_U$ follows from the definition of $B'$ and the assumption; and $\forall s' \in succ_T(s) \setminus c . s' \in LFP_1$ follows from the definition of $c$ and equation 4.32. Equation 4.33 implies that $c \subseteq f_1(LFP_1)$ which contradicts $c \subseteq B$.

This means we can conclude that there exists a sequence $s = s_0 \rightarrow_T . s_1 \rightarrow_T . \cdots . s_i$ such that $s_i \in S_U$ and $\forall j \in \{0, \ldots, i\} . s_j \in B$.

Now choose some $M = \langle S, S_0, T \rangle \sqcup M^*$ such that $\forall s \in S_U . \exists s' \in succ_T(s) . s' \not\models_{M^*} \phi_1 AU \phi_2$. It is to be proved that $\forall s \in B . s \not\models_{M^*} \phi_1 AU \phi_2$.

From the definition of $M$ it then follows that $s_i$ has a successor $s_{i+1}$ such that $s_{i+1} \not\models_{M^*} \phi_1 AU \phi_2$. This means that there exists a path $\psi : s = s_0 \rightarrow_T . s_1 \rightarrow_T . \cdots$ in $M$ such that $\psi(s_{i+1})$ is fair and $\psi(s_{i+1}) \not\models_{M^*} \phi_1 U \phi_2$. Together with the fact that $\forall j \in \{0, \ldots, i\} . s_j \not\models_{M^*} \phi_2$ (equation 4.31), it follows using equations 4.17 and 4.19 that $s \not\models_{M^*} \phi_1 AU \phi_2$.

Like in the previous theorem, the combination of the four properties proved above gives equations 4.28 and 4.29.

This leaves us with finding a decision procedure for $E_{true}$ — in other words for deciding whether or not a state has a departing fair path. This follows readily from theorem 4.7. In least-fixed-point formulation, a state $s$ has at least one departing fair path, if it is part
Chapter 4 Evaluating TCTL properties

of a fair SCS, or if one of its successors has a departing fair path. Dually, a state has no departing fair paths if it is part of a non-fair SCS for which all successors outside the SCS have no departing fair paths.

Theorem 4.10 (characterizing $E_{true}$). Let $M = (S^*, S_0^*, T^*, S_U)$ be a partial model and $F$ a fairness property. Then

1. the set of states $\{s \in S^* | s \models E_{true}^M \}$ is equivalent to the least fixed point of
   
   $$f_1(X) = \{s \in S^* | (\exists s' \in succ_T(s) . s' \in X) \lor (\exists c \in SCS . s \in c \land F\text{-fair}(c))\}$$

2. the set of states $\{s \in S^* | s \nmodels E_{true}^M \}$ is equivalent to the least fixed point of
   
   $$f_2(X) = \{s \in S | \exists c \in SCS . s \in c \land \neg F\text{-fair}(c) \land \forall s' \in c . (s' \notin S_U \land \forall s'' \in succ_T(s') \setminus c . s'' \in X)\}$$

The proof is similar to those for $\phi_1 AU \phi_2$ and $\phi_1 EU \phi_2$, and therefore not shown here.

4.3.6 Summarizing

In section 4.3.5, decision criteria were defined for the temporal operators of fair CTL by means of least fixed point characterizations. The rules for the other operators can be directly derived from the semantics of fair CTL. Below are the rules for $EX$.

if $\exists s' \in succ(s) . s' \models F^M . \phi$ then $S_T := S_T \cup \{s\}$
if $\forall s' \in succ(s) . s' \nmodels F^M . \phi$ then $S_N := S_T \cup \{s\}$

while the rules for $\phi_1 \lor \phi_2$ are equally straightforward:

if $s \models F^M . \phi_1 \lor s \models F^M . \phi_2$ then $S_T := S_T \cup \{s\}$
if $s \nmodels F^M . \phi_1 \land s \nmodels F^M . \phi_2$ then $S_N := S_T \cup \{s\}$

As a result, a complete hierarchical decision procedure can be constructed. For each subproperty of the property to be verified, decision criteria are available. Note that in principle, evaluation can be done in any order. To evaluate a subproperty $\phi'$ of a property $\phi$, it is not required that all subproperties of $\phi'$ have already been decided. The total set of decision criteria is optimal, which means that if for each subproperty, no additional decisions can be made, then the set of states for which $\phi$ was decided, is the largest set of states that can be decided given the currently available information (i.e. the explored portion of the state space).

An actual evaluation approach based on the criteria defined here would probably not maintain separate evaluation state for each subproperty. For several operators, decision procedures can be combined. For example, nested boolean operators can be combined. However, as will be shown in chapter 6, at least for each temporal operator, a separate evaluation state is required.

4.4 Discussion

This chapter described a general approach to evaluating fair TCTL properties using time-abstracting bisimulation abstractions. As was shown this problem has two subproblems: The first was the translation of a TCTL verification problem on a timed transition system
to a CTL verification problem on an abstraction of this timed transition system. The second problem was to deal with the CTL verification problem. The approaches — both the TCTL-CTL transformation as well as the CTL decision criteria — described in this section are of a general nature, and can also be applied in other contexts than partition refinement.

For the first problem, it turned out that the verification problem on an abstract model based on time-abstracting bisimulation can be defined using CTL, using an intermediate logic, that we called D-CTL. To be able to do so, XTG models under investigation had to be extended with some additional structure. Although this does not affect the validity of the verification, it does make the XTG more complex, that is, it leads to more complex state spaces. However, this cannot be avoided since without the extensions, XTG models lead to abstract models that are not suitable to evaluate TCTL properties.

The second subproblem was CTL verification on finite state spaces — an issue that has been addressed in literature already (see for example [35, 19, 39, 38]). However, the goal here was to approach this problem as much as possible without committing to any specific algorithm. More precisely, our goal was to prescribe general decision rules that are optimal, in the sense that they specify exactly when decisions can be made when evaluating properties on-the-fly.

While for many CTL verification approaches this is not so relevant, it is interesting in the context of infinite-state symbolic approaches like our partition refinement approach. For symbolic model checkers, the representation and manipulation of symbolic state spaces has major impact on its resource usage and thus their usability. Compared to explicit model checking approaches, more complex decision making procedures could then be justified. It could very well be that certain procedures would be too expensive when checking fair CTL properties on an explicit state space, while for a symbolic approach it would be worthwhile, because it could avoid unnecessary exploration of parts of the state space. Furthermore, for partition refinement techniques there is much room for choosing a particular evaluation approach, in contrast with reachability-based techniques, where the evaluation approach is tightly bound to the exploration approach.

For this reason, we systematically approached the problem of evaluating fair CTL properties, introducing the notion of partial models to be able to reason about partially explored state spaces and optimal decision procedures. Note that although some decision rule may be optimal from a theoretic point of view, it may be impractical in practice, for example because checking its conditions is computationally too expensive. The actual instantiation of the decision rules is dealt with in chapter 6.

Note that in [31] also partial state spaces are modelled. There, the intention is to deal with incomplete state spaces that occur as a result of abstractions. The partial models are more involved than the ones discussed here. Our approach is a rather straightforward adaptation of labelled transition systems that allows us to reason about on-the-fly property evaluation.

In [61], also a transformation from TCTL to CTL is reported. There, a TCTL verification problem on timed structures (timed transition systems) induced by timed automata is reduced to a CTL verification problem on Kripke structures (labelled transition systems) induced by detailed region graph equivalence. Essentially, here a detailed region graph is built according to [2]. The approach is not implemented, but detailed region graphs are inherently much larger than models based on time-abstracting bisimulation. Also note that the expressiveness of our system and property specification language would not allow such a transformation.
In [104] a transformation from TCTL to CTL is defined which is more elegant and efficient than ours. There a different version of TCTL is used, but their approach would also work for our version of TCTL. The idea is in fact quite simple. Rather than introducing a property graph that deals with property updates, a smarter transformation from TCTL to CTL is used. As an example, consider a property $AG(c=0. AF(p \land c < 10))$. Applying the idea of [106] it is transformed to the CTL property $AG(c = 0 \Rightarrow AF(p \land c < 10))$. This property can be verified on a system in which the property variables are added, but no extension with a property variable automaton is needed.

Several aspects of our approach to dealing with fairness are based on existing practice. However, as was discussed above, evaluation of fair CTL properties was dealt with in a very general manner. Related to that is the way divergence is checked in terms of fairness. Our approach for dealing with divergence is more involved compared to commonly used approaches, because of the expressiveness of the XTG language. For (original) timed automata, only simple clock constraints and updates are allowed, which enables a simpler approach that does not require the introduction of additional structure [62].

The automata-theoretic approach to branching time model checking [19] has also been extended to deal with real-time systems. Automata-theoretic approaches to branching time model checking use alternating tree automata to deal with branching-time logics. In [62], this approach is extended to deal with TCTL verification. This is again based on a reduction to CTL verification on detailed region graphs. In a sense there is some distant similarity to the automata-theoretic approach and the approach described here — although the former tends to be more formal. Branching-time approaches translate properties to automata, which are combined with the automaton of the system. Our approach to dealing with property variables, also constructs an automaton which is combined with the original system. However, in our case, this is restricted to what is minimally needed to be able to evaluate our property, to avoid useless complexity of the resulting state space.
Chapter 5

Symbolic state space representation

A key aspect of a symbolic model checking technique is the representation of the symbolic state space. This chapter discusses a state space representation for partition refinement approaches to symbolic model checking of real-time systems.

5.1 Introduction

The central idea behind symbolic model checking is that state space exploration operates on (often infinite) sets of states instead of individual states. It is facilitated by the ability to store and manipulate these sets symbolically — thus without explicitly representing the states in these sets. It is safe to say that efficient symbolic state space representation is a main enabler of symbolic model checking. This is illustrated by the fact that the breakthrough of symbolic model checking came with the discovery of an efficient state space representation, namely binary decision diagrams.

The capability of a model checker to handle complex verification problems strongly depends on the efficiency of the symbolic state space representation. For symbolic model checking, efficiency in terms of time as well as storage has proven to be important. In some cases the inability to solve a verification problem stems from the lack of storage capability, while in other cases time will be the bottleneck.

To be useful, a symbolic state space representation must have two characteristics. First, it should efficiently store symbolic states, and second, it should allow efficient manipulation of symbolic states. These two characteristics of symbolic state space approach have clear links to the two dimensions of efficiency. The storage performance is determined by the space efficiency of the representation itself, while the time performance is determined by the efficiency of the operations. The efficiency of representation and operations depends on exactly what has to be represented (i.e. the form of symbolic states) and which operations are needed (i.e. the manipulations on symbolic states).

The form of the symbolic states is induced by the type of verification problem that is to be handled. This usually follows directly from the type of system models that are to be verified. If for example, in a timed automaton formalism arbitrary linear constraints are allowed in the guards of edges, then the state space representation will have to deal with symbolic states that are (at least partly) defined by linear constraints.
Chapter 5  Symbolic state space representation

The types of required manipulations on symbolic states follows from the exploration technique that is used. For example, for a partition refinement exploration approach splitting and propagation operations are be needed, while for a forward reachability approach different operations are needed (like intersection and union). The realization of the operations is again dependent of the form of verification problem. In general, more complex constraints and updates in system models and properties will result in more complex operations.

The classic symbolic model checking approach introduced by Clarke and coworkers [35] is targeted at finite state systems. For finite state systems, Boolean formulas can be used to represent sets of states. The binary decision diagrams (BDD’s) technique [33] has proven to be an efficient representation for the resulting boolean formulas. The exploration technique is based on fixed point computation [39], of which the required operations can be implemented conveniently using BDD’s.

The techniques discussed in this thesis fall into another major category of application of symbolic model checking. These approaches deal with infinite state spaces induced by system models like timed automata or one of its variations. Here, clocks and other variables with infinite domains induce infinite states spaces, and thus introduce the need for symbolic approaches. This chapter focuses on defining a state space representation approach for such state spaces that fits well to the partition refinement approach to state space exploration. It is essential to note that the exploration and representation aspects should not be viewed in isolation. A representation approach that is optimal for some exploration algorithm is not necessarily the best approach for another exploration algorithm, because different exploration algorithms will require different operations on the symbolic state space.

For our type of system models, states are typically defined by a control component and a data component. We will focus on symbolic representation of the data component. Symbolic states — referred to as regions — can then be represented as tuples \( (l, Z) \), where \( l \) is control component (the location), and \( Z \) is a representation for a set of valuations for the system’s variables, called a zone. Such a region represents the set of states \( (l, \rho) \) for which \( \rho \) is a valuation that is a member of the set represented by \( Z \). Thus, we are looking for a representation for sets of variable valuations. In principle it is possible to also involve the control component in the symbolic representation (see for example [86, 63]). For the partition refinement approach this does not seem useful, because state sets will almost always be split according to their control location. The reason for this is that two states having a different control location will most often not agree on the set of edges that are enabled.

Zones are characterized by constraints. For real-time model checking, the most important type of constraint are constraints on clocks, or more general, constraints on real variables. For the timed automaton formalisms traditionally used in dense real-time model checking, the only variables are clocks. In that case, zones are completely characterized by constraints on clocks. The model checking approach discussed here also allows other data items besides clocks, but the main focus will be on dealing with zones defined by constraints on real-valued variables, that arise as a consequence of the clocks in timed automaton models.

As was mentioned already, the applicability and efficiency of state space representations is strongly related to the type of constraints that can occur in the state space. For real-valued variables three important categories can be distinguished:

- **difference constraints**: constraints of the form \( x - y \sim c \) and \( x \sim c \) where \( x \) and \( y \) are variables, \( c \) is an integer constant and \( \sim \in \{<, \leq\} \).
Introduction

- linear constraints: constraints of the from \( ax + by + \cdots \sim c \), where \( a, b \) and \( c \) are integer constants, \( x \) and \( y \) are variables, and \( \sim \in \{ <, \leq \} \).

- non-linear constraints

In the timed automata models used in many dense real-time model checking approaches, the expressiveness of updates and constraints is limited such that the resulting states sets can be characterized by means of difference constraints. For such state sets, efficient representations can be found. Difference bound matrices (DBM’s) [47] are often used in this context, but also alternative approaches have been used [75, 85]. Model checkers like Uppaal [73] and Kronos [44] rely on the limitation to difference constraints. For linear constraints, things are getting considerably more complex. Only a few model checkers implement the capability to deal with linear constraints. HyTech [9] was the first symbolic model checking tool to deal with state spaces defined by linear constraints, using a convex polyhedra representation [53, 58]. Finally, for non-linear constraints, model checking becomes mostly infeasible, although for some limited problems, solutions have been found [57]. We will focus on dealing with the first two categories of constraints.

The type of constraints that can be represented determines the type of systems and verification problems that can be handled by the model checker. If a system specification contains only difference constraints on clocks (and a similar restriction to updates is applied) then only difference constraints will occur in the exploration. Likewise, if non-difference, linear constraints occur in a system specification, linear constraints will occur in the exploration. Furthermore, note that if besides clocks also discrete real variables are allowed in specifications, and these variables are mixed with clocks in guards or updates, non-difference constraints will occur, even if the system specification contains only difference constraints. As a consequence, most parametric real-time verification problems will need a model checker that is based on linear constraints. The reason for this is that when using parameters in formalisms like XTG, parameters (which are discrete real variables) are often mixed with clocks. Finally, being able to deal with linear constraints also enables the verification of (a limited set of) system models with arbitrary continuous variables, like for example hybrid automata [6].

Ideally, one should match the symbolic state space representation to the type of verification problem — more specifically, to the "complexity" category of the symbolic state space that is induced by it. The latter is a result of the data types as well as the constraints and updates allowed in system models. This is why we built two implementations of our model checking approach, one for difference constraints on real variables and one for linear constraints on real variables.

Traditionally, in symbolic model checking canonical representations have been used. However, the time and data resources needed for maintaining such canonical representations have great impact on the performance of the model checking techniques. Most model checking approaches require several operations on zones (union, intersection, difference, time projection) which justifies the burden of using canonical representations, because these operations are performed relatively efficiently on such representations. When applying a partition refinement technique, this is not a natural approach because the required operations on the state space are very limited and specific. In fact, only two basic operations are required, namely splitting and propagation. In this case, there appears to be no advantage in using canonical structures like DBM’s. As we will show, the required operations can be implemented in a surprisingly simple way on state spaces represented by a dedicated, non-standard representation.

Besides clocks, other data types can be used in system modelling. For such data
5.2 Splitting and propagation

As became clear in chapter 3, two basic operations on the symbolic state space occur in partition refinement exploration, namely constraint_split and propagated_split. The first is needed to ensure the enabling aspect of stability and enforce that the initial partition (i.e. atomic properties and fairness properties) is respected. It simply splits a region into a set of subregions that respect a given constraint (originating from a guard, an invariants, an atomic property or a fairness property).

Splitting a region means that the zone that defines the region is split. Formally, splitting a region \( \langle l, Z \rangle \) by a constraint \( c \) results in a set of regions \( \{ \langle l, Z' \rangle \mid Z' \in ZZ \} \) such that

\[
Z = \bigcup ZZ \land \forall Z' \in ZZ .\ (Z' \subseteq Z \cap [c] \lor Z' \subseteq Z \setminus [c])
\]

By defining splitting this way, we do not define exactly how the zone is split, but only specify what the result should be, namely that the resulting subzones of \( Z \) respect \( c \). Obviously, splitting in a minimal manner would result in \( ZZ = \{ Z \cap [c], Z \setminus [c] \} \).

The second operation, propagate_split, is needed for maintaining the propagation aspect of stability. Given a region that is instable with respect to an outgoing edge, the region is split into subregions that are again stable. In a partition refinement algorithm for real-time systems three variations of propagated splitting occur:

1. If a region has a discrete edge to another region and the latter region is becomes split, then this might cause the former region to become instable. To fix this situation, the region is split into subregions that are again stable with respect to the edge. Formally, consider a stable region graph edge \( \langle l_1, Z_1 \rangle \rightarrow_e \langle l_2, Z_2 \rangle \) with \( e = \langle l_1, g, u, l_2 \rangle \), where \( Z_2 \) has been split into subzones \( Z_{2,1} \ldots Z_{2,n} \). Propagating this split back through the edge means that \( Z_1 \) has to be split into subzones \( Z_{1,1} \ldots Z_{1,m} \) such that \( \forall i \in \{1, \ldots, m\} .\ \forall j \in \{1, \ldots, n\} .\ pre_u(Z_{1,j}, Z_{2,i}) \in \{\emptyset, Z_{1,j}\} \).

2. If the destination region of a time edge is split, a different kind of propagation is to be performed, because the passing of time has to be taken into account. Formally, consider a stable region graph edge \( \langle l_1, Z_1 \rangle \rightarrow_e \langle l_2, Z_2 \rangle \) with \( Z_1 \neq Z_2 \), where \( Z_2 \) has been split into subzones \( Z_{2,1} \ldots Z_{2,n} \). Propagating this split back through the edge means that \( Z_1 \) has to be split into subzones \( Z_{1,1} \ldots Z_{1,m} \) such that \( \forall i \in \{1, \ldots, n\} .\ \forall j \in \{1, \ldots, m\} .\ pre_e(Z_{1,j}, Z_{2,i}) \in \{\emptyset, Z_{1,j}\} \).

3. A special case of this situation occurs if a region is split by a clock constraint (i.e. a constraint involving one or more clocks). This may result in instable time edges between the newly obtained subregions. In that case an additional split is needed. Formally, consider a region \( \langle l, Z \rangle \) which is stable with respect to an outgoing time edge (if it has one). Suppose that \( Z \) is split into subzones \( Z_1 \ldots Z_n \). Then each subzone \( Z_i \) has to be split such that a set of zones \( Z_1 \ldots Z_m \) is obtained for which \( \forall i \in \{1, \ldots, m\} .\ \forall j \in \{1, \ldots, m\} .\ pre_e(Z_i, Z_j) \in \{\emptyset, Z_i\} \). In fact, this case coincides with the second variation in case of a self-looping time-edge. We describe it separately because it requires a somewhat different treatment.
Note that the third case cannot occur as a consequence of a propagated time split, since
the regions resulting from a propagated time split are always stable with respect to time
edges. Thus, the third case only applies if a region \( (l, Z) \) has been split as a result of a
constraint_split operation, or if it has been split because of propagation over a discrete
dge.

5.3 Splitting trees

Now that we have described the nature of the operations on the state space, a representa-
tion for the state space can be defined.

5.3.1 An introduction to splitting trees

An efficient representation for zones is needed that allows efficient implementations of the
splitting and propagation operations. Zones that arise in infinite-state symbolic model
checking algorithms are defined by constraints on variables. Thus, in fact a representation
for such constraints is needed.

In principle, the XTG formalism allows variables of any data type. For each data type,
a representation is needed for representing the aspects of zones that are associated with
that data type. To be able to treat the representation of each data type separately, we
assume that each constraint can be expressed as a conjunction of constraints, one for each
data type. This means that a zone \( Z \) can be defined as a set \( Z_1, \ldots, Z_n \) of "aspect" zones,
one for each data type (thus \( Z = \bigwedge_i Z_i \)). We will come back to this assumption later.

The most prominent data type that needs to be dealt with are the real variables, be-
cause the clocks belong to this type (in fact, for most timed automaton variants, clocks are
the only type of variables). In many symbolic real-time model checking approaches, canonical
representations are used to represent zones over real variables. These are generally
applicable approaches for representing sets of constraints over real variables.

A much-used canonical representation for convex zones defined by difference constraints
is the difference bound matrices (DBM) representation [47]. A DBM represents a set of
difference constraints as a weighted directed graph, having a vertex for each variable. A
weighted edge between two vertices defines a bound on the difference between the variables
corresponding to the source and destination vertex (thus defining a constraint \( x - y < c \)).
A special 0-vertex is added to represent the \( x < c \) type of constraints. By requiring that
the set of constraints represented by the graph includes all possible implied constraints, a
canonical representation is obtained. This representation allows efficient implementation
of several operations on convex zones. Note that non-convex zones have to be represented
as sets of DBM’s.

The central idea behind canonical representations like DBM’s is that a canonical model
is maintained for each relevant convex zone, allowing efficient implementation of operations
like set union and difference. The representation we will use is rather tailored to partition
refinement approaches, because it is able to efficiently deal with the two operations required
for such approaches. For these operations it is not necessary to have a canonical model
available.

In partition refinement, zones are the consequence of repeated splitting. It therefore
seems natural to represent zones as leafs of so-called splitting trees. A splitting tree is a
binary tree that records a history of splits. Internal nodes are labelled with constraints,
which represent the splitting of a zone into two subzones, one satisfying the constraint and
one dissatisfying the constraint. Each node, internal or leaf, represents a zone. The set of constraints defining the zone that is associated with a node is obtained by collecting all constraints while traversing the tree upwards from the node, negating constraints when appropriate. The set of leaf nodes defines a complete partitioning of the set of valuations.

![Figure 5.1: An example of a splitting tree](image)

Figure 5.1 shows an example splitting tree dividing the value space into six zones. In this figure, right-side children hold the subzones that satisfy the constraint of the parent node, while left-side children hold the subzones that dissatisfy the constraint of the parent node. For example, the zone $Z_5$ represents the set of valuations that satisfy $x < 3 \land y > 0 \land \neg z > 4$

This leads to a forest structure in which for every location, a splitting tree is kept that holds the splitting history that is relevant for that location. The set of leaf nodes of the tree associated with a location represents the complete set of zones that are relevant for that location.

### 5.3.2 Definition of splitting trees

**Definition 5.1 (splitting trees).** A splitting tree over a set of variables $V$ is a binary tree defined by a tuple $\langle n_t, N_I, N_L \rangle$ where

- $n_t$ is the top node
- $N_I$ is a set of internal nodes
- $N_L$ is a set of leaf nodes

Together with following attributes:

- $\text{cons} : (\{n_t\} \cup N_I) \to \text{Expr}_V$ assigns a constraint to each non-leaf node.
- $\text{parent} : (N_I \cup N_L) \to (\{n_t\} \cup N_I)$ returns the parent of each non-top node.
- $\text{type} : (N_I \cup N_L) \to \{\text{SAT}, \text{DISSAT}\}$ returns the type of each non-top node.
- $\text{left} : (\{n_t\} \cup N_I) \to (N_I \cup N_L)$ returns the "left" child of each non-leaf node.
- $\text{right} : (\{n_t\} \cup N_I) \to (N_I \cup N_L)$ returns the "right" child of each non-leaf node.

such that for each $n \in (n_t \cup N_I)$

$$\text{type}(\text{left}(n)) = \text{DISSAT \ and \ type}(\text{right}(n)) = \text{SAT \ and}$$

$$\text{parent}(\text{left}(n)) = n \ \text{and \ parent}(\text{right}(n)) = n$$

$ST_V$ denotes the set of possible splitting trees over $V$. \qed
The type of the node indicates whether or not the constraint of its parent node should be negated or not.

Each splitting tree node defines a zone, defined by a conjunction of single constraints.

**Definition 5.2 (semantics of splitting tree nodes).** Let \((n_l, N_l, N_r)\) be a splitting tree. Then the function \(S : (N_l \cup N_r \cup \{n_r\}) \rightarrow \text{Expr}_V\) assigns to each splitting tree node a constraint, as defined below. We define \(S[n] = S[n]^{n'}\) where

\[
S[n]^n = \text{true} \\
S[n]^{n'} = \begin{cases} 
S[\text{parent}(n)]^{n'} \land \text{cons}(\text{parent}(n)) & \text{if } \text{type}(n) = \text{sat} \\
S[\text{parent}(n)]^{n'} \land \neg\text{cons}(\text{parent}(n)) & \text{if } \text{type}(n) = \text{dissat}
\end{cases}
\]

for \(n \neq n'\) \(\square\)

With each location we can associate a single splitting tree, to represent all regions associated with that location. This leads to the following representation for partially explored state spaces.

**Definition 5.3 (a representation for regions).** Consider an XTG \(X\) with variables \(V\) and locations \(L\). Let \(L_{\text{exp}} \subseteq L\) be the set of (partially or fully) explored locations. Then a state space for the set of the explored locations \(L_{\text{exp}}\) is given by a function \(\text{forest} : L_{\text{exp}} \rightarrow ST_V\).

A region of this state space is represented by a tuple \(\langle l, n \rangle\) where \(l \in L_{\text{exp}}\) is a location and \(n \in N_L\), with \((\neg, \neg, N_L) = \text{forest}(l)\), holds an appropriate leaf node. \(\langle l, n \rangle\) is referred to as a region representation. \(\square\)

**Definition 5.4 (semantics for region representations).** The function \(F : (L \times N_L) \rightarrow (L \times \text{Expr}_V)\) assigns to each region representation \(\langle l, n \rangle\), a region:

\[
F[\langle l, n \rangle] = \langle l, [S[n]] \rangle
\]

\(\square\)

Note that a splitting tree defines a complete partitioning of a value space. This means that the splitting-tree representation also represents regions that are not part of the state space because these not reachable or excluded by an invariant.

In the subsequent chapter we will be using the splitting tree representation in the model checking algorithm described there. To that end, we define a convenient set of operations.

**Definition 5.5 (forest operations).** Consider a forest \(\text{forest}\), a region representation \(\langle l, n \rangle\) from \(\text{forest}\) and a constraint \(c\). Let \(\text{forest}(l) = \langle n_l, N_l, N_r \rangle\). Then the following operations are defined.

- \(\text{new\_tree}(l)\) for \(l \notin L_{\text{exp}}\), denotes the extension of the forest with a new location. As a result of \(\text{new\_tree}(l)\), \(l\) is added to \(L_{\text{exp}}\) and \(\text{forest}(l) = \langle n, \emptyset, \{n\} \rangle\), where \(n\) is a new node.
- \(\text{get\_tree}(l) = \langle l, n_l \rangle\) for \(l \in L_{\text{exp}}\)
- \(\text{extend}(\langle l, n \rangle, c)\) denotes the extension of \(\langle l, n \rangle\) by splitting according to \(c\). It results in an updated forest in which \(\text{forest}(l)\) is updated to become \(\langle n_l, N_l \cup \{n\}, N_r \setminus \{n\} \cup \{n_l, n_r\} \rangle\), where \(\text{cons}, \text{type}, \text{parent}, \text{left}\) and \(\text{right}\) are updated as follows
  - \(\text{cons}(n_l) = c\)
  - \(\text{type}(n_l) = \text{dissat}\) and \(\text{type}(n_r) = \text{sat}\).
  - \(\text{parent}(n_l) = n\) and \(\text{parent}(n_r) = n\)
  - \(\text{left}(n) = n_l\) and \(\text{right}(n) = n_r\)
and where \( n_l \) and \( n_r \) are two new nodes.

- \( \text{left}((l,n)) = (l, \text{left}(n)) \) for any non-leaf node \( (l,n) \)
- \( \text{right}((l,n)) = (l, \text{right}(n)) \) for any non-leaf node \( (l,n) \)
- \( \text{parent}((l,n)) = (l, \text{parent}(n)) \) for any non-top node \( (l,n) \)
- \( \text{location}((l,n)) = l \)
- \( \text{sub}^*(n) = \begin{cases} \{n\} & \text{if } n \in N_L \\ \{n\} \cup \text{sub}^*(\text{left}(n)) \cup \text{sub}^*(\text{right}(n)) & \text{otherwise} \end{cases} \)

If one of the \text{left}, \text{right} or \text{parent} is inappropriately applied (for example, applying \text{parent} to a top node), \( \bot \) is returned.

### 5.3.3 Restrictions

To use splitting trees for model checking, some restrictions are needed on the form of the constraints that can be assigned to internal nodes. The form of these restrictions is specific for each data type that is allowed and depends on the possibilities for defining the required operations.

The most important data type for a timed automaton formalism is the real type, since this includes the clocks. Like in the example of figure 5.1, we want to limit the constraints in the tree to linear equations. Complex constraints that require boolean operators (e.g. \( x > 4 \land x < 8 \)) are not allowed. By restricting constraints this way, the resulting zones will always be convex, since the intersection of a set of linear equations always results in a convex zone. As will be shown later, this allows us to efficiently implement the required operations.

As a consequence, splitting trees can only be used to represent zones defined by conjunctions of inequalities. In XTG, arbitrary boolean combinations of linear inequalities are still allowed. This means that zones defined by disjunctions have to be divided into smaller zones that can be defined purely by conjunctions. Also this means that splitting a zone by a complex constraint has to be decomposed in a number of subsequent splits by single constraints.

For other data types, similar limitations may be needed. We use the term single constraints to denote constraints that are the atomic constraints for the corresponding data type that are allowed in splitting trees. Thus, single constraints do not contain any boolean operators.

The model checkers that resulted from the work described in this thesis, implement three data types, namely real variables, integer variables and enumeration variables, although the approach is easily extended to deal with other types. For our data types, allowed single constraints are defined as follows.

- For real variables, difference inequalities or linear inequalities, depending on the reduction algorithm (see section 5.5).
- For integer variables, difference inequalities.
- For enumeration-type variables, restrictions are not needed. We assume the following assignments on enumerated types \( X ::= Y \) and \( X ::= c \) and the following constraints \( X = Y, \ X \neq Y, \) and \( X \in \{c_1, \ldots, c_n\} \), where \( X, Y \) and \( Z \) are variables and \( c \) and \( c_i \) denote constants. The form \( X \in \{c_1, \ldots, c_n\} \) is a generalization constraints like \( X = c \) and \( X \neq c \).
In the coming sections, it is shown that implementing constraint_split and propagated_split on a symbolic state space requires two basic operations, called propagation and reduction.

## 5.4 Propagation

Consider a stable edge \( \langle l_1, Z_1 \rangle \xrightarrow{e} \langle l_2, Z_2 \rangle \) and suppose that \( \langle l_2, Z_2 \rangle \) has been split into subregions \( \langle l_2, Z_{2,1} \rangle \ldots \langle l_2, Z_{2,n} \rangle \). In early partition refinement approaches (e.g. [4]) propagating the split of \( \langle l_2, Z_{2,i} \rangle \) back to \( Z_1 \) was done by splitting \( Z_1 \) into \( \text{pre}_u(Z_1, Z_{2,i}) \) and \( Z_1 \setminus \text{pre}_u(Z_1, Z_{2,i}) \) for each zone \( Z_{2,i} \in \{Z_{2,1}, \ldots, Z_{2,n}\} \). For most types of zones — in particular convex zones over real variables — a simpler procedure [109] can be used. It suffices to only compute \( \{\text{pre}_u(Z_1, Z_{2,1}), \ldots, \text{pre}_u(Z_1, Z_{2,n})\} \), because under the restriction of convexity the resulting set of zones is a proper partition of \( Z_1 \). The advantage is that the usually costly set complement operation needed for the set difference, is avoided. This approach works similarly for time edges, where \( \text{pre}_u \) is replaced by \( \text{pre}_\tau \). However, it requires that the zones have a canonical representation (unlike splitting trees), that allows efficient computation of the \( \text{pre}_u \) and \( \text{pre}_\tau \) functions.

We use a different approach, specific for our splitting tree representation. Thanks to the fact that the splitting history of zones is recorded in splitting trees, we can implement the propagation operation in a simple manner.

![Figure 5.2: Propagation and splitting trees](image)

The basic idea is to perform the propagated splitting by following the splitting history of the split region that causes the instability. As an example, consider a stable discrete edge \( \langle l_1, Z_1 \rangle \xrightarrow{e} \langle l_2, Z_2 \rangle \), where \( Z_2 \) is split into a set of subzones. This splitting is recorded in a fragment of the splitting tree of \( l_2 \). Let \( c_1 \) denote the constraint associated with the top node of the splitting tree fragment. This is illustrated in the splitting tree fragment shown in the left-hand side of figure 5.2. There, the arrow represents the discrete edge \( \langle l_1, Z_1 \rangle \xrightarrow{e} \langle l_2, Z_2 \rangle \).

In this situation, the region \( \langle l_1, Z_1 \rangle \) is potentially instable, due to the outgoing edge to the split region \( \langle l_2, Z_2 \rangle \). Suppose that it is indeed instable. Then we would be looking for the constraint \( c_1' \) that splits \( Z_1 \) into two subzones \( Z_1 \setminus [c_1'] \) and \( Z_1 \cap [c_1'] \) which are stable with respect to edges to \( Z_2 \setminus [c_1] \) and \( Z_2 \cap [c_1] \), respectively. This is illustrated in the right-hand fragment of figure 5.2.
Now, the procedure can be repeated for the new edges $\langle l_1, Z_1 \setminus [c'_1] \rangle \xrightarrow{e} \langle l_2, Z_2 \setminus [c_1] \rangle$ and $\langle l_1, Z_1 \cap [c'_1] \rangle \xrightarrow{e} \langle l_2, Z_2 \cap [c_1] \rangle$. The resulting iteration ends when the leaf nodes of the splitting tree are reached, in which case the resulting regions are stable.

As was discussed before, we assume that all constraints in splitting trees are single constraints. This means that the problem of propagation can be defined in terms of the simpler problem of propagating a split by a single constraint. For discrete edges this can be formalized as follows. Consider a stable region graph edge $\langle l, Z_1 \rangle \xrightarrow{e} \langle l, Z_2 \rangle$. Suppose that $\langle l, Z_2 \rangle$ is split by a single constraint $c'$ into sub-regions $\langle l, Z_2 \cap [c] \rangle$ and $\langle l, Z_2 \setminus [c] \rangle$.

Then we are looking for a single constraint $c'$ that splits $Z_1$ into subzones $\langle l, Z_1 \cap [c'] \rangle$ and $\langle l, Z_1 \setminus [c'] \rangle$ such that either $\langle l, Z_1 \cap [c'] \rangle$ is empty or $\langle l, Z_1 \setminus [c'] \rangle \xrightarrow{e} \langle l, Z_2 \setminus [c] \rangle$ is stable, and that either $\langle l, Z_1 \setminus [c'] \rangle$ is empty or $\langle l, Z_1 \setminus [c'] \rangle \xrightarrow{e} \langle l, Z_2 \setminus [c] \rangle$ is stable. Note that $c'$ could be true or false, in which case no splitting is required. For time edges, there is a similar formulation.

Thus, the problem to be solved is to find single constraint propagation operations for the different types of single constraints that can occur. A distinction has to be made between propagation over a discrete edge and propagation over a time edge. We therefore define two operations, $\text{propD}$ and $\text{propT}$, that will implement propagation over discrete edges and internal edges, respectively. $\text{propT}$ is only relevant for the domain of reals (since clocks are real variables). $\text{propD}$ on the other hand, has to be defined for each relevant data type. Note that we assume here that single constraints are closed under the application of the propagation operation, i.e., that propagation of a split by a single constraint results in a split by another single constraint. As will be shown, this is the case for propagation over time edges. For propagation over discrete edges this will depend on the data type, but for regular data types, like reals and enumerated types this will be the case.

In some cases, splitting will not be needed because the source region appears to be already stable. In terms of our example this would mean that there is either a stable edge from $\langle l_1, Z_1 \rangle$ to $\langle l_1, Z_2 \setminus [c_1] \rangle$ or to $\langle l_2, Z_2 \cap [c_1] \rangle$.

Below are some properties of $\text{pre}_r$ and $\text{pre}_u$ that will be needed in the coming subsections.

**Theorem 5.1 (properties of $\text{pre}_r$ and $\text{pre}_u$).** For any zones $X, Y, Z$ and $W$ the following properties hold

1. $\text{pre}_u(X, Y) \cap \text{pre}_u(X, Z) = \text{pre}_u(X, Y \cap Z)$
2. $\text{pre}_u(\text{pre}_u(X, Y), Y) = \text{pre}_u(X, Y)$
3. $X \subseteq Y \Rightarrow X \cap \text{pre}_u(Y, Z) = \text{pre}_u(X, Z)$
4. $\text{pre}_u(X, Y) \cup \text{pre}_u(X, Z) = \text{pre}_u(X, Y \cup Z)$
5. $(Z \subseteq X \land X = \text{pre}_u(X, Y)) \Rightarrow Z = \text{pre}_u(Z, Y)$
6. $X \subseteq Y \Rightarrow \text{pre}_r(Z, X) \subseteq \text{pre}_r(Z, Y)$
7. $X \subseteq Y \Rightarrow \text{pre}_r(X, Z) \subseteq \text{pre}_r(Y, Z)$
8. $\text{pre}_r(X, Y) = \text{pre}_r(\text{pre}_r(X, Y), Y)$
9. $(X \cup Y) \cap Z = \emptyset \land X \cap Y = \emptyset \Rightarrow \text{pre}_r(Z, X) \cap \text{pre}_r(Z, Y) = \emptyset$
10. $\text{pre}_r(X, Y) = X \land \text{pre}_r(W, Y) = W \land X \subseteq W \land Z \subseteq Y \land Y \cap W = \emptyset \Rightarrow \text{pre}_r(X, Z) = X$

$\square$
Proof. Properties 1 to 8 follow immediately from the definitions of \( \text{pre}_u \) and \( \text{pre}_c \). Property 9 will be proved by considering a \( \rho \in \text{pre}_c(Z,X) \cap \text{pre}_c(Z,Y) \) and showing inconsistency with the left-hand side of the implication. According to the definition of \( \text{pre}_c \), \( \rho \in \text{pre}_c(Z,X) \cap \text{pre}_c(Z,Y) \) means that there must exist \( \delta_1 \geq 0 \) and \( \delta_2 \geq 0 \) such that

\[
\rho[+\delta_1] \in X \land \forall \delta'_1. (0 < \delta'_1 < \delta_1 \Rightarrow \rho[+\delta'_1] \in X \cup Z) \quad \text{and} \\
\rho[+\delta_2] \in Y \land \forall \delta'_2. (0 < \delta'_2 < \delta_2 \Rightarrow \rho[+\delta'_2] \in Y \cup Z)
\]

If \( \delta_1 = \delta_2 \) then \( \rho[+\delta_1] \in X \) and \( \rho[+\delta_1] \in Y \) which contradicts the left-hand side of the implication. If \( \delta_1 > \delta_2 \) we know that \( \rho[+\delta_2] \in Y \) and that \( \rho[+\delta_2] \in X \cup Z \). This also contradicts the left-hand side of the implication. The case for \( \delta_1 < \delta_2 \) is the same.

To prove property 10, we show that given the condition in the left-hand side of the implication, \( X \subseteq \text{pre}_c(X,Z) \) — thus that for any \( \rho \) that satisfies the left-hand side of the implication, it must be the case that \( \rho \in X \Rightarrow \rho \in \text{pre}_c(X,Z) \). From \( \text{pre}_c(X,Y) = X \land \text{pre}_c(W,Z) = W \) it follows that for any \( \rho \in X \subseteq W \) there must be \( \delta_1 \geq 0 \) and \( \delta_2 \geq 0 \) such that

\[
\rho[+\delta_1] \in Y \land \forall \delta'_1. (0 < \delta'_1 < \delta_1 \Rightarrow \rho[+\delta'_1] \in X \cup Y) \quad \text{and} \\
\rho[+\delta_2] \in Z \land \forall \delta'_2. (0 < \delta'_2 < \delta_2 \Rightarrow \rho[+\delta'_2] \in W \cup Z)
\]

Again the three cases \( \delta_1 = \delta_2, \delta_1 > \delta_2 \) and \( \delta_1 < \delta_2 \) can be distinguished. For each of these cases one can derive (using \( Z \subseteq Y \) and \( Y \cap W = \emptyset \)) that \( \rho \in \text{pre}_c(X,Z) \). \( \square \)

5.4.1 Propagation over discrete edges

Propagation of discrete edges is relatively straightforward. To propagate a split over a discrete edge, we have to know the update associated with the XTG edge from which the discrete region graph edge originates. Consider a stable edge \( (l_1, Z_1) \xrightarrow{(l_2, Z_2)} (l_2, Z_2) \), and a constraint \( c \) that splits \( Z_2 \). We use the fact that \( \text{pre}_u(Z_1, Z_2 \cap [c]) = \text{pre}_u(Z_1, Z_2) \cap \text{pre}_u([\text{true}], [c]) \), meaning that the constraint that stabilizes \( Z_1 \) can be computed from \( u \) and \( c \) only. Consider the example in figure 5.3. For clarity, the edges are labelled with updates instead of the XTG edges containing them. Here, \( Z_2 \) is split by a constraint \( x < 5 \). This causes the \( Z_1 \) to become instable, because some of its states will have transitions to the region represented by the zone satisfying \( x < 5 \), while others will have transitions to the region represented by the zone satisfying \( x \geq 5 \). Propagating the split of \( Z_2 \) to \( Z_1 \) requires that \( Z_1 \) is split by \( y < 3 \), which is computed by substituting the right-hand side of the assignment for \( x \) in the original constraint.

![Figure 5.3: Propagation for real variables over discrete edges](image)

Thus, the required \( \text{propD}(c, u) \) procedure returns a constraint \( c' \) such that for each valuation \( \rho \) that satisfies \( c' \), \( \rho[u] \) satisfies \( c \), and that for each \( \rho \) that dissatisfies \( c' \), \( \rho[u] \) dissatisfies \( c \). The definition below implicitly defines \( \text{propD}(c, u) \), independent of any data type.
Definition 5.6 (\(propD\)). Given a constraint \(c\) and an update \(u\), let \(propD(c, u)\) be a function that returns a constraint such that:

\[propD(c, u) = \text{pre}_u([\text{true}], [c])\]

The theorem below justifies the use of \(propD\) for propagation over discrete edges.

Theorem 5.2 (propagation over discrete edges). Let \((\ell_1, Z_1) \xrightarrow{(l_1, g, u, l_2)} (\ell_2, Z_2)\) be a stable region graph edge and let \(c\) be a (single) constraint that splits \(Z_2\). Then

- \((\ell_1, Z_1 \cap [propD(c, u)]) \xrightarrow{(l_1, g, u, l_2)} (\ell_2, Z_2 \cap [c])\)
- \((\ell_1, Z_1 \setminus [propD(c, u)]) \xrightarrow{(l_1, g, u, l_2)} (\ell_2, Z_2 \setminus [c])\)

are stable region graph edges.

Proof. From theorem 3.4 it follows that we have to prove the following two properties

1. \(Z_1 \cap [propD(c, u)] = \text{pre}_u(Z_1 \cap [propD(c, u)], Z_2 \cap [c])\) and
2. \(Z_1 \setminus [propD(c, u)] = \text{pre}_u(Z_1 \setminus [propD(c, u)], Z_2 \setminus [c])\)

Note that it follows from stability of the original region graph edge (theorem 3.4) that

\[Z_1 = \text{pre}_u(Z_1, Z_2)\] (5.1)

The first case is proved as follows:

- \(Z_1 \cap [propD(c, u)] = Z_1 \cap \text{pre}_u([\text{true}], [c]) = \text{[theorem 5.1, property 3]}\)
- \(Z_1 \cap \text{pre}_u(Z_1, [c]) = \text{[equation 5.1]}\)
- \(\text{pre}_u(Z_1, Z_2) \cap \text{pre}_u(Z_1, [c]) = \text{[theorem 5.1, property 1]}\)
- \(\text{pre}_u(Z_1, Z_2 \cap [c]) = \text{[theorem 5.1, property 2]}\)
- \(\text{pre}_u(Z_1 \cap [propD(c, u)], Z_2 \cap [c])\)

The second case:

- \(Z_1 \setminus [propD(c, u)] = Z_1 \setminus (Z_1 \cap [propD(c, u)]) = \text{[proof of the first case]}\)
- \(Z_1 \setminus \text{pre}_u(Z_1, Z_2 \cap [c]) = \text{[equation 5.1]}\)
- \(\text{pre}_u(Z_1, Z_2 \setminus [c]) = \text{[theorem 5.1, property 4]}\)
- \(\text{pre}_u(Z_1, Z_2 \setminus [c]) = \text{[theorem 5.1, property 2]}\)
- \(\text{pre}_u(Z_1 \setminus [propD(c, u)], Z_2 \setminus [c])\)

This completes the proof of the theorem. 

For an actual implementation, \(propD\) has to be made explicit for each relevant data type. For the most important one, namely for the case of linear inequalities over real variables, a definition is given below. Let \(I = \{1, \ldots, n\}\) be an index set for the set \(V = \{x_1, \ldots, x_n\}\) of all real-valued variables, including clocks. First, assume that the update \(u\) consists of a single assignment. Without loss of generality, suppose that this assignment is formulated as follows: \(x_j := \sum_{i \in I} a_i x_i + b\) with \(j \in I\), and that the constraint \(c\) is formulated as follows: \(\sum_{i \in I} c_i x_i \sim d\), with \(\sim \in \{<, \leq, >, \geq\}\). Then one can show

\[propD(c, u) = \sum_{i \in I \setminus \{j\}} (c_i + c_j a_i) x_i + (c_j a_j) x_j \sim d - c_j b\]

In case \(u\) consists of more than one assignment, the computation can be repeated for each assignment. Thus \(propD(c, \{a\} \cup u) = propD(propD(c, a), u)\).
<table>
<thead>
<tr>
<th>$Y = X$</th>
<th>$X := Z$</th>
<th>$X := c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y \neq X$</td>
<td>$Y \neq Z$</td>
<td>$Y \in {c}$</td>
</tr>
<tr>
<td>$Y \in {c_1, \ldots, c_n}$</td>
<td>$Y \in {c_1, \ldots, c_n}$</td>
<td>$Y \in V \setminus {c}$</td>
</tr>
</tbody>
</table>

Figure 5.4: Propagation for enumerated types

Obviously, for integer variables, the definition would be the same. In fact, for any data type one wishes to include, it should be possible to define propagation procedures. Here, we will also consider enumerated types. Figure 5.4 shows the propagation procedure. The left column shows the original splitting constraints, and the top row shows the assignment associated with the edge over which propagation takes place. The constraints in the body of the table show the constraints that result from propagation.

As a final note, from theorem 5.1, property 5 it follows that splitting a region into smaller regions, retains stability with respect to outgoing discrete edges. Thus, when splitting a region $r$, the resulting subregions inherit the outgoing discrete edges of the original region, maintaining stability. Thus, if a $r \xrightarrow{e} r'$ was stable, then for each subregion $rr$ of $r$, $rr \xrightarrow{e} r'$ is also stable.

5.4.2 Propagation over time edges

As was mentioned already in section 5.2, there are two cases of propagation over time edges. The first case is illustrated in figure 5.5, which shows partitions into zones of the value space. In the left side of that figure, the zone defined by the single constraint $c \geq 3$ (indicated by the solid line) is split by a constraint $d < 5$ (indicated by a dashed line), $c$ and $d$ both being clocks. The region corresponding to the zone satisfying $c < 3$ has a time edge to the region corresponding to the zone $c \geq 3$. The former becomes instable as a result of the split of the latter, since not all states in the region represented by the zone satisfying $c < 3$ have a time edge to the states in the region represented by the zone $c \geq 3 \land d < 5$. This can be resolved by splitting the zone satisfying $c < 3$ with the constraint $d - c < 2$, as is shown in the right side of the figure. As a result of the propagation, the resulting regions are again stable with respect to time edges.

![Figure 5.5: Propagation over time edges](image)

The second type of propagation is illustrated in figure 5.6. In this case the instability is introduced in the zone that has been split itself. If the region defined by $c < 3$ is split by a constraint $d < 5$, then this results in a subregion satisfying $c < 3 \land d < 5$, which is clearly instable. A similar split as in the first case is needed here. The constraint $d - c \leq 2$ separates the states having time edges to $c \geq 3$ from those that have time edges
to \( c < 3 \land d \geq 5 \).

![Figure 5.6: Internal time propagation](image)

Note that in the following we restrict ourselves to linear constraints on real-valued variables. For non-linear constraints, the concepts presented below are not always valid. Thus, unless otherwise indicated, a constraint refers to a single linear constraint on real-valued variables.

Below some definitions are introduced to enable the definition of the propagation operations.

**Definition 5.7 (bounds).** Let \( c \) be a constraint. Then \( c \) is an upper bound if for any valuation \( \rho \), there is a \( \delta \in \mathbb{R} \geq 0 \) such that \( \rho[+\delta] \) dissatisfies \( c \). \( c \) is a lower bound if for any valuation \( \rho \), there is a \( \delta \in \mathbb{R} \geq 0 \) such that \( \rho[-\delta] \) dissatisfies \( c \). \( c \) is a bound if it is either an upper or a lower bound, while it is a non-bound if it is not a bound.

An upper bound restricts the passing of time. A non-bound can be seen as being ”parallel to the passing time”. Constraints resulting from time-propagation can only be non-bounds, as is illustrated in figures 5.5 and 5.6. Note that the negation of a upper bound constraint is a lower bound, and conversely, while the negation of a non-bound is again a non-bound.

**Definition 5.8 (upper bound of a zone).** A constraint \( cb \) is an upper bound of a zone \( Z \) in case
\[
\forall \rho \in Z . \forall \delta \geq 0 . (\rho[+\delta] \in Z \iff \rho[+\delta] \in [cb])
\]
\( cb \) is a single upper bound of \( Z \) if \( cb \) defines a linear inequality. A zone \( Z \) is unbounded in case
\[
\forall \rho \in Z . \forall \delta > 0 . \rho[+\delta] \in Z
\]

The terms introduced above are sometimes also applied to regions, with the obvious meaning.

**Definition 5.9 (separating constraint).** Let \( Z_1 \) and \( Z_2 \) be two disjunct convex zones such that \( Z_1 = \text{pre}_\tau(Z_1, Z_2) \). Then a constraint \( c \) is referred to as a separating constraint of \( Z_1 \) and \( Z_2 \) if \( Z_1 \cap [c] = Z_1 \) and \( Z_2 \cap [c] = \emptyset \).

For example, in the right-hand diagram of figure 5.6, \( c < 3 \) is the separating constraint of \([d - c \leq 2 \land c < 3]\) and \([c \geq 3]\). Note that because of the formulation in the definition, a separating constraint will always be an upper bound.

It is easy to prove that if \( Z_1 \) and \( Z_2 \) are convex, a separating constraint exists.

**Theorem 5.3 (separating constraint as an upper bound).** Let \( Z_1 \) and \( Z_2 \) be two disjoint convex zones such that \( Z_1 = \text{pre}_\tau(Z_1, Z_2) \). Then a single constraint \( c \) exists that is a separating constraint for \( Z_1 \) and \( Z_2 \). Also, the separating constraint \( c \) is an upper bound of \( Z_1 \).

\( \square \)
Proposition. Since $Z_1$ and $Z_2$ are disjoint and convex, a separating constraint $c$ for $Z_1$ and $Z_2$ must exist. Since $Z_1$ is convex, $c$ must be a definable by a conjunction of linear inequalities. Because $Z_2$ is also convex, the negation of $c$ must also be definable by a conjunction of linear inequalities. Together this means that $c$ must be definable by a single linear inequality.

To prove that $c$ is an upper bound for $Z_1$, it has to be shown that for any $\rho \in Z_1$ and $\delta \geq 0$, it is the case that $\rho[+\delta] \in Z_1 \iff \rho[+\delta] \in [c]$. If $\rho[+\delta] \in Z_1$ then from $Z_1 \subseteq [c]$ it follows that $\rho[+\delta] \in [c]$. For the other direction, if $\rho[+\delta] \notin Z_1$ then because $Z_1 = \text{pre}_s(Z_1, Z_2)$ there must be a $\delta'$ with $0 < \delta' \leq \delta$ such that $\rho[+\delta'] \in Z_2$, which means that because $Z_2 \cap [c] = \emptyset$, $\rho[\delta'] \notin [c]$ and thus because $c$ is an upper bound, $\rho[+\delta] \notin [c]$. □

Theorem 5.4 (unique time-successors). A stable region $\langle l, Z \rangle$ has at most one outgoing time edge.

Proof. If a region $R$ would have two outgoing time edges, one leading to $R_1$ and another one leading to $R_2$, then because of stability each state $s$ in that region would have two outgoing time transitions to different regions, say $s \xrightarrow{\delta_1} s[+\delta_1]$ and $s \xrightarrow{\delta_2} s[+\delta_2]$, with $s[+\delta_1] \in R_1$ and $s[+\delta_2] \in R_2$. Suppose without loss of generality $\delta_1 > \delta_2$. Then according to the definition of region graphs, $s[+\delta_2] \in R \cup R_1$. This cannot be since $s[+\delta_2] \in R_2$ and regions are non-overlapping. □

Figures 5.5 and 5.6 suggest an efficient approach to propagation of single constraint splits over time edges. For the first case, the splitting constraint (in the example, $d - c > 2$) for the source region can be straightforwardly computed from the splitting constraint of the destination region ($d < 5$) and the separating constraint ($c < 3$). For the second case, a similar approach is used. Given a region with an upper bound (in the example, $c < 3$), splitting that region by some constraint ($d < 5$), requires an additional split with can be computed from the splitting constraint and the upper bound. However, there may be cases in which such a single inequality upper bound is not available for the region that is being split. Usually (like in the example) this is not a problem, because the upper bound is the separating constraint associated with an outgoing time edge. However, because of invariants it is possible to have regions (in which time can pass) but which do not have outgoing time edges. Figure 5.7 illustrates this. There, a region is defined by two constraints ($d < 7$ and $c < 3$), which coincide with the invariant of the location. As a consequence, the area $d \geq 7 \lor c \geq 3$ is inaccessible and no outgoing time edges are present. Now if the region is split (for example by $d < 5$), we would not know which constraint to use to compute the stabilizing constraint. In this case, it would be $c < 3$, but for a different splitting constraint, it could be $d < 5$. Thus, the fact that there is no single upper bound for the region complicates the computation considerably. Therefore, we require that each reachable non-urgent region has an upper bound in the form of a single inequality, where a non-urgent region is a region in no outgoing urgent edges are enabled. For the example this means that the region in figure 5.7 would have to be replaced by two regions by splitting it with the constraint $d - c \leq 4$.

The two cases of propagation over time edges can be defined using a function $\text{prop}T$ that, for the moment, we define only implicitly.

Definition 5.10 ($\text{prop}T$). Given two upper bounds $c_1$ and $c_2$, $\text{prop}T(c_1, c_2)$ denotes a function that returns a constraint such that:

$$[c_2] \cap [\text{prop}T(c_1, c_2)] = \text{pre}_s([c_2], [c_1] \setminus [c_2])$$

□
Chapter 5  Symbolic state space representation

Figure 5.7: Illustrating the need for upper bounds

The theorem below formalizes the use of $propT$ for first type of propagation over time edges. It distinguishes two cases which are illustrated in figure 5.8. The first case represents the case that was discussed and illustrated earlier. The second case covers the situation in which a region is split by a constraint that does not interfere with clocks (thus, it is a non-bound). In this case, propagation over time edges is trivial, since it results in an unchanged constraint.

![Figure 5.8: Propagation over time edges](image)

**Theorem 5.5 (propagation over time edges).** Let $Z_1$ and $Z_2$ be two regions such that $Z_1 = pre_\tau(Z_1, Z_2)$. Let $cb$ denote the single constraint that separates $Z_1$ and $Z_2$. Let $c$ be a single constraint that splits $Z_2$ into two non-empty zones.

1. If $c$ is a bound, then (without loss of generality) assume that it is formulated as an upper bound and let $c' = propT(c, cb)$. Then

   \[
   Z_1 \cap [c'] = \text{pre}_r(Z_1 \cap [c'], Z_2 \cap [c]) \quad \text{and} \quad Z_1 \cap [-c'] = \text{pre}_r(Z_1 \cap [-c'], Z_2 \cap [-c])
   \] (5.2)

2. Alternatively, if $c$ is not a bound, let $c' = c$. Then again, equation 5.2 holds.

**Proof.** We start with the case that $c$ is a bound. From the fact that $cb$ is a separating constraint of $Z_1$ and $Z_2$ it follows that $Z_1 \subseteq [cb]$ and $Z_2 \subseteq [-cb]$ and thus also that $Z_2 \cap [c] \subseteq [-cb] \cap [c]$. Using this it follows from theorem 5.1, properties 6 and 7 that

\[
\text{pre}_r(Z_1, Z_2 \cap [c]) \subseteq \text{pre}_r([cb], [-cb] \cap [c])
\] (5.3)

In the same fashion, it follows that

\[
\text{pre}_r(Z_1, Z_2 \cap [-c]) \subseteq \text{pre}_r([cb], [-cb] \cap [-c])
\] (5.4)

Also, using $Z_1 = \text{pre}_r(Z_1, Z_2)$ and thus $Z_1 \subseteq \text{pre}_r(Z_1, Z_2)$, it follows that

\[
\text{pre}_r(Z_1, Z_2 \cap [c]) \cup \text{pre}_r(Z_1, Z_2 \cap [-c]) \subseteq Z_1
\] (5.5)
Also, using the fact that \([c] \cap \lnot c = \emptyset\) and thus \([(c) \cap \lnot cb] \cap (\lnot c) \cap \lnot cb) = \emptyset\), it follows from theorem 5.1, property 9 that

\[
\text{pre}_\tau([cb], [c] \cap \lnot cb) \cap \text{pre}_\tau([-cb], [c] \cap [-cb]) = \emptyset
\] (5.6)

Combining equations 5.3 – 5.6 it then follows using basic set theory that

\[
\begin{align*}
Z_1 \cap \text{pre}_\tau([cb], [c] \cap \lnot cb) &= \text{pre}_\tau(Z_1, Z_2 \cap [c]) \quad \text{and} \\
Z_1 \setminus \text{pre}_\tau([-cb], [c] \cap [-cb]) &= \text{pre}_\tau(Z_1, Z_2 \cap [-c])
\end{align*}
\] (5.7)

Finally, by applying property 8 from theorem 5.1 to the above equalities we get equation 5.2.

For the case that \(c\) is not a bound, it follows from the definition of non-bounds that (since \(c\) and \(\lnot c\) do not constrain the passing of time),

\[
\forall \rho \in [c] \Rightarrow \forall \delta \geq 0 . \rho[+\delta] \in [c] \\
\forall \rho \in [\lnot c] \Rightarrow \forall \delta \geq 0 . \rho[+\delta] \in [\lnot c]
\] (5.8)

Also, from the fact that \(Z_1 = \text{pre}_\tau(Z_1, Z_2)\) it follows that

\[
\forall \rho \in Z_1 \cdot \exists \delta \geq 0 . (\rho[+\delta] \in Z_2 \land \forall \delta' . (0 \leq \delta' < \delta \Rightarrow \rho[+\delta'] \in (Z_1 \cup Z_2)))
\]

Using equation 5.8, this means that

\[
\forall \rho \in Z_1 \cap [c] \cdot \exists \delta \geq 0 . (\rho[+\delta] \in Z_2 \cap [c] \land \forall \delta' . (0 \leq \delta' < \delta \Rightarrow \rho[+\delta'] \in (Z_1 \cup Z_2) \cap [c]))
\]

In other words \(\text{pre}_\tau(Z_1 \cap [c], Z_2 \cap [c]) = Z_1 \cap [c]\). For \(Z_1 \cap [\lnot c]\) the same reasoning applies, using equation 5.9 instead of 5.8 and replacing \(Z_1 \cap [c]\) by \(Z_1 \cap [\lnot c]\).

Note that the choice of \(c\) being an upper bound is not restrictive, because an upper bound can be converted to a lower bound by negating it.

The above theorem facilitates the implementation of propagation over time edges. Consider a stable region graph edge \((l, Z_1) \stackrel{\tau}{\rightarrow} (l, Z_2)\). Let \(c\) be a single constraint that splits \(Z_2\) into two non-empty zones. From the above theorem and theorem 3.4 it then follows that \((l, Z_1 \cap [c]) \rightarrow_{\tau} (l, Z_2 \cap [c])\) and \((l, Z_1 \cap [\lnot c]) \rightarrow_{\tau} (l, Z_2 \cap [\lnot c])\) are stable region graph edges — provided that the resulting regions are non-empty.

While the theorem above dealt with the consequences of splitting a time-successor region, the theorem below deals with consequences of splitting for the region itself (internal time propagation). There are four cases, illustrated in figure 5.9. Only the third case, requires an additional split.

\[
\begin{align*}
\text{c} & \quad \text{c} & \quad \text{cb} & \quad \text{cb} \\
1. & \quad 2. & \quad 3. & \quad 4.
\end{align*}
\]

\text{Figure 5.9: Propagation over time edges}
Finally, by applying property 8 from theorem 5.1 to the above equalities we get

\[ \text{Combining equations 5.14-5.17 it follows using basic set theory that} \]

- If \( Z \) is unbounded, then
  1. If \( c \) is a bound, then \( \text{pre}_\tau(Z \cap [c], Z \setminus [c]) = Z \cap [c] \), assuming that (without loss of generality) \( c \) is formulated as an upper bound.
  2. If \( c \) is a non-bound, then \( Z \cap [c] \) and \( Z \setminus [c] \) are both unbounded (and thus stable with respect to time edges).

- If \( Z \) has upper bound \( cb \), then
  3. if \( c \) is a bound, assume that \( c \) is formulated as an upper bound and let \( c' = propT(cb, c) \). Then

\[ \text{pre}_\tau(Z \cap [c] \cap [c'], Z \setminus [c]) = Z \cap [c] \cap [c'] \] (5.10)

Furthermore, if for some \( Z' \), \( \text{pre}_\tau(Z, Z') = Z \), then

\[ \text{pre}_\tau((Z \cap [c]), Z') = (Z \cap [c]) \cap [c'] \text{ and} \] (5.11)

\[ \text{pre}_\tau(Z \setminus [c], Z') = Z \setminus [c] \] (5.12)

4. If \( c \) is a non-bound, then if for some \( Z' \), \( \text{pre}_\tau(Z, Z') = Z \), then

\[ \text{pre}_\tau(Z \cap [c], Z') = Z \cap [c] \text{ and} \] (5.13)

\[ \text{pre}_\tau(Z \setminus [c], Z') = Z \setminus [c] \] (5.13) \( \square \)

**Proof.** For the cases that \( Z \) is unbounded (1. and 2.), proofs follow immediately.

The case in which \( Z \) has an upper bound \( cb \) and \( c \) is a bound (3.) is the most interesting one. The proof is similar to that of the first case of the previous theorem. From \( Z \cap [\neg c] \subseteq [cb] \cap [\neg c] \) and \( Z \cap [c] \subseteq [cb] \cap [c] \) follows using theorem 5.1, properties 6 and 7 that

\[ \text{pre}_\tau(Z \cap [c], Z \cap [\neg c]) \subseteq \text{pre}_\tau([cb] \cap [c], [cb] \cap [\neg c]) \] (5.14)

In the same fashion it follows that

\[ \text{pre}_\tau(Z \cap [c], [\neg cb]) \subseteq \text{pre}_\tau([cb] \cap [c], [\neg cb]) \] (5.15)

From the fact that for any \( X \) and \( Y \), \( \text{pre}_\tau(X, Y) \subseteq X \), it follows that

\[ \text{pre}_\tau(Z \cap [c], Z \cap [\neg c]) \cup \text{pre}_\tau(Z \cap [c], [\neg cb]) = Z \cap [c] \] (5.16)

From \( [cb] \cap [\neg c] \cap [\neg cb] = \emptyset \) follows using property 9 from theorem 5.1 that

\[ \text{pre}_\tau([cb] \cap [c], [cb] \cap [\neg c]) \cap \text{pre}_\tau([cb] \cap [c], [\neg cb]) = \emptyset \] (5.17)

Combining equations 5.14-5.17 it follows using basic set theory that

\[ (Z \cap [c]) \cap \text{pre}_\tau([c], [cb] \cap [\neg c]) = \text{pre}_\tau(Z \cap [c], Z \cap [\neg c]) \]

\[ (Z \cap [c]) \setminus \text{pre}_\tau([c], [cb] \cap [\neg c]) = \text{pre}_\tau(Z \cap [c], [\neg cb]) \]

Finally, by applying property 8 from theorem 5.1 to the above equalities we get

\[ \text{pre}_\tau(Z \cap [c] \cap [c'], Z \setminus [c]) = Z \cap [c] \cap [c'] \] (5.18)

\[ \text{pre}_\tau((Z \cap [c]) \setminus [c'], [\neg cb]) = (Z \cap [c]) \setminus [c'] \] (5.19)
Equation 5.18 coincides with equation 5.10. Furthermore, equation 5.11 follows from applying property 10 to equation 5.19 and \( \text{pre}_r(Z, Z') = Z \). Finally, equation 5.12 follows immediately from \( \text{pre}_r(Z, Z') = Z \) and the fact that \( c \) is an upper bound (and thus satisfies equation 5.9).

Finally, for the case that \( c \) is a non-bound (4.), then equations 5.8 and 5.9 hold, which combined with \( \text{pre}_r(Z, Z') \), leads to the correctness of equation 5.13.

Summarizing, the three theorems above completely specify how to stabilize the region graph as a consequence of splitting. If a region is split by some constraint, then regions that have discrete edges to that region may become unstable. Theorem 5.2 defines how to split such predecessors to resolve the instability. Theorem 5.5 does the same for regions that have a time edge to the region that was split. Finally, due to self-looping time edges, splitting a region may result in unstable subregions. This is the focus of theorem 5.6. Note that for urgent regions, no additional splits of this type are needed to ensure stability with respect to time edges, since urgent regions (and subregions) cannot have outgoing time edges.

This leaves us with the implementation of prop\( T \), i.e. a concrete formulation of prop\( T \) such that

\[
[c_2] \cap [propT(c_1, c_2)] = \text{pre}_r([c_2], [c_1] \setminus [c_2])
\] (5.20)

Let \( I \) denote the index set for the set \( \{x_1, \ldots, x_n\} \) of real variables and \( I_C \subseteq I \) an index set for the clock variables in \( \{x_1, \ldots, x_n\} \). Let \( c_1 \) be formulated as \( \sum_{i \in I} a_i x_i \sim_1 b \) with \( \sim_1 \in \{<, =\} \), and \( c_2 \) be formulated as \( \sum_{i \in I} d_i x_i \sim_2 e \) with \( \sim_2 \in \{<, =\} \). Note that restricting \( c_1 \) and \( c_2 \) such that the operator is in \( \{<, =\} \) does not destroy generality.

\[
\text{pre}_r([c_2], [c_1] \setminus [c_2]) \text{ is equal to}
\]

\[
\{ \rho \in [c_2] | \exists \delta \geq 0. \rho[+\delta] \in [c_1] \setminus [c_2] \land \forall \delta'. (0 < \delta' < \delta \Rightarrow \rho[+\delta'] \in [c_2] \cup ([c_1] \setminus [c_2])) \}
\] (5.21)

From the fact that \( c_1 \) is an upper bound, it follows that \( \rho \in [c_1] \Rightarrow \forall \delta > 0. \rho[-\delta] \in [c_1] \).

Therefore for any \( c_1, c_2, \rho, \delta > 0 \),

\[
\rho[+\delta] \in [c_1] \setminus [c_2] \Rightarrow \forall \delta'. (0 < \delta' < \delta \Rightarrow \rho[+\delta'] \in [c_1]) \Rightarrow \\
\forall \delta'. (0 < \delta' < \delta \Rightarrow \rho[+\delta'] \in [c_2] \cup ([c_1] \setminus [c_2]))
\]

which means that equation 5.21 can be simplified to

\[
\{ \rho \in [c_2] | \exists \delta > 0. \rho[+\delta] \in [c_1] \setminus [c_2] \}
\]

which is equal to

\[
[c_2] \cap \{ \rho | \exists \delta > 0. \rho[+\delta] \in [c_1] \setminus [c_2] \}
\] (5.22)

From equation 5.20 we can read that the right-hand side of the intersection should characterize \([propT(c_1, c_2)]\). Given the formulations of \( c_1 \) and \( c_2 \) above, the right-hand side of the intersection is equal to the set of valuations that satisfy

\[
\exists \delta > 0. \sum_{i \in I_C} a_i (x_i + \delta) + \sum_{i \in I \setminus I_C} a_i x_i \sim_1 b \land \text{not} (\sum_{i \in I_C} d_i (x_i + \delta) + \sum_{i \in I \setminus I_C} d_i x_i \sim_2 e)
\]

Solving \( \delta \) in this equation results in

\[
\sum_{j \in I}(a_j) \sum_{i \in I_C} (d_i - d_j) \sum_{i \in I_C} a_i x_j \sim b \sum_{i \in I_C} d_i - e \sum_{i \in I_C} a_i
\] (5.23)
where $\sim = \begin{cases} \leq & \text{if } \sim_1=\leq \text{ and } \sim_2=\leq \\ < & \text{otherwise} \end{cases}$. Here we use that fact that because $c_1$ is an upper bound, $\sum_{i \in I_c} a_i > 0$. Thus, equation 5.23 gives the needed explicit definition of $propT$.

5.5 Reduction

Now that the propagation operations have been defined we focus on the splitting operation. The splitting itself is a trivial operation (extending the tree), but what is needed is a mechanism to avoid the useless splitting of zones. If a constraint is satisfied by all valuations in a zone or by none of the valuations in a zone, then splitting that zone with that constraint is not needed. We will refer to this mechanism of avoiding redundant splits as reduction. It is applied whenever a region split is considered. The reduction procedure integrates two tests, one checking whether or not $Z \Rightarrow c$, and one checking $Z \Rightarrow \neg c$. If either condition is true, then no splitting is performed. Note that if reduction would not be performed then splits would keep propagating through the symbolic state space, endlessly splitting regions with redundant constraints.

Reduction is a critical aspect of our model checking approach, since it is by far the most time consuming operation. As was shown in the previous section, propagation is a computationally simple operation. It is the reduction procedure that determines the efficiency of a splitting tree state space presentation and of the model checkers using it.

A reduction procedure is needed for each data type for which constraints occur in the state space. Being the most important data type for real-time model checking, we first focus on reduction procedures for real variables. Two reduction algorithms are discussed, one for difference constraints and one for linear constraints. After that, also reduction for enumerated variables is discussed.

We require a reduction procedure $\text{reduce}(c, n)$ that takes a constraint $c$ and a region representation $n$, and which returns

- \text{sat} if all regions in $\mathcal{F}[n]$ satisfy $c$;
- \text{dissat} if all regions $\mathcal{F}[n]$ do not satisfy $c$;
- \text{split} if some regions in $\mathcal{F}[n]$ satisfy $c$ and some others do not satisfy $c$.

To arrive at simpler descriptions, the specific reduction algorithms presented below assume that a zone, which is a conjunction of single constraints, is given as a list of single constraints.

Reduction procedures have to be defined for each specific data type. These are defined as procedures that take a list of constraints $\text{conslist}$ of some data type and a single constraint $c$ of the same data type, and which return either sat, dissat or split. The main reduction procedure then becomes a dispatching procedure that selects the appropriate type-specific reduction procedure. See figure 5.10.

Note that in an implementation, the process of collecting constraints can be integrated in the reduction procedure, thus collecting the constraints on-the-fly.

5.5.1 Reduction for difference constraints

Reduction on convex zones defined by difference constraints can be performed by a variation of a shortest path algorithm. A zone is defined by a conjunction of difference constraints. Without loss of generality we assume that constraints are of the form $X - Y \sim c$ or $X \sim c$, where $X$ and $Y$ are variables, $c$ is a constant, and $\sim \in \{<, \leq\}$.
Reduction

\[ \text{reduce}(c, n) : \]
\[ \text{let } \text{conslist} = \{ c' \in S[n] \mid \text{type}(c') = \text{type}(c) \} \]
\[ \text{cases } \text{sort}(c) \]
\[ \text{case } \text{type A:} \]
\[ \text{return } \text{reduction-for-type-A}(c, \text{conslist}) \]
\[ \text{case } \text{type B:} \]
\[ \text{return } \text{reduction-for-type-B}(c, \text{conslist}) \]
\[ \ldots \]

Figure 5.10: the reduction procedure

For the moment we restrict ourselves to strict constraints. A set of difference constraints can be expressed in a graph in which the vertices represent variables and edges represent difference constraints on these variables. We will refer to such graphs as reduction graphs. For example, the presence of an edge \( x \rightarrow y \) in the graph corresponds to the fact that \( x - y < 3 \) is one of the constraints defining the zone. An additional node, referred to as \( 0 \), is introduced to accommodate constraints on single variables. An edge \( x \rightarrow 0 \) represents the constraint \( x < 3 \).

As an example, figure 5.11 shows the graph that corresponds to the constraints \( x - z < 1 \), \( z - x < 0 \), \( z - y < 1 \) and \( x < 3 \).

Reduction is based on the fact that some constraint \( x - y < c \) is implied by the set of constraints expressed in the graph if there is a path from \( x \) to \( y \) with an accumulated weight less than \( c \). Likewise, the negation of a constraint \( x - y < c \) is implied by the set of constraints if there is a path from \( y \) to \( x \) with accumulated weight smaller or equal to \( -c \). For the example of figure 5.11, consider a new constraint \( x - y < 3 \). There is a path in the graph from \( x \) to \( y \) with an accumulated weight of 2, which means that the constraint \( x - y < 2 \) is implied by the set of constraints represented by the graph. As a consequence the new constraint \( x - y < 3 \) is always true in the zone defined by the given four constraints. Similarly, the constraint \( z > 5 \) is always false because there is a path from \( z \) to \( 0 \) with weight 3 (which is smaller than 5). In both cases, performing the split according to the new constraint would not be performed.

This idea can be extended to also deal with non-strict constraints, by also labelling edges with an indication of strictness. Besides a weight, each edge can be labelled with an element from \( \{<, \leq\} \), indicating the strictness or non-strictness of the constraint. We then get:

- \( Z \Rightarrow x - y < c \) if there exists a path from \( x \) to \( y \) with accumulated weight smaller than \( c \), or if a path exists with only \( '<\)\'-edges, with accumulated weight equal to \( c \).
- \( Z \Rightarrow x - y \leq c \) if a path exists from \( x \) to \( y \) with accumulated weight smaller or equal to \( c \).
For $Z \Rightarrow \neg(x-y < c)$ and $Z \Rightarrow \neg(x-y \leq c)$ similar conditions exist.

We could store a reduction graph for each zone in the system, but that would involve large amounts of storage. One could even choose to build and store optimized, canonical versions of reduction graphs. In fact, one then ends up with difference bound matrices [47], which are applied in successful real-time model checkers like Uppaal. These are also based on the same principle of representing sets of constraints. However, the approach chosen here intends to avoid the storage and normalization involved with canonical representations by only using the splitting tree representation of zones.

In fact, we do not explicitly build reduction graphs at all. The reduction graph of a zone is implicitly encoded in the constraints that define the zone and which are represented in the splitting tree. In other words, the shortest path algorithm is performed directly on the splitting tree representation of the zone — more precisely, on the part of the splitting tree that is traversed from the leaf node to the top node.

Figure 5.12 shows the reduction algorithm for difference constraints. It abstracts from from the actual collection of constraints that is done by traversing up the splitting tree. Instead it assumes the existence of a list, called conslist, of constraints that is obtained by traversing from the leaf node up the tree, selecting the constraints on real-valued variables, and negating constraints when necessary. Note that conslist has to be collected each time reduction is performed. Without loss of generality, all constraints are assumed to be of the form $A-B \sim c$, where $A$ and $B$ are variables or the special $0$ variable, $c$ is a constant, and $\sim \in \{<, \leq\}$. The $0$ variable is used to represent single variable constraints. For example, a constraint $x < 3$ is represented as $x - 0 < 3$. The reduction procedure assumes that conslist defines a feasible region.

Here, the reduction algorithm is given for a constraint $X-Y < z$ (or $Y-X \geq -z$). The reduction algorithm for $X-Y \leq z$ (or $Y-X > -z$) is analogous. $\text{Vars}$ denotes the set of real-valued variables. The algorithm takes a constraint $X-Y < z$ and the mentioned list of constraints, and returns $\text{sat}$ if the new constraint is implied by the set of constraints, $\text{dissat}$ if the negation of the new constraint is implied by the set of constraints, and $\text{split}$ if the new constraint represents a true split of the zone defined by the list of constraints.

The algorithm is a combination of two bidirectional shortest path algorithms. The first is aimed at finding evidence for $X-Y < z$ (the sat case), while the second is aimed at finding evidence for $X-Y \geq z$ (the dissat case). We can apply bidirectional shortest path algorithms because the begin- and end-points are known. In that case a bidirectional search is more efficient. Lines 8 – 19 deal with the first shortest path search, while lines 20 – 27 deal with the second short path search.

For each variable $A$, it maintains the values $\delta_X(A), \delta_Y(A), \Delta_X(A), \Delta_Y(A), \alpha_X(A)$ and $\alpha_Y(A)$, which represent the current state of the algorithm. The values have the following meaning

- $\delta_X(A)$ holds the smallest known upper bound on the difference between $X$ and $A$
- $\delta_Y(A)$ holds the smallest known upper bound on the difference between $A$ and $Y$.
- $\alpha_X(A)$ is a boolean that equals $\text{true}$ iff $\delta_X(A)$ is to be interpreted as a strict constraint.
- $\alpha_Y(A)$ is a boolean that equals $\text{true}$ iff $\delta_Y(A)$ is to be interpreted as a strict constraint.
- $\Delta_X(A)$ holds the largest known lower bound on the difference between $X$ and $A$
- $\Delta_Y(A)$ holds the largest known lower bound on the difference between $A$ and $Y$.

Note that, as will be become clear from the algorithm, for the largest known lower bounds ($\Delta_X(A)$ and $\Delta_Y(A)$), strictness is not relevant. For notational convenience $\infty$ and $-\infty$
reduce_difference_constraint("X – Y < z"); conslist): 
1. for each A ∈ Vars do δX(A) := −∞; δY(A) := −∞; ΔX(A) := −∞; ΔY(A) := −∞ 
2. δX(X) := 0; δY(Y) := 0; ΔX(X) := 0; ΔY(Y) := 0; 
3. αX(X) := FALSE; αY(Y) := FALSE 
4. change := TRUE 
5. while change do 
6. change := FALSE 
7. for each "A − B ∼ c" ∈ conslist do 
8. if δX(A) + c < δY(B) 
9. or (δX(A) + c = δY(B) and ¬αX(B) and (αX(A) or ∼="<")) then 
10. δX(B) := δX(A) + c; αX(B) := αX(A) ∨ (∼="<"); change := TRUE 
11. if δX(B) + δY(B) < z 
12. or (δX(B) + δY(B) = z and (αX(B) or αY(B))) then 
13. return SAT 
14. if δY(B) + c < δY(A) 
15. or (δY(B) + c = δY(A) and ¬αY(A) and (αY(B) or ∼="<")) then 
16. δY(A) := δY(B) + c; αY(A) := αY(B) ∨ (∼="<"); change := TRUE 
17. if δX(A) + δY(A) < z 
18. or (δX(A) + δY(A) = z and (αX(A) or αY(A))) then 
19. return SAT 
20. if ΔX(B) − c > ΔX(A) then 
21. ΔX(A) := ΔX(B) − c; change := TRUE 
22. if ΔX(A) + ΔY(A) ≥ c then 
23. return DISSAT 
24. if ΔY(A) − c > ΔY(B) then 
25. ΔY(B) := ΔY(A) − c; change := TRUE 
26. if ΔX(B) + ΔY(B) ≥ c then 
27. return DISSAT 
28. return SPLIT 

Figure 5.12: reduction for difference constraints

are used to denote imaginary values that are respectively larger and smaller than any other value that can occur.

The algorithm repeatedly inspects the list of constraints, strengthening the above defined attributes of variables whenever possible. If at some point from the values of the attributes a constraint can be inferred that implies the constraint X – Y < z, then SAT is returned (lines 13 and 19). Similarly, if the negation of X – Y < z is implied by the attribute values, then DISSAT is returned (lines 23 and 27). If at some point the attribute values cannot be strengthened any more, then neither X – Y < z nor its negation are implied by the set of constraints in the list. In that case SPLIT has to be returned (line 28). The change variable is used to signal this situation.

To show correctness, we use the fact that at all times, for each variable A, including 0
Consider the case of

\[
\begin{align*}
X - A &\sim_1 \delta_X(A), \\
A - Y &\sim_2 \delta_Y(A), \\
X - A &\geq \Delta_X(A) \\
A - Y &\geq \Delta_Y(A)
\end{align*}
\]

which can be shown directly from the algorithm. After the initialization (lines 1-3) the property clearly holds. Also, it is easy to see that the updating of the \(\Delta, \delta\) and \(\alpha\) values (lines 10, 16, 21 and 25) retains the property. For example, for the updating of \(\delta_X(B)\) in line 10, combining the constraint \(A - B \sim c\) with what we know about \(\delta_X(A)\) in the first line of equation 5.24, we get that \(X - B \sim \delta_X(A) + c\), where \(\sim_x\) is strict only if either \(A - B \sim c\) is a strict constraint or if \(\alpha_X(A) = \text{true}\). Now if \(\delta_X(A) + c\) is smaller than \(\delta_X(B)\), then a smaller upper bound on \(X - B\) is found. \(\delta_X(B)\) then has to be updated to reflect the smaller bound (line 10). The condition in line 9 handles the case that there is strengthening by changing the upper bound from non-strict to strict. Note that the conditions starting in lines 8, 14, 20 and 24 ensure that the assignment is performed if and only if proper strengthening of the attributes can be done.

The equations in 5.24 justify the \texttt{sat} and \texttt{dissat} decisions in lines 13, 19, 23 and 27. For example, from 5.24 it follows that \(X - Y \sim \delta_X(A) + \delta_Y(A)\) where \(\sim_<\) if either \(\alpha_X(A) = \text{true}\) or \(\alpha_Y(A) = \text{true}\), and \(\sim_\leq\) otherwise. This immediately shows the correctness of the \texttt{sat} decision in line 12.

We still have to show that, if applicable, \texttt{sat} and \texttt{dissat} decisions are eventually made, under the assumption that the set of constraints in \texttt{conslist} defines a feasible zone. In case a difference constraint \(X - Y < z\) is implied by a feasible conjunction of difference constraints then it must be the case that there exist a sequence of different constraints from \texttt{conslist}, such that

\[
V_1 - V_2 \sim_1 c_1, \ V_2 - V_3 \sim_2 c_2, \ldots, \ V_n - V_{n+1} \sim_n c_n
\]

such that \(V_1 = X, \ V_{n+1} = Y\), and for which \(\sum_{1 \leq i \leq n} c_i < z\), or \(\sum_{1 \leq i \leq n} c_i = z\) and at least one of the constraints is strict. From the algorithm it follows that in this case a \texttt{sat} decision will be made (within at most \((n+1)/2\) iterations). This means that whenever \(\subseteq \bigland \texttt{conslist} \subseteq [X - Y < z]\), a \texttt{sat} is decision will ultimately be made. A similar argument holds for \texttt{dissat} decisions.

Finally, it remains to show termination and correctness of the \texttt{split} decision. The \texttt{change} variable ensures that the algorithm terminates with a \texttt{RSPLIT} decision as soon as in an iteration no stronger bound could be derived. Thus to prove termination we have to show that the bounds in \(\delta_X(), \delta_Y(), \Delta_X()\) and \(\Delta_Y()\) can not be strengthened forever. Consider the case of \(\delta_X()\). Let \(N\) denote the number of variables. Infinite strengthening of bounds in \(\delta_X(V)\) would have to mean that for any \(n > 0\) during the algorithm some value \(\delta_X(V)\) is the result of \(n\) strengthening updates, i.e. there is a sequence of constraints

\[
X - V_1 \sim_1 c_1, \ V_1 - V_2 \sim_2 c_2, \ldots, \ V_{n-1} - V_n \sim_n c_n
\]

such that \(V_n = V\). Thus, each constraint \(V_i - V_{i+1} \sim_i c_i\) strengthens the bound on \(X - V_{i+1}\). Consider the case that \(n > N\). Because there is only a finite amount of variables, it must be that for some \(i,j\) with \(1 < i < j\), \(V_i = V_j\). This means that \(\sum_{1 \leq k \leq i} c_k < \sum_{1 \leq k \leq j} c_k\).
However, then \( \sum_{i \leq k \leq j} c_k < 0 \) and thus that \( V_i < V_j \), which would mean that the zone is infeasible. This reasoning applies to each of the four bounds. Therefore, if the zone defined by the set of constraints is feasible, then the algorithm will terminate. A \texttt{SPLIT} decision is returned in this case, which is correct because apparently no \texttt{SAT} or \texttt{DISSAT} decision could be made.

For infeasible zones, the reduction procedure may not terminate. The restriction to feasible zones is not a problem since the reduction procedures guarantees that infeasible zones cannot occur. Also, it is not hard to extend the procedure to recognize infeasible zones. Infeasible zones are represented by graphs that have negative cycles.

Note that the shortest path search for deciding \texttt{DISSAT} is simpler because in that case strictness is not relevant. The reason for this is that in this case evidence for the non-strict constraint \( X - Y \geq z \) has to be found. For the \texttt{SAT} case strictness is relevant because finding evidence for \( X - Y < z \) will do, while evidence for \( X - Y \leq z \) is not sufficient. A second reduction procedure is needed for dealing with \( X - Y \leq z \), which is analogous to the one for \( X - Y < z \). For this procedure, strictness is relevant for the \texttt{DISSAT} case. Note that the constraint list in the algorithm presented here is not constructed explicitly. An implementation of the algorithm can directly operate on the splitting tree representation.

A different way to implement the algorithm would be to build the graph that represents the constraints explicitly and then perform the shortest path search on that graph. The advantage would be that the search becomes more efficient. However, additional processing is needed to construct the graph each time reduction is to be performed.

For difference constraints on integer variables, the reduction procedure is the same except that there is no need to deal with the difference between strict and non-strict constraints, since each non-strict constraint on integer variables can be transformed to a strict constraint.

### 5.5.2 Reduction in case of linear constraints

For linear constraints the reduction problem is equivalent to a linear programming problem. Let \( C \) be the set of constraints that define a zone, and \( c \) a new constraint, then the reduction operation can be done by checking

1. whether or not \( \Lambda(C \cup \{c\}) \) is empty, and
2. whether or not \( \Lambda(C \cup \{-c\}) \) is empty

If both are non-empty, then the new constraint \( c \) represents a true split. Both problems are linear programming problems which can be solved using standard linear programming algorithms, like the simplex algorithm.

Existing linear programming solvers tend to focus on solving huge problems efficiently. What is needed for our reduction procedure is a solver that efficiently solves very large amounts of relatively small problems. It is therefore worthwhile to implement a dedicated linear programming solver for the reduction procedure for linear constraints. The details of the implementation of the linear-programming based reduction procedure used in our model checking implementation can be found in [15].

This approach could also be applied to linear constraints on integer variables. However, this would involve using an integer programming solver, which will in general demand much more computational resources than a linear programming solver. In our implementation, linear constraints on integer variables are not implemented. On integer variables, only difference constraints are allowed.
5.5.3 Reduction for enumerated types

For enumerated types, a reduction procedure can be defined that somewhat resembles the one for difference constraints. Let \( \text{Vars} \) denote the set of variables of some enumerated type, and \( \text{Values} \) the set of values for that type. Furthermore, let \( \text{conslist} \) denote a list of constraints for that same enumerated type. Three types of constraints are assumed, namely \( X = Y, X \neq Y \) and \( X \in \{v_1, \ldots, v_n\} \).

Figure 5.13 shows the reduction procedure for constraints of the form \( X \in \{v_1, \ldots, v_n\} \) while figure 5.14 shows the procedure for constraints \( X = Y \). Obviously the reduction procedure for \( X \neq Y \) is the same as that for \( X = Y \) (switching \( \text{sat} \) and \( \text{dissat} \) conclusions).

\[
\text{enum\_reduce\_1}("X \in \{v_1, \ldots, v_n\", \text{conslist})
\]

29. for each \( A \in \text{Vars} \) do \( \mathcal{E}(A) := \text{Values} \)
30. \( \text{cur} := \emptyset; \text{new} := \emptyset \)
31. for each ”\( A \in \{c_1, \ldots, c_n\}” \in \text{conslist} do
32. \( \mathcal{E}(A) := \mathcal{E}(A) \cap \{c_1, \ldots, c_n\}; \text{cur} := \text{cur} \cup \{A\} \)
33. while \( \text{cur} \neq \emptyset \) do
34. if \( \mathcal{E}(X) \cap \{v_1, \ldots, v_n\} = \emptyset \) then return \( \text{dissat} \)
35. if \( \mathcal{E}(X) \subseteq \{v_1, \ldots, v_n\} \) then return \( \text{sat} \)
36. for each ”\( A \sim B” \in \text{conslist} do
37. if ”\( =” \) then
38. if \( A \in \text{cur} \) and \( \mathcal{E}(B) \not\subseteq \mathcal{E}(A) \) then
39. \( \mathcal{E}(B) := \mathcal{E}(B) \cap \mathcal{E}(A); \text{new} := \text{new} \cup \{B\} \)
40. if \( B \in \text{cur} \) and \( \mathcal{E}(A) \not\subseteq \mathcal{E}(B) \) then
41. \( \mathcal{E}(A) := \mathcal{E}(A) \cap \mathcal{E}(B); \text{new} := \text{new} \cup \{A\} \)
42. if ”\( \neq” \) then
43. if \( A \in \text{cur} \) and \( \mathcal{E}(A) \subseteq \mathcal{E}(B) \) and \( |\mathcal{E}(A)| = 1 \) then
44. \( \mathcal{E}(B) := \mathcal{E}(B) \setminus \mathcal{E}(A); \text{new} := \text{new} \cup \{B\} \)
45. if \( B \in \text{cur} \) and \( \mathcal{E}(B) \subseteq \mathcal{E}(A) \) and \( |\mathcal{E}(B)| = 1 \) then
46. \( \mathcal{E}(A) := \mathcal{E}(A) \setminus \mathcal{E}(B); \text{new} := \text{new} \cup \{A\} \)
47. \( \text{cur} := \text{new}; \text{new} := \emptyset \)
48. return \( \text{split} \)

Figure 5.13: reduction for enumerated types: \( X = c \)

The reduction procedure for \( X \in \{v_1, \ldots, v_n\} \) maintains a function \( \mathcal{E} : \text{Vars} \to 2^\text{Values} \), which holds the current knowledge on the possible values for a variable \( A \), such that \( c \notin \mathcal{E}(A) \) implies that variable \( A \) cannot take on the value \( c \) in the zone defined by the set of constraints in \( \text{conslist} \). In other words, at all times

\[
\forall A \in \text{Vars} . \forall c \in \text{Values} . ((\exists \rho \in \text{sol} . \rho(A) = c) \Rightarrow c \in \mathcal{E}(A)) \tag{5.25}
\]

where \( \text{sol} = [\bigwedge_{\text{cons} \in \text{conslist}} \text{cons}] \).

The first two lines of the algorithm strengthen \( \mathcal{E}(A) \) for each \( A \in \text{Vars} \), such that it reflects all \( A \in \{v_1, \ldots, v_n\} \) constraints. The rest of the algorithm iteratively takes into account the \( A = B \) and \( A \neq B \) constraints. The sets \( \text{new} \) and \( \text{cur} \) keep track of changes in \( \mathcal{E} \) ensuring termination and efficient evaluation of the constraints.

The validity of equation 5.25 follows directly from the initialization of \( \mathcal{E} \) in lines 29 and 31 and the updating steps in lines 39, 41, 44 and 46. The\( \text{sat} \) and \( \text{dissat} \) decisions in lines 35 and 34 are justified by equation 5.25.
In case the algorithm departs from the while loop, then also the converse of of equation 5.25 holds, because in that case $E$ cannot be strengthened any further by any constraint in $\text{conslist}$. This is ensured by maintaining the variables $\text{new}$ and $\text{cur}$. Also because in that case the conditions in lines 34 and 35 are not satisfied we know that $E(X) \setminus \{v_1, \ldots, v_n\} \neq \emptyset$ and $E(X) \cap \{v_1, \ldots, v_n\} \neq \emptyset$. Together with the converse implication of equation 5.25 this means that $\exists \rho, \rho' \in \text{sol} \cdot \rho(X) \in \{v_1, \ldots, v_n\}$ and $\rho'(X) \notin \{v_1, \ldots, v_n\}$. This justifies the $\text{split}$ decision in the final line. Termination follows from the fact that the set of values is finite, that only correct decisions are made, and that non-termination would require an infinite amount of decisions.

Again, feasibility of $\text{sol}$ is assumed, but the procedure can easily be extended with a feasibility check — in fact, that would mean checking $E(X) \neq \emptyset$. Note that because $\text{Values}$ is finite and typically not very large, $E$ can be efficiently represented and manipulated (using bit-strings).

\begin{verbatim}
enum_reduce_2("X = Y", conslist):
 49. for each $A$ in vars do $E(A) := \text{values}$
 50. $eq_X := \{X\}$; $eq_Y := \{Y\}$; $cur := \{X, Y\}$; $new := \emptyset$
 51. for each "$A" \in \{c_1, \ldots, c_n\} \in \text{conslist} do$
 52. $E(A) := E(A) \cap \{c_1, \ldots, c_n\}$; $cur := cur \cup \{A\}$
 53. while $cur \neq \emptyset$
 54. if $eq_X \cap eq_Y \neq \emptyset$ then return SAT
 55. if $E(X) = E(Y)$ and $|E(X)| = 1$ then return SAT
 56. if $E(X) \cap E(Y) = \emptyset$ then return DISSAT
 57. for each "$A \sim B" \in \text{conslist} do$
 58. if $\sim = \"\neq\"$ then
    59. if $A \in cur$ then
      60. if $A \in eq_X$ and $B \notin eq_X$ then $eq_X := eq_X \cup \{B\}$; $new := new \cup \{B\}$
      61. if $A \in eq_Y$ and $B \notin eq_Y$ then $eq_Y := eq_Y \cup \{B\}$; $new := new \cup \{B\}$
      62. if $E(B) \subseteq E(A)$ then $E(B) := E(B) \cap E(A)$; $new := new \cup \{B\}$
      63. if $B \in cur$ then
        64. if $B \in eq_X$ and $A \notin eq_X$ then $eq_X := eq_X \cup \{A\}$; $new := new \cup \{A\}$
        65. if $B \in eq_Y$ and $A \notin eq_Y$ then $eq_Y := eq_Y \cup \{A\}$; $new := new \cup \{A\}$
        66. if $E(A) \subseteq E(B)$ then $E(A) := E(A) \cap E(B)$; $new := new \cup \{A\}$
      67. if $\sim = \"=\"$ then
        68. if $A \in cur$ then
          69. if $(A \in eq_X \land B \in eq_Y)$ or $(A \in eq_Y \land B \in eq_X)$ then return DISSAT
          70. if $|E(A)| = 1$ and $E(A) \subseteq E(B)$ then $E(B) := E(B) \setminus E(A)$; $new := new \cup \{B\}$
        71. if $B \in cur$ then
          72. if $(A \in eq_X \land B \in eq_Y)$ or $(A \in eq_Y \land B \in eq_X)$ then return DISSAT
          73. if $|E(B)| = 1$ and $E(B) \subseteq E(A)$ then $E(A) := E(A) \setminus E(B)$; $new := new \cup \{A\}$
        74. $cur := new$; $new := \emptyset$
 75. return $\text{split}$
\end{verbatim}

Figure 5.14: Reduction for enumerated types: $X = Y$

The reduction procedure for $X = Y$ is an extension of that for $X \in \{v_1, \ldots, v_n\}$. It builds a function $E$ in the same way. The decisions in lines 55 and 56 are justified by equation 5.25. In these lines, decisions are made based on the possible values of variables, as stored in $E$. 
Chapter 5 Symbolic state space representation

Additionally, \( X = Y \) can be implied by one or more \( A = B \) and \( A \neq B \) constraints in the constraint list. Two sets \( eq_X \) and \( eq_Y \) are maintained which hold the variables which are already known to be equal to \( X \) and \( Y \), respectively. Then whenever \( eq_X \) and \( eq_Y \) have a variable in common, we know that \( X = Y \) (line 54). Also, if a variable in \( eq_X \) is known to be unequal to a variable in \( eq_Y \) we know that \( X \neq Y \) (lines 69, 72).

At all times in the algorithm, all updates to \( eq_X \) and \( eq_Y \) ensure that \( A \in eq_X \Rightarrow sol \subseteq [X = A] \), which, together with equation 5.25, means that at all times

\[
\begin{align*}
(\mathcal{E}(X) = \mathcal{E}(Y) \wedge |\mathcal{E}(X)| = 1) \vee eq_X \cap eq_Y \neq \emptyset & \Rightarrow sol \subseteq [X = Y] \\
\mathcal{E}(X) \cap \mathcal{E}(Y) = \emptyset \vee (\exists "A \neq B" \in \text{conslist} . A \in eq_X \wedge B \in eq_Y) & \Rightarrow sol \subseteq [X \neq Y]
\end{align*}
\]

(5.26)

which justifies all sat and dissat decisions in the algorithm.

In case the while loop is departed then \( \mathcal{E}() \) can not be strengthened any more while \( eq_X \) and \( eq_Y \) cannot be extended any more. In that case the reverse implications in equation 5.26 and 5.25 can also be shown to hold. This justifies the split decision.

Note that a reduction procedure for an enumerated variables constraint only has to take into account those constraints that correspond to the same enumerated type. Finally, for enumerated types with only a few values, simpler reduction approaches can be found.

5.6 Extensions

The splitting tree representation and the associated reduction procedures can be extended in several ways, leading to potential improvements in the efficiency of the representation in a model checking tool.

5.6.1 Optimizing reduction

The approach discussed in section 5.5.2 applies a linear programming algorithm to each reduction problem. However, often reduction problems can be decided in a straightforward manner. This is for example the case if the new constraint is equal to one of the constraints that define the zone. By first checking for such cases, the linear programming-based reduction can in many cases be avoided.

In terms of implementation, such a check can be done very efficiently if the constraints in the state space are stored in such a way that each constraint is represented only exactly once. All occurrences of a constraint then refer to this single representation. Checking the equality of constraints can then be done in a kind of symbolic manner — by comparing their physical address.

This approach can be pushed further. If for a constraint \( \sum_{i \in I} a_i x_i < b \) there is a constraint that define the zone which has the same "body" \( \sum_{i \in I} a_i x_i \), then often the reduction problem can be decided immediately. Thus if there is a constraint \( \sum_{i \in J} a_i x_i \) in the set of constraints with \( c < b \), then sat can be decided. Likewise if there is a constraint \( \sum_{i \in J} a_i x_i > b \), with \( c > b \), then dissat can be decided.

By also using a unique representation for the body \( \sum_{i \in J} a_i x_i \) of constraints, this check can again be performed very efficiently. To realize unique representations for constraints and their bodies it is essential that constraints are put in some canonical form. Note that as a byproduct of using unique presentations for constraints and their bodies, data space
performance of the model checker improves because the number of unique constraints is a very small fraction of the total number of constraints that occur.

Possibly other, more involved optimizations can be found to avoid linear programming-based reduction in more cases.

5.6.2 Approximate reduction

The reduction procedures discussed until now produce exact results. An alternative approach is what one could call approximate reduction, which exploits the fact that unneeded splitting of regions does not invalidate the algorithm. It does introduce infeasible regions in the state space, but this is not a problem for the validity of the algorithm, since these regions will never become reachable. Typically, an approximate reduction algorithm only recognizes an unneeded split (i.e. returning \texttt{SAT} or \texttt{DISSAT}) in case this conclusion can be reached in a computationally simple way. Otherwise it will return \texttt{SPLIT}, regardless of the correctness of this decision. This will increase the amount of data resources required by the algorithm because the splitting trees will become larger. It is hoped that this is compensated by the gain in computation time obtained by avoiding an exact reduction procedure. Especially for linear constraints the potential reduction can be very large.

The simplest form of approximate reduction only recognizes redundancy of split in case the splitting constraint is one of the constraints that defines the region. An approximate reduction strategy for linear constraints that is a little more sophisticated returns \texttt{SAT} or \texttt{DISSAT} only if the constraint is implied by exactly one of the constraints that defines the zone. In fact, this means that one uses the optimized reduction approach of the previous subsection, but then without performing the actual exact reduction routine if optimized reduction fails. In experiments, this simple reduction strategy proved to be quite successful for a few verification problems. It seems worth investigating more involved optimized reduction strategies which could be more complex than what is suggested here, but still have a huge computational benefit over linear-programming based reduction.

5.6.3 Optimizing splitting trees

Obviously, the splitting trees representation is in many ways a representation that has inherent redundancy. This subsection discusses possibilities to optimize the splitting tree representation to make it more efficient. Such optimizations could potentially improve performance in terms of storage (more compact representations) or in terms of time (avoiding that the reduction procedure is given redundant information). Note that none of the extensions discussed in this section have as of yet been implemented in the model checkers that are discussed in chapter 6.4.

For many verification problems, the resulting splitting trees exhibit forms like depicted in the left diagram of figure 5.15. We use an asterisk to distinguish reachable nodes. Here, regions are repeatedly split, while each time only one of the two subregions turn out to be reachable. One way to such parts of splitting trees more optimally, is to merge the constraints into one splitting constraint, as depicted in the right-hand diagram. For constraints on real variables, this violates our requirement that zones are convex, which is essential for the propagate and reduction operations. However, this is only relevant for zones that correspond to reachable regions. If a region is not reachable, its zone will not be needed for any operation. In the figure, reachable zones are still convex. If a non-convex zone becomes reachable, then the required splitting can still be performed, unfolding the compressed splitting tree node.
Also, it will often be the case that constraints are made redundant by other constraints further down the tree. For example, if in figure 5.15 $c_2$ would be implied by $c_4$, then there is no harm in removing $c_2$. The problem is that to check such redundancy, one has to perform the reduction procedure. Thus checking for such redundancies each time the tree is extended, is too costly. So smarter strategies for checking for redundancies are needed (for example, checking for redundancy only once every $n$ tree updates). Also the approximate reduction procedures discussed above could be used to at least remove part of the redundant constraints.

The splitting trees are related to BDD-inspired canonical representations for difference constraints like clock decision diagrams (CDD’s) and difference decision diagrams (DDD’s). One of the fundamental concepts of these representations is the sharing of sub-trees, resulting in a directed graph rather than a tree. This results in representations for non-convex zones in the form of unions of convex zones. Since the ideas presented here all assume convexity, such an extension of splitting trees would not work straightforwardly. It is probably possible to extend the propagation and reduction operations to deal with unions of convex zones. However, the gain would not be as large as in symbolic forward or backward reachability algorithms. In the latter algorithms explored regions are only distinguished in two categories, namely reachable and non-reachable, which means that there is a lot of opportunity for unifying convex zones. However, in the partition refinement algorithm, regions are distinguished by bisimulation, which results in far fewer opportunities for unification of zones. There is one aspect of a partition refinement algorithm were a representation for unions of convex zones can be used, namely for regions that have been fully decided. On such regions, propagation and reduction will not applied any further. Also, stability with respect to outgoing edges is irrelevant for these regions. As a consequence, there are possibilities to "collapse" subtrees, by integrating equally decided regions. In extreme cases this could result in "empty" trees, because all regions corresponding to some location have been decided to the same value. However, it is hard to estimate whether such an extension would lead to a better model checking performance.

In our definition of splitting trees, all types of constraints are mixed in one tree. A different approach could be to use different splitting trees for different data types. This is illustrated in figure 5.16. The right side of that figure shows an alternative representation using multiple splitting trees for the splitting tree shown at the left. Note that only $Z_1$ and $Z_2$ are reachable. Thus, in such a multiple splitting tree approach, each zone is defined by a set of nodes, one from each splitting tree.
For several verification problems, the resulting splitting trees turn out to become extreme cases of the one shown at the left of figure 5.16. For such problems, multiple splitting trees are likely to dramatically improve the performance. On the other hand, consider the splitting tree of figure 5.17, which has 8 reachable zones. If these zones where to be represented by multiple splitting trees then this would, in the worst case, result in a total of 24 different zones. As a result of the separate trees the dependencies between constraints of different types are not exploited, resulting in a finer-grained splitting into zones. Again, without practical application, it hard to say whether or not multiple splitting trees are more efficient than single trees. Most likely, this will depend on the form of the verification problem.

Finally, another way of reducing the complexity of splitting trees as illustrated in the left side of figure 5.16, could be to reorder the constraints in the tree, as illustrated in figure 5.18. Heuristic algorithms could be introduced that try to optimize trees by changing the order of the splits. If applied in a smart way, such strategies could potentially decrease the size of the splitting trees, and thus improve time and data performance of a model checker.

5.6.4 Continuous variables

For many real-time model checking applications the ability to model so-called continuous variables is very useful [5]. Continuous variables are a generalization of clocks in the sense that continuous variables can have different rates than clocks, which have a fixed rate of 1. One can easily extend XTG with continuous variables — in the spirit of hybrid automata [6] — by introducing continuous variables and specifying for each location, the
progress rate of each continuous variable.

Given a model checker that can deal with linear constraints, extending it to also deal
with continuous variables is relatively simple (at least for the case that the rates of continuous variables are constant in a given location). The reduction procedure stays the same, since continuous variables with constant rates do not result in more complex constraints.

One would need a different, more general, propagation operation, which can be derived in the same way as the one for clock and discrete variables. Most of the theory remains the same, except that the $\rho[+\delta]$ operation obtains a different meaning. Let $V_R = \{x_1, \ldots, x_n\}$ be the set of real variables of such a system, which includes continuous variables, clock variables (having a rate 1) and discrete variables (having a rate 0). Let $I$ be the index set $\{1, \ldots, n\}$ for $\{x_1, \ldots, x_n\}$. Given a location $l$, $\gamma^l \in (\mathbb{R}^{\geq 0})^n$ assigns a rate to each variable $x_i$. Thus $\gamma^l_i$ holds the rate for variable $x_i$ that is associated with location $l$. Now $\rho[+\delta]$ is defined as follows:

$$\rho[+\delta](v) = \rho(v) + \gamma^l \ast \delta$$

Using this, a new propagation operation can be derived in a similar way as in section 5.4.2, which is parameterized on a rate assignment $\gamma^l$:

$$\sum_{j \in I_R} \left( (a_j \sum_{i \in I_R} \gamma^l_i d_i - d_j \sum_{i \in I_R} \gamma^l_i a_i) x_j \right) \sim \sum_{i \in I_R} \sum_{i \in I_R} \gamma^l_i d_i - e \sum_{i \in I_R} \gamma^l_i a_i$$

where $\sim = \begin{cases} < & \text{if } \sim_1 = < \text{ or } \sim_2 = < \\ \leq & \text{otherwise} \end{cases}$.

5.7 Discussion

The splitting trees are applied to serve two purposes. First, we exploit the fact that a splitting tree holds the splitting history of zones to realize the propagation operation in an efficient way. This idea is due to [98]. Note that splitting trees were already applied in [109], but in a different way. If a splitting history were not used, propagation would have to be an operation that has to be applied directly to zones, not using the knowledge of the splitting constraints.

An example of this traditional approach to partition refinement is [105], which is completely based on the DBM representation for zones. Propagation then becomes a more complex operation. In that case propagating the split of $Z_2$ into $Z_{2,1}$ and $Z_{2,2}$ back to $Z_1$ means that one has to compute DBM's for the zones $\text{pre}_r(Z_1, Z_{2,1})$ and $\text{pre}_r(Z_1, Z_{2,2})$ given DBM's for $Z_1$, $Z_{2,1}$ and $Z_{2,2}$. By using the splitting history, the propagation operation truly propagates the splitting, rather than the product of a splitting.
Note that in [4] propagation is done in very straightforward manner, not exploiting
the fact that for convex partitions \( \text{pre}_\tau(Z_1, Z_2, 1) \) and \( \text{pre}_\tau(Z_1, Z_2, 2) \) results in the de-
sired splitting. There, propagation of splits is done by computing \( \text{pre}_\tau(Z_1, Z_2, i) \) and
\( Z_1 \setminus \text{pre}_\tau(Z_1, Z_2, i) \) for \( i \in 1, 2 \).

The second reason for using splitting trees is that the splitting trees are used as the
means to represent zones. Unlike other representations for real-time state spaces, this
representation is not canonical — two equivalent zones will generally have different rep-
resentations. Also it is not optimal in the sense that some of the constraints that define
a zone will be redundant. The canonicity required for commonly used representations is
essential for implementing the needed operations in an efficient way. We do not aim at a
canonical representation because partition refinement approaches only require propagation
and split operations on zones. If more operations than propagation and reduction would
be needed on zones, then the use of splitting trees for the representation of zones would
probably not be very attractive.

Summarizing, a main gain of using splitting trees, comes from the fact that the storage
and normalization of canonical representations is avoided, while the required operations
can be efficiently implemented. Note that in [109] and [98], splitting trees were not used
for the representation of the zones.

There is a link between the reduction procedure for difference constraints and the
DBM representation for zones defined by difference constraints [47]. The idea behind
DBM’s to represent a set of difference constraints as weighted graphs is also the basis
for our reduction algorithm. However, we do not maintain such weighted graphs for our
zones, but only use the concept. Our reduction algorithm is similar to the normalization
algorithm for DBM’s, be it that for the reduction algorithm the shortest path computation
often does not have to be fully executed.

Several alternatives to DBM’s have been introduced (DDD’s [85], CDD’s [75]), inspired
by binary decision diagrams (BDD’s) [33]. The latter have been successfully used in non-
real-time model checking to represent state sets defined by boolean variables. CDD’s and
DDD’s could be seen as a more general form of splitting trees. Instead of trees, directed
acyclic graphs are used which allows the efficient representation of unions of convex zones.
As was discussed in section 5.6.3, such representations, which are based on sharing, do not
fit well to partition refinement approaches. Also, CDD’s and DDD’s build canonical models
to allow efficient implementation of the required operations. As was discussed already, for
partition refinement approaches, there is not a clear need for canonical representations.

For the most important type of constraints, namely constraints on real variables, two
reduction procedures were discussed. The first deals with difference constraints, while the
second is able to deal with linear constraints, but is obviously less efficient. Because of the
great difference in efficiency, it would be desirable to have two similar model checkers avail-
able, only differing in the complexity of the constraints that can be handled. Depending
on the verification problem at hand, one could use the appropriate model checker.

We focussed not only on state space representation for traditional timed automata
state spaces, which are typically defined by difference constraints on clock variables. Also,
the inclusion of non-clock real variables and enumerated variables was considered. Fur-
thermore, it was shown that the splitting tree approach is also valid for state spaces defined
by linear constraints.

The reduction procedure for difference constraints on real variables can easily be sim-
plified to deal with difference constraints on integer variables.

In general, the concept of splitting trees is independent of the kind of data that is
represented. This makes the incorporation of other data types in the input language of our model checker relatively easily.

The next chapter describes a model checking algorithm that builds on the splitting tree representation that is described here.
Chapter 6

Model checking TCTL

This chapter constructs a model checking algorithm, based on the material of the previous chapters. As will be shown, the algorithm for parametric model checking of TCTL specifications is an extension of the parametric reachability analysis algorithm presented in chapter 3. A big difference is the inclusion of mechanism to decide values, an efficient on-the-fly decision procedure for deciding TCTL properties using region graphs. This will be based on the TCTL evaluation approach presented in chapter 4. Furthermore, the state space representation approach presented in chapter 5 also brings extensions of the algorithm as it is presented in chapter 3. In particular, the splitting tree concept leads to a more optimal splitting approach.

6.1 Introduction

The model checking algorithm we are aiming at, builds on-the-fly, a partial stable region graph, and decides, also on-the-fly, values of regions of this partial region graph. In this way, unnecessary exploration of regions as well as unnecessary stabilization of regions is avoided, since the algorithm can restrict its focus to undecided, reachable regions.

On-the-fly construction of the partial region graph means the global expanded XTG is not constructed beforehand, but that global locations and edges are generated on demand. In this way, the generation of unreachable global control locations is avoided.

On-the-fly decision making requires the introduction of some additional structure based on the general solutions provided in chapter 4. Like in the algorithm of section 3.6, the idea is that regions in a region graph can be given (boolean) values that indicate whether regions satisfy certain relevant properties. In section 4.3.5 we saw that a decision procedure for CTL (and TCTL) is naturally specified in a hierarchical manner. This means that conceptually, the satisfaction of a property in an initial state follows from evaluating a large decision tree, possibly ranging over all subproperties of that property and all reachable states. Rather than performing this complete evaluation once enough information is available, intermediate results can be stored. This means that for relevant states in the state space, values of subproperties are maintained. This approach enables on-the-fly evaluation.

The advantage of on-the-fly evaluation lies in the fact that it is often not necessary to build the complete state space. If (abstract) state space construction and evaluation were separated, then one would always have to construct a complete model of the state space,
while for the particular verification problem at hand, parts of this state space might not be relevant. By maintaining intermediate decisions, on-the-fly evaluation becomes possible.

The parametric reachability algorithm presented in section 3.6 also used this idea. Regions were assigned values to administrate the state of the evaluation process (the value attribute). In that algorithm, the value of a region indicated whether or not the states represented by the region satisfy the reachability property that is under investigation. Because of the simplicity of the property, the property did not need to be decomposed into subproperties. The value of a reachability property follows directly from the characteristics of the current state and the value of the reachability property in its direct successors. For the verification of TCTL formulas, a similar approach is used, but a more complex notion of ‘value’ is needed, as well as a more complex mechanism to decide values. The groundwork for such a decision procedure was done in chapter 4.

The reachability algorithm of chapter 3 abstracted away from the concrete state space representation. In this chapter we will include the splitting tree representation for symbolic state spaces. As will be shown, this has some impact on how the refinement of the regions graphs is done.

The construction of a concrete on-the-fly decision approach — based on the results of chapter 4 — is the main topic of section 6.2. Using this, section 6.3 describes a model checking algorithm based on the partition refinement approach of chapter 3 and the splitting trees of chapter 5.

6.2 Evaluation of TCTL properties

This section derives a concrete evaluation approach for deciding TCTL properties on partial region graphs.

6.2.1 Partial region graphs

Because region graphs are constructed on-the-fly and decision making is also done on-the-fly, we have to formalize a notion of partially explored region graphs. A so-called partial region graph reflects the fact that most of the time only a subset of the reachable locations and edges have yet been explored. This means that for some of the regions of the partial region graph, there are still outgoing edges to be explored, some of them leading to yet unexplored destinations and others leading to existing regions of the partial region graph. Obviously, partial region graphs are more concrete instances of partial models.

**Definition 6.1 (partial region graph).** Let $M = \langle S, S_0, T \rangle$ be a timed transition system corresponding to an XTG $(V, V_C, V_P, \rho_0, L, l_0, I, E, \cup)$. Then a partial region graph corresponding to $M$ is a graph $RG^* = \langle R, R_0, E_R, R_U \rangle$, with

- $R \subseteq L \times \mathcal{P}(Env_V)$ is a set of (partially and fully explored) regions, such that $\forall r_1, r_2 \in R : r_1 \cap r_2 = \emptyset$ and for some $L' \subseteq L$, $\bigcup R = L' \times Env_V$.
- $R_0 = \{ r \in R \mid \exists s \in r . s \in S_0 \}$ defines the initial regions.
- $E_R \subseteq R \times (E \cup \{ \tau \}) \times R$, such that for each $r, r' \in R, e \in E$:

$$
\begin{align*}
    r \xrightarrow{e} E_R r' & \Rightarrow \exists s \in r . \exists s' \in r' . s \xrightarrow{e} T s' \\
    r \xrightarrow{\tau} E_R r' & \Rightarrow \exists s \in r . \exists s' \in r' . s \xrightarrow{\tau} T s'
\end{align*}
$$
• \( R_U \subseteq R \) denotes the set of partially explored regions, such that for any \( r \in R \) whenever

\[
(\exists s \in r \cdot \exists s' \in S : s \xrightarrow{\tau} s') \text{ and } \not\exists r' \in R : (s' \in r' \text{ and } r \xrightarrow{\tau_{E_R}} r') \text{ or }
(\exists s \in r \cdot \exists r' \in R : s \xrightarrow{\tau} r' \text{ and } \exists s' \in r' : s \xrightarrow{\tau} r' \text{ and } \not r \xrightarrow{\tau_{E_R}} r')
\]

then \( r \in R_U \).  \( \square \)

Thus, rather than defining a complete partition, \( R \) covers only part of the state space. \( R_U \) identifies the regions from which still edges could be missing. While for a non-partial region graph, the set of edges was induced by the partition, for a partial region graph we only require correctness of edges, not completeness. If a region represents states which have outgoing transitions that do not have a corresponding edge in \( R_E \), then this region is in \( R_U \). We do require that for each location, either all or none of the states are represented in the partial region graph.

Optimally deciding verification problems on-the-fly implies that decision making is based on partial region graphs. This is based on the idea that if some property holds for all possible (non-partial) region graphs that can evolve from a partial region graph, then the property can be decided for the partial region graph.

Partial region graphs are concrete instances of partial models (definition 4.8).

**Definition 6.2 (partial regions graphs are partial models).** The partial model defined by a partial region graph \( \langle R, R_0, E_R, R_U \rangle \) is the partial model \( \langle R, R_0, E_R, R_U \rangle \).

In the remainder of this chapter, we will not use the term partial model any more, but let a partial region graph implicitly denote a partial model.

### 6.2.2 Deciding TCTL using region graphs

In chapter 4 it was shown that to be able to evaluate TCTL properties, an XTG has to be extended. Our partial region graphs will originate from models that are the result of an extension with a divergence graph and a property variable graph. We want to be able to recognize the property variable graph aspect of partial region graphs. The definition below gives some auxiliary definitions that allow this.

**Definition 6.3 (Additional notation for partial region graphs).** Consider a partial region graph \( \langle R, R_0, E_R, R_U \rangle \) corresponding to an XTG that is extended with a property graph and the divergence graph as defined in chapter 4. Let \( M \) and \( M' \) be the timed transition systems corresponding to the original and the extended XTG, respectively. Then

- \( E_R^p \subseteq E_R \) denotes the subset of region graph edges that correspond to discrete edges that originate from the property variable graph
- \( E_R^d = E_R \setminus E_R^p \) denotes the remaining edges (i.e. discrete edges originating from the original XTG and the divergence graph, as well as all time edges).

Based on this, two successor functions are defined:

\[
\begin{align*}
\text{Suc}_P(r, u) &= \{ r' \in R \mid \exists (r, (\tau, u), r', \rho) \in E_R^p \} \\
\text{Suc}_S(r) &= \{ r' \in R \mid \exists (r, (\tau), r', \rho) \in E_R^d \text{ and } r \neq r' \}
\end{align*}
\]

Given a state \( \langle l, \rho \rangle \) of \( M \), let \( \text{ext}((l, \rho)) \) denote a state of \( M' \) such that \( \text{ext}((l, \rho)) = (l + l^d + l^u, \rho + \rho^d + \rho^u) \) (using the ” + ” notation of definition 2.22), where \( l^d \) and \( \rho^d \) denote the location and the initial valuation of the divergence graph, and \( l^u \) and \( \rho^u \) denote the initial location and initial valuation of the property variable graph.

Furthermore, let \( \text{reg}(s) \) denote the region that contains the state \( s \) of \( M' \).  \( \square \)
Chapter 6 Model checking TCTL

The $r \neq r'$ condition in the definition of $\text{Suc}_S$ comes from the fact that we would like to avoid the representation of self-looping time edges.

The theorem below expresses how partial region graphs can be used for verifying TCTL properties. It integrates the theorems of section 4.2, and theorem 3.3.

**Theorem 6.1 (deciding TCTL using region graphs based on extended XTG’s).**

Let $X = (V, V_C, V_P, \rho_0, I, E, U)$ be an XTG, $\psi$ a normalized TCTL property, and $F$ a fairness constraint for $X$. Let $X'$ be the extension of $X$ with the divergence graph and the property variable automaton corresponding to $\psi$, and let $F' = F \cup F_{\text{div}}$. Let $M$ and $M'$ denote the timed structures induced by $X$ and $X'$, respectively.

- Let $RG^* = (R, R_0, E_R, R_U)$ be a stable partial region graph corresponding to $M'$ that respects all atoms of $\psi$ and all predicates in $F$. Then for any state $s$ of $M$ for which $\text{reg}(\text{ext}(s)) \in R$,

$$\text{reg}(\text{ext}(s)) \models_{RG^*} A(\Gamma(\text{stripU}(\psi))) \Rightarrow s \models_{M} \psi$$

(6.1)

- Let $RG = (R, R_0, E_R, R_U)$ be a stable region graph corresponding to $M'$ that respects all atoms of $\psi$ and all predicates in $F$. Then for any state $s$ of $M$,

$$\text{reg}(\text{ext}(s)) \models_{RG} A(\Gamma(\text{stripU}(\psi))) \iff s \models_{M} \psi$$

(6.2)

**Proof.** We start with equation 6.2. From theorem 4.1 it follows that all fair runs of $M'$ are divergent. This means that theorems 4.2 and 4.3 can be combined to yield

$$\text{ext}(s) \models_{M} \Gamma(\text{stripU}(\psi)) \iff s \models_{M} \psi$$

Now since a stable region graph is a time-abstracting bisimulation abstract model of a timed structure (theorem 3.3), we know from theorem 4.5 that

$$\text{reg}(\text{ext}(s)) \models_{RG} A(\Gamma(\text{stripU}(\psi))) \iff \text{ext}(s) \models_{M} \Gamma(\text{stripU}(\psi))$$

which completes the proof of equation 6.2.

As for equation 6.1, if the left-hand side of the implication of equation 6.1 holds, then we know from the definition of partial models that for each completion $RG$ of $RG^*$, $\text{reg}(\text{ext}(s)) \models_{RG} A(\Gamma(\text{stripU}(\psi)))$ holds for each state $s$ that corresponds to the explored part of the state space ($\text{reg}(\text{ext}(s))$ exists in $RG^*$). Then using equation 6.2, the right-hand side of the implication follows.

6.2.3 Decision rules for TCTL

The hierarchical decision approach discussed in the introduction, implies that intermediate decision values for regions are maintained. The basic idea is that with each region, boolean values can be associated that express whether or not the states represented by that region satisfy relevant subproperties of the property under investigation. When deciding the satisfaction of a formula in a hierarchical manner, one could first decide all atomic properties in that formula, then decide all subproperties that have only atomic subproperties, and so on. In the end, one can then decide the formula itself. In fact, this idea was used in one of the first model checking solutions [39]. The approach here is more involved, but it is useful to take this idea as a starting point and consider differences.
First of all, the aim is to decide properties as soon as possible, based on a partial state space. Furthermore, a specific set of values is relevant for a given region. Remember that the property variable graph introduces a set of subspaces (see section 4.2.3) each of which is targeted at a part of the TCTL property. The set of relevant values depends on the subspace, in particular the location of the property variable graph in the region (since this location identifies the subspace). Another issue is that we want to propagate decision information in a backward manner. This has the advantage that there is no need to store outgoing stable edges (just like in the algorithm of section 3.6). To achieve this, it is necessary to maintain a few additional values for each temporal operator (for example $\text{Enext}(\phi_1 \text{EU} \phi_2)$, see below).

Instead of labelling each region with subproperties and negations of subproperties (as in [39]), we associate with each region a valuation that holds for each relevant subproperty a decision value that can be $T$ (the region satisfies the subproperty), $F$ (the region dissatisfies the subproperty) or $U$ (no decision could be made yet). Decision rules change valuations of regions, either from $U$ to $T$ or from $U$ to $F$. A decision rule is applied to a region and assigns — if appropriate — new values to this region. Thus, we have to find a set of decision rules such that when applying them repeatedly to yet undecided regions, ultimately all regions will be become decided to the correct values. Since the work in chapter 4 was aimed at (abstractly) defining such a set of decision rules, it should not be hard to arrive at such a set of concrete rules.

We first formalize the notion of values of regions of a region graph, as well as the notion of decision rules that can change these values. Definition 6.4 defines which values are required for a region in a region graph (the function $\text{props}$). Most of these values are just TCTL formulas that are subproperties of the TCTL property under investigation. Again $\text{Etrue}$ is used as a shorthand for $\text{TRUE}$ $\text{EU}$ $\text{TRUE}$. There are three additional constructs, namely $\text{Enext}(\phi_1 \text{EU} \phi_2)$, $\text{Anext}(\phi_1 \text{EU} \phi_2)$ and $\text{Enext}(\text{Etrue})$. The intuitive meaning given to $\text{Enext}(\phi_1 \text{EU} \phi_2)$ is that there is a successor region in which the states satisfy $\phi_1$ $\text{EU}$ $\phi_2$. The other two have analogous meanings.

We will use the term extended subproperty to denote these kind of values.

**Definition 6.4 (extended subproperties).** Extended subproperties are generated by the following syntax.

$$\psi ::= \phi \mid \text{Anext}(\phi \text{AU} \phi) \mid \text{Enext}(\phi \text{EU} \phi) \mid \text{Etrue} \mid \text{Enext}(\text{Etrue})$$

where $\phi$ denotes a normalized TCTL property.

Let $RG^* = (R, R_0, E_R, R_U)$ be as defined in theorem 6.1. Then, $\text{props}$ denotes a function over the set of regions $R$, that returns the set of extended subproperties that are relevant for that region. It is defined as follows ($l_\phi$ denoting the update graph location):

$$\text{props}(l_\phi + l, Z) = X \cup \{\text{Etrue}, \text{Enext}(\text{Etrue})\} \cup \{\text{Anext}(\phi_1 \text{AU} \phi_2) \mid \phi_1 \text{AU} \phi_2 \in X\} \cup \{\text{Enext}(\phi_1 \text{EU} \phi_2) \mid \phi_1 \text{EU} \phi_2 \in X\}$$

where

$$X = \{\phi' \in \text{subprop}(\phi) \mid \text{loc}(\phi') = l_\phi\}$$

$\text{props}(r)$ returns the set of extended subproperties that are relevant for a region $r$. For example, if $\phi = p_1 \text{AU} (p_2 \text{EU} (p_3 \text{EU} u. (p_4 \text{AU} p_5)))$, with $p_i$ atomic properties, then $\text{props}(l_\phi + \text{Etrue})$...
Definition 6.6 (valuations and decision rules). Each rule in that set, any application of that rule does not change the valuation. In the model checking algorithm, each type is applied in a specific manner. The four types are described below:

1. atomic property decisions, \( b \) and \( le \) from TCTL
2. successor-based decisions, \( Enext(\phi_1 \ EU \ \phi_2) \), \( Enext(Etrue) \), and \( Anext(\phi_1 \ AU \ \phi_2) \), as well as \( u. \ \phi_1 \).
3. local decisions, for boolean logic operators and part of the cases for temporal operators.

For decision making we have to identify the "parent" of an extended subproperty, which is the next-higher extended subproperty of that extended subproperty.

Definition 6.5 (parent values). Let \( par \) denote a function that given an extended sub-property, returns the set of parent subproperties.

\[
par(\phi) = \{ \phi' \mid \phi \in sb(\phi') \}
\]

where

\[
\begin{align*}
    sb(\phi_1 \ EU \ \phi_2) &= \{ Enext(\phi_1 \ EU \ \phi_2), Etrue, \phi_1, \phi_2 \} \\
    sb(\phi_1 \ AU \ \phi_2) &= \{ Anext(\phi_1 \ AU \ \phi_2), Etrue, \phi_1, \phi_2 \} \\
    sb(Etrue) &= \{ Enext(Etrue) \} \\
    sb(\phi_1 \lor \phi_2) &= \{ \phi_1, \phi_2 \} \\
    sb(\neg \phi_1) &= \{ \phi_1 \} \\
    sb(u. \ \phi_1) &= \{ \phi_1 \} \\
    sb(\phi_1) &= \emptyset \quad \text{for all other extended subproperties}
\end{align*}
\]

Definition 6.6 (valuations and decision rules). A valuation for a partial region graph \( RG^* \) associates with each region \( r \in R \) a function \( P \rightarrow \{ T, F, U \} \), where \( P = props(r) \). The initial valuation for a region graph is a valuation \( val \) for which \( val(r)(\phi) = U \) for each \( r \in R, \phi \in props(r) \).

The application of a decision rule transforms a valuation to a new one. A decision rule for \( RG^* \) is a combination of a condition and an update. A condition is a boolean formula in which atoms are conditions on \( RG^* \) and the valuation \( val \). An update of \( val \) specifies a transformation of \( val \) to a new valuation \( val' \). Updates are written as \( val(r)(\phi) := T \) or \( val(r)(\phi) := F \). The consequence of applying a decision rule is that only if its condition is satisfied, then the update is performed.

A valuation \( val \) over a set of nodes \( R \) is final with respect to a set of decision rules, if for each rule in that set, any application of that rule does not change the valuation.

Thus, a valuation for a partial region graph holds for each explored region, a value for each relevant extended subproperty. Note that the definition of the form of the rules is not made completely formal because the rules themselves will below be described more formally.

Four types of decisions can be distinguished. The distinction is based on the fact that in the model checking algorithm, each type is applied in a specific manner. The four types are described below:

1. atomic property decisions, \( b \) and \( le \) from TCTL
2. successor-based decisions, \( Enext(\phi_1 \ EU \ \phi_2) \), \( Enext(Etrue) \), and \( Anext(\phi_1 \ AU \ \phi_2) \), as well as \( u. \ \phi_1 \).
3. local decisions, for boolean logic operators and part of the cases for temporal operators.
4. **global decisions**, for part of the cases for temporal operators, involving investigation of strongly connected components.

For example, consider a region \( r \) and a property \( EG \, p \). This property has a subproperty \( p \) which can be decided immediately by inspecting the characteristics of \( r \). Thus if for example \( r \) satisfies the atomic property \( p \) then one assigns \( val(r)(p) := T \) (type 1 decision). Now if a region has a successor that satisfies \( EG \, p \) then we can decide that \( r \) satisfies \( Enext(EG \, p) \). In this case we can assign \( val(r)(\, Enext((EG \, p)) \) := \( T \) (type 2 decision). Furthermore if \( val(r)(p) = T \) and \( val(r)(\, Enext(EG \, p)) = T \), then we can also decide \( val(r)(EG \, p) := T \) (type 3 decision). On the other hand, if \( r \) is part of a fair cycle in which for all regions \( r' \) in that cycle we know that \( val(r)(p) = \mathcal{F} \), then we can decide \( val(r)(EG \, p) := \mathcal{F} \) (type 4 decision).

Figure 6.1 depicts the resulting decision rules. The four types of decision rules are reflected in the figure. Note that for the derived TCTL operators (\( EF \), \( EG \), \( AG \), \( AF \)), simplified decision rules can be derived. Also, more optimal decision rules can be derived, that require less storage of values. This discussed in section 6.2.4. The actual application of the decision rules is part of the algorithm that is discussed towards the end of this chapter.

The separate decision rules and values (\( Enext \) and \( Anext \)) for successor-based decisions are in principle not needed. If we would not introduce \( Enext \) and \( Anext \) values then the decision rules for the temporal operators would have to evaluate values of successor regions. The \( Enext \) and \( Anext \) values decouple the evaluation of the temporal operators from the propagation of successor-based decision information. The decision rules for the \( Enext \) and \( Anext \) values can simply be triggered by successors for which values have been set. This allows pure backward propagation of decision information. As a result, stable region graph edges only have to be stored with destination regions. Also, it makes the propagation of decision information a bit more efficient — be it at the cost of slightly more storage.

The following theorem states the correctness of the decision rules. This means that (1) decisions made by the decision rules are always correct, and that (2) the decision rules define an optimal decision procedure.

**Theorem 6.2 (deciding TCTL properties).** Let \( X, \, \psi, \, F, \, X', \, F', \, M' \) and \( RG^* = \langle R, R_0, E_{R}, R_U \rangle \) be defined as in theorem 6.1. Then the decision rules of figure 6.1 makes correct decisions, i.e. for any state \( s \) of \( M \) and each valuation \( val \) derived by applying the decision rules of figure 6.1 to the initial valuation, it always the case that

\[
val(\, reg(\, ext(s)\,))(\, stripU(\, \psi)\,) = T \quad \Rightarrow \quad s \models^\mathcal{F}_M \psi \tag{6.3}
\]

\[
val(\, reg(\, ext(s)\,))(\, stripU(\, \psi)\,) = \mathcal{F} \quad \Rightarrow \quad s \not\models^\mathcal{F}_M \psi \tag{6.4}
\]

Furthermore, the decision rules of figure 6.1 are optimal, i.e. the decision rules lead to a unique final valuation \( val_f \) for which for any state \( s \) of \( M \),

\[
val(\, reg(\, ext(s)\,))(\, stripU(\, \psi)\,) = T \quad \iff \quad s \models^\mathcal{F}_M \psi \tag{6.5}
\]

\[
val(\, reg(\, ext(s)\,))(\, stripU(\, \psi)\,) = \mathcal{F} \quad \iff \quad s \not\models^\mathcal{F}_M \psi \tag{6.6}
\]

**Proof.** Let \( \text{props}^*(r) = \{ \phi \in \text{props}(r) \mid \phi \text{ is a TCTL property} \} \), thus \( \text{props}^*(r) \) is equal to \( \text{props}(r) \) except that it does not contain the \( Enext \) and \( Anext \) constructs (but does include \( E_{true} \)). For notational convenience, in this proof \( \models \) is often written to denote
### Chapter 6 Model checking

#### Type 1 rules

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{L}_X<a href="l">l</a> )</td>
<td>( \text{val}(r)(l):= T )</td>
</tr>
<tr>
<td>( \neg \mathcal{L}_X<a href="l">l</a> )</td>
<td>( \text{val}(r)(l):= F )</td>
</tr>
<tr>
<td>( \exists \rho \in Z . \forall V \llbracket b \rrbracket(\rho) )</td>
<td>( \text{val}(r)(b):= T )</td>
</tr>
<tr>
<td>( \exists \rho \in Z . \neg \forall V \llbracket b \rrbracket(\rho) )</td>
<td>( \text{val}(r)(b):= F )</td>
</tr>
</tbody>
</table>

where \( r = (l, Z) \)

#### Type 2 rules

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists r' \in \text{Suc}_S(r) . \text{val}(r')(\phi_1) = T )</td>
<td>( \text{val}(r)(\text{Enext}(\phi_1)):= T )</td>
</tr>
<tr>
<td>( \exists r' \in \text{Suc}_S(r) . \text{val}(r')(\phi_1) = F )</td>
<td>( \text{val}(r)(\text{Anext}(\phi_1)):= F )</td>
</tr>
<tr>
<td>( \exists r' \in \text{Suc}_P(r, u) . \text{val}(r')(\phi_1) = T )</td>
<td>( \text{val}(r)(u, \phi_1):= T )</td>
</tr>
<tr>
<td>( \exists r' \in \text{Suc}_P(r, u) . \text{val}(r')(\phi_1) = F )</td>
<td>( \text{val}(r)(u, \phi_1):= F )</td>
</tr>
</tbody>
</table>

#### Type 3 rules

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{val}(r)(Etrue) = T \land (\text{val}(r)(\phi_2) = T \lor (\text{val}(r)(\phi_1) = T \land \text{val}(r)(\text{Enext}(\phi_1 \text{ EU } \phi_2)) = T)) )</td>
<td>( \text{val}(r)(\phi_1 \text{ EU } \phi_2):= T )</td>
</tr>
<tr>
<td>( \text{val}(r)(Etrue) = F \lor (\text{val}(r)(\phi_2) = F \land \text{val}(r)(\phi_1) = F) )</td>
<td>( \text{val}(r)(\phi_1 \text{ EU } \phi_2):= F )</td>
</tr>
<tr>
<td>( \text{val}(r)(Etrue) = F \lor \text{val}(r)(\phi_2) = T )</td>
<td>( \text{val}(r)(\phi_1 \text{ AU } \phi_2):= T )</td>
</tr>
<tr>
<td>( \text{val}(r)(Etrue) = T \land (\text{val}(r)(\phi_2) = F \land \text{val}(r)(\phi_1) = F) )</td>
<td>( \text{val}(r)(\phi_1 \text{ AU } \phi_2):= F )</td>
</tr>
<tr>
<td>( \text{val}(r)(\text{Enext}(Etrue)) = T \lor (\text{val}(r)(\phi_2) = T \land \text{val}(r)(\phi_1) = F) )</td>
<td>( \text{val}(r)(Etrue):= T )</td>
</tr>
<tr>
<td>( \text{val}(r)(\phi_1) = T \lor (\text{val}(r)(\phi_2) = T \land \text{val}(r)(\phi_1) = F) )</td>
<td>( \text{val}(r)(\phi_1 \lor \phi_2):= T )</td>
</tr>
<tr>
<td>( \text{val}(r)(\phi_1) = F )</td>
<td>( \text{val}(r)(\neg \phi_1):= T )</td>
</tr>
</tbody>
</table>

#### Type 4 rules

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall r \in C . (r \notin R_U \land \text{val}(r)(\phi_2) = F \land \forall r' \in \text{Suc}_S(r) \setminus C . \text{val}(r')(\phi_1 \text{ EU } \phi_2) = F) )</td>
<td>( \text{val}(r)(\phi_1 \text{ EU } \phi_2):= F )</td>
</tr>
<tr>
<td>( \neg \text{fair}(C) \land \forall r \in C . (r \notin R_U \land \text{val}(r)(\phi_1) = T \land \forall r' \in \text{Suc}_S(r) \setminus C . \text{val}(r')(\phi_1 \text{ AU } \phi_2) = T) )</td>
<td>( \text{val}(r)(\phi_1 \text{ AU } \phi_2):= T )</td>
</tr>
<tr>
<td>( \neg \text{fair}(C) \land \forall r \in C . (r \notin R_U \land \forall r' \in \text{Suc}_S(r) \setminus C . \text{val}(r')(Etrue) = F) )</td>
<td>( \text{val}(r)(Etrue):= F )</td>
</tr>
<tr>
<td>( \text{fair}(C) \land \forall r \in C . \text{val}(r)(\phi_2) = F )</td>
<td>( \text{val}(r)(\phi_1 \text{ AU } \phi_2):= F )</td>
</tr>
<tr>
<td>( \text{fair}(C) \land \forall r \in C . \text{val}(r)(Etrue) = T )</td>
<td>( \text{val}(r)(\phi_1 \text{ AU } \phi_2):= F )</td>
</tr>
</tbody>
</table>

where \( C \) denotes an arbitrary strongly connected subgraph in \( \mathcal{R} \) and \( \text{fair}(C) \) is written to denote \( \mathcal{A}(F')\)-\( \text{fair}(C) \)

Figure 6.1: decision rules for TCTL model checking (theorem 6.2)

\( \models_{\mathcal{A}(F')} \). Also, we will write \( \text{fair}(C) \) rather than the correct notation \( \mathcal{A}(F')\)-\( \text{fair}(C) \) for fairness properties on the abstract model.
We prove that for any \( r \in R \), any \( \phi \in \text{props}(r) \), and any valuation \( \text{val} \)

\[
\text{val}(r)(\phi) = T \Rightarrow r \models_{RG^*}^A(\Gamma(\phi)) \tag{6.7}
\]

\[
\text{val}(r)(\phi) = \mathcal{F} \Rightarrow r \not\models_{RG^*}^A(\Gamma(\phi)) \tag{6.8}
\]

and that for any final valuation \( \text{val}_f \),

\[
\text{val}_f(r)(\phi) = T \iff r \models_{RG^*}^A(\Gamma(\phi)) \tag{6.9}
\]

\[
\text{val}_f(r)(\phi) = \mathcal{F} \iff r \not\models_{RG^*}^A(\Gamma(\phi)) \tag{6.10}
\]

By substituting \( \text{reg}(\text{ext}(s)) \) for \( r \) and \( \text{strip}U(\psi) \) for \( \phi \) in equations 6.7 – 6.10 and combining the result with theorem 6.1, the equations of theorem 6.2 follow immediately. Note that substituting \( \text{strip}U(\psi) \) for \( \phi \) is allowed because from the definitions of \( \text{ext} \) and \( \text{reg} \) it follows that \( \text{strip}U(\psi) \in \text{props}(\text{reg}(\text{ext}(s))) \).

Equations 6.7 – 6.10 are proved by induction. We show that equations 6.7 – 6.10 hold for any \( \phi \) and any \( r \in R \) for which \( \phi \in \text{props}(r) \), assuming that for subproperties of \( \phi \), this is already the case. Thus concretely, we prove that

1. for the case that \( \phi \) is an atomic property, equations 6.7 – 6.10 always hold.
2. for the case that \( \phi = \text{Etrue} \), equations 6.7 – 6.10 always hold.
3. for the case that \( \phi \) is of the form \( \phi_1 \lor \phi_2 \) or \( \neg\phi_1 \) we prove that the equations hold, assuming that they hold already for \( \phi_1 \) and \( \phi_2 \).
4. for \( \phi = u. \phi_1 \) we prove that the equations hold, assuming that they hold for \( \phi_1 \).
5. for \( \phi = \phi_1 \text{EU} \phi_2 \) and \( \phi = \phi_1 \text{AU} \phi_2 \) we prove that the equations hold, assuming that they hold already for \( \phi_1 \) and \( \phi_2 \).

Given the above facts, it follows by induction that equations 6.7 – 6.10 hold for any \( r \in R \) and \( \phi \in \text{props}(r) \). The first two constitute the base case of the induction, while the others constitute the induction case.

We start with the atomic properties. Consider an atomic property \( p \) (i.e. a location expression or a boolean value expression). For an arbitrary \( \text{val} \), and \( r \in R \) with \( p \in \text{props}(r) \) it follows directly from the decision rules, the semantics of TCTL and the definition of \( \Gamma \) that

\[
\text{val}(r)(p) = T \Rightarrow \exists s \in r . s \models_{M'}^F \Gamma(p) \tag{6.11}
\]

Also, from the definition of abstract models follows that

\[
\exists s \in r . s \models_{M'}^F \Gamma(p) \iff r \models_{RG^*}^A(\Gamma(p))
\]

Combining the above equations proves equation 6.7 for the case that \( \phi \) is an atomic property. Equation 6.8 is proved in the same way. Equation 6.9 follows from the fact that for a final valuation also the reverse direction of the implication in equation 6.11 holds, since the rules for atomic properties can always be applied. Equation 6.10 follows in an analogous manner.

Now consider the case for \( \text{Etrue} \). From theorem 4.10 it follows that \([\text{Etrue}]_{RG^*}^{A(F')}\) is equal to

\[
\text{LFP}(X = \{ r \in R \mid \exists r' \in \text{succ}_{E_R}(r) . r' \in X \lor (\exists C \in \text{SCS} . r \in C \land \text{fair}(C))\})
\]

where again \( \text{SCS} \) denotes the set of strongly connected subgraphs in \( R \) and writing \( \text{LFP}(X = f(X)) \) to denote the least fixed point of \( X = f(X) \).
From the definition of the property graph it follows that each subspace has exactly the same fairness properties. We can therefore ignore property graph edges. This means that $\text{succ}_{E_R}(r)$ can be replaced by $\text{Suc}_{G}(r)$ resulting in the following decision rules.

- if $\exists r' \in \text{Suc}_{G}(r) \cdot \text{val}(r')\langle \text{Etrue}\rangle = \text{T}$ then $\text{val}(r)\langle \text{Etrue}\rangle := \text{T}$
- if $\exists C \in \text{SCS} \cdot \text{fair}(C)$ then $\forall r' \in C \cdot \text{val}(r')\langle \text{Etrue}\rangle := \text{T}$

Splitting the first decision rule, using the intermediate value $\text{Enext}(Etrue)$ results in the positive $\text{Etrue}$ rules of figure 6.1. Equations 6.7 and 6.9 then follow from the fact that these decision rules are correct and optimal, for $\text{Etrue}$. For negative $\text{Etrue}$ decisions (Equations 6.8 and 6.10), the rules can be derived in the same way. This is not shown here.

We continue with the induction cases. We start with $\phi_1 \lor \phi_2$:

$$\text{val}(r)\langle \phi_1 \lor \phi_2\rangle = \text{T} \Rightarrow \text{[decision rules]}$$

$$\exists r' \in \text{Suc}_{P}(r,u) \cdot \text{val}(r')\langle \phi_1\rangle = \text{T} \Rightarrow \text{[induction hypothesis]}$$

$$\exists r' \in \text{Suc}_{P}(r,u) \cdot r' \models A(\Gamma(\phi_1)) \Rightarrow \text{[D-CTL semantics, definitions of A and } \Gamma]$$

This proves equations 6.7 for $\phi_1 \lor \phi_2$. Equation 6.8 is proved in the same way. Equation 6.9 (and analogously equation 6.10) follows from the fact that for a final valuation also the reverse directions of the implications hold. The case for $\neg \phi_1$ is similar and therefore not shown here.

Now the case for $u, \phi_1$:

$$\exists r' = (l_\phi + l, Z) \in \text{Suc}_{P}(r,u) \cdot \text{val}(r')\langle \phi_1\rangle = \text{T} \Rightarrow \text{[induction hypothesis]}$$

$$\exists r' \in \text{Suc}_{P}(r,u) \cdot r' \models \langle A\text{upd} \circ \text{loc}(\phi_1) \rangle \land r' \models \langle \Gamma(\phi_1) \rangle$$

This proves equations 6.7 for the case that $\phi = u, \phi_1$. Equation 6.8 is proved in the same way. Equation 6.9 follows from the fact that for a final valuation also the reverse directions of the implications hold. Equation 6.10 follows in analogous manner.

This leaves us with the temporal operators. We start with positive decisions for $\phi_1 \text{ AU } \phi_2$.

From the definitions of $\Gamma$ and $A$ it follows that $\Gamma(\phi_1 \text{ AU } \phi_2)$ is equal to

$$\text{val}(r)\langle \phi_1 \text{ AU } \phi_2\rangle = \text{T} \Rightarrow \text{[definition of A} \Gamma\text{ and A]}$$

$$\exists r' \in \text{Suc}_{E_R}(r) \cdot r' \models \langle A\text{upd} \circ \text{loc}(\phi_1) \rangle \land r' \models \langle \Gamma(\phi_1) \rangle$$

From theorem 4.9 it follows that this is equal to

$$\exists r' \in \text{Suc}_{E_R}(r) \cdot r' \models \langle A\text{upd} \circ \text{loc}(\phi_1) \rangle \land r' \models \langle \Gamma(\phi_1) \rangle$$

Extracting $r \models \langle A\text{upd} \circ \text{loc}(\phi_2) \rangle$ from the fixed points, this becomes equivalent to

$$\text{val}(r)\langle \phi_1 \text{ AU } \phi_2\rangle = \text{T} \Rightarrow \text{[definition of A} \Gamma\text{ and A]}$$

$$\exists r' \in \text{Suc}_{E_R}(r) \cdot r' \models \langle A\text{upd} \circ \text{loc}(\phi_1) \rangle \land r' \models \langle \Gamma(\phi_1) \rangle$$

Equations 6.7 and 6.9 then follow from the fact that these decision rules are correct and optimal, for $\text{Etrue}$. For negative $\text{Etrue}$ decisions (Equations 6.8 and 6.10), the rules can be derived in the same way. This is not shown here.

We continue with the induction cases. We start with $\phi_1 \lor \phi_2$:
We know that \( \text{succ}_{E_S}(r') = \text{Suc}_S(r') \cup \bigcup_u \text{Suc}_P(r', u) \). Also, \( r \in A(G_{\text{upd} @ \text{loc}}(\phi_2)) \) and thus (because \( \{ r, r' \} \subseteq C \)), \( r' \in A(G_{\text{upd} @ \text{loc}}(\phi_2)) \) and thus \( \text{Suc}_S(r') \subseteq A(G_{\text{upd} @ \text{loc}}(\phi_2)) \). Also, because \( r \in A(G_{\text{upd} @ \text{loc}}(\phi_2)) \) it follows that \( \text{succ}_{E_S}(r') \setminus \text{Suc}_S(r') = \bigcup_u \text{Suc}_P(r', u) \subseteq A(\neg G_{\text{upd} @ \text{loc}}(\phi_2)) \). Using these facts it follows that equation 6.12 is equivalent to

\[
\left[ A(\neg G_{\text{upd} @ \text{loc}}(\phi_2)) \cup LFP(X = \{ r \in R \mid r \not\in E_{\text{true}} \lor r \models A(\Gamma(\phi_2)) \lor \exists C \in \text{SCS} . (r \in C \land \neg \text{fair}(C) \land \forall r' \in C . (r' \models A(\Gamma(\phi_1)) \land \neg r' \in R_{\text{loc}} \land \forall r'' \in \text{Suc}_S(r') \setminus C . r'' \in \text{T}_S) ) \right] 
\]

where the left-hand side of the union can be omitted because we are only interested in regions \( r \) for which \( \phi_1 \text{AU} \phi_2 \in \text{props}(r) \) and thus \( r \models G_{\text{upd} @ \text{loc}}(\phi_2) \).

This results in the following optimal decision rules for the set \( S_T \) of regions that satisfy \( A(\Gamma(\phi_1) \text{AU} \phi_2) \):

- if \( r \not\models E_{\text{true}} \lor r \models A(\Gamma(\phi_2)) \) then \( S_T := S_T \cup \{ r \} \)
- if \( \exists C \in \text{SCS} \) such that \( \neg (\text{fair}(C)) \land \forall r' \in C . r' \models A(\Gamma(\phi_1)) \land r' \not\in R_{\text{loc}} \land \forall r'' \in \text{Suc}_S(r') \setminus C . r'' \in S_T \) then \( S_T := S_T \cup C \)

Introducing the induction hypothesis this leads to the following decision rules for our context.

- if \( \text{val}(r)(E_{\text{true}}) = \emptyset \lor \text{val}(r)(\phi_2) = \emptyset \) then \( \text{val}(r)(\phi_1 \text{AU} \phi_2) := \emptyset \)
- if \( C \in \text{SCS} \) such that \( \neg (\text{fair}(C)) \land \forall r' \in C . \text{val}(r')(\phi_1) = \emptyset \land r' \not\in R_{\text{loc}} \land \forall r'' \in \text{Suc}_S(r') \setminus C . \text{val}(r'')(\phi_1 \text{AU} \phi_2) = \emptyset \) then \( \forall r \in C . \text{val}(r)(\phi_1 \text{AU} \phi_2) := \emptyset \)

These match exactly the rules for positive decisions for \( \phi_1 \text{AU} \phi_2 \) in figure 6.1. Equations 6.7 and 6.9 follow from the correctness and optimality of the above decision rules. Now for the negative decisions for \( \neg (\phi_1 \text{AU} \phi_2) \) (i.e. equations 6.8 and 6.10), we know from the definitions of \( \Gamma \) and \( A \) that \( \neg (\phi_1 \text{AU} \phi_2) \) is equal to

\[
\neg (A(\Gamma(\phi_1)) \text{AU} (A(\Gamma(\phi_2)) \lor A(\neg G_{\text{upd} @ \text{loc}}(\phi_2))))
\]

From theorem 4.9 it follows that this is equal to

\[
LFP(X = \{ r \in R \mid r \models E_{\text{true}} \land r \not\models A(\Gamma(\phi_2)) \land r \models A(\text{upd} @ \text{loc}(\phi_2)) \land (r \not\models A(\Gamma(\phi_1)) \lor \exists r' \in \text{succ}_{E_S}(r) . r' \in X) \lor \exists C \in \text{SCS} . (r \in C \land \text{fair}(C) \land \forall r' \in C . (r' \not\models A(\Gamma(\phi_2)) \land r \models A(\text{upd} @ \text{loc}(\phi_2)))) \})
\]

Extracting \( r \models A(G_{\text{upd} @ \text{loc}}(\phi_2)) \) from the fixed point and using the fact that \( \text{succ}_{E_S}(r') \setminus \text{Suc}_S(r') \subseteq A(\neg G_{\text{upd} @ \text{loc}}(\phi_2)) \), it follows that this is equivalent to

\[
\left[ A(G_{\text{upd} @ \text{loc}}(\phi_2)) \cup LFP(X = \{ r \in R \mid r \models E_{\text{true}} \land r \not\models A(\Gamma(\phi_2)) \land (r \not\models A(\Gamma(\phi_1)) \lor \exists r' \in \text{Suc}_S(r) . r' \in X) \lor \exists C \in \text{SCS} . (r \in C \land \text{fair}(C) \land \forall r' \in C . r' \not\models A(\Gamma(\phi_2)))) \})
\]

where the left-hand side of the intersection can be omitted because we are only interested in regions \( r \) for which \( \phi_1 \text{AU} \phi_2 \in \text{props}(r) \) and thus \( r \models G_{\text{upd} @ \text{loc}}(\phi_2) \). In the same way as above this leads to the following decision rules:
6.3 Model checking

• if $val(r)(Etrue) = \mathcal{T} \land val(r)(\phi_2) = \mathcal{F} \land (val(r)(\phi_1) = \mathcal{F} \lor \exists r' \in Suc_S(r). \, val(r')(\phi_1 \land \phi_2) = \mathcal{F})$
  then $val(r)(\phi_1 \land \phi_2) := \mathcal{F}$

• if $C \in SCS$ such that $fair(C) \land \forall r' \in C. \, val(r'')(\phi_2) = \mathcal{F}$
  then $\forall r \in C. \, val(r)(\phi_1 \land \phi_2) := \mathcal{F}$

The first rule can be split, using the $A_{next}$ value, resulting in the following rules:

• if $val(r)(Etrue) = \mathcal{T} \land val(r)(\phi_2) = \mathcal{F} \land (val(r)(\phi_1) = \mathcal{F} \lor \\\ldots$
  then $val(r)(\phi_1 \land \phi_2) := \mathcal{F}$

• if $\exists r' \in Suc_S(r). \, val(r')(\phi_1 \land \phi_2) = \mathcal{F}$
  then $val(r)(\phi_1 \land \phi_2) := \mathcal{F}$

• if $C \in SCS$ such that $fair(C) \land \forall r' \in C. \, val(r'')(\phi_2) = \mathcal{F}$
  then $\forall r \in C. \, val(r)(\phi_1 \land \phi_2) := \mathcal{F}$

These rules correspond to the negative $AU$ rules in figure 6.1, proving equations 6.8 and 6.10 for $AU$.

The cases for $\phi_1 \mathcal{E} \phi_2$ can be proved in the same way. This is not shown here. \hfill \Box

6.2.4 Optimizations

In the decision rules presented above, some optimizations were not done in order to avoid complex definitions and proofs. Some of these optimizations are briefly discussed here.

First of all, there is no real need to store the intermediate values for boolean operators. For example, for a property $AG(x = 1 \lor AF y = 1)$, we would store values for the atomic properties $x = 1$ and $y = 1$ for $AF y = 1$, and for the integral property itself. However, there is no need to store values for $x = 1 \lor AF y = 1$ since these can be directly derived from the values of $x = 1$ and $AF y = 1$. Thus, in an implementation of the decision rules, no values for boolean operators would have to be stored, reducing the amount of information that has to be stored for each region.

Furthermore, the value $EX Etrue$ is also not needed, since $EX Etrue$ and $Etrue$ will always have the same value. So instead of setting the value of $EX Etrue$, one can immediately set the value for $Etrue$.

6.3 Model checking

In section 3.6, a partition-refinement based algorithm for parametric reachability analysis was presented. Here, an algorithm is described that can be viewed as both an extension as well as a refinement of that algorithm. It is an extension because it targeted at checking TCTL properties rather than reachability properties. To that end, it integrates an on-the-fly decision procedure that is based on the decision rules of section 6.2. It can be seen as a refinement because it takes into account a specific symbolic state space representation, namely the splitting tree representation of chapter 5. This means that splitting and stabilization are given an interpretation in terms of the splitting, reduction, and propagation operations of chapter 5. Also, it induces a more efficient way of splitting, because splitting and reachability checking is done at a finer level of detail.

In the remainder of this section, let $X$ denote an XTG system, $\psi$ a normalized TCTL property, and $F$ a fairness constraint for $X$. Let $X'$ denote the extension of $X$ with the divergence graph and the property automaton corresponding to $\psi$, and let $F' = F \cup F^{div}$.
Let $M = \langle S, S_0, T \rangle$ denote the timed transition system corresponding to $X'$. $V$ denotes the set of variables in $X'$.

For the sake of clarity, it is assumed that guards, invariants, fairness predicates and atomic properties have the form of conjunctions over single constraints, represented as sets of single constraints. Extension to more complex constraints is straightforward, but would make the pseudocode descriptions much more complex.

In the pseudocode descriptions, some lines are enclosed in brackets. This concerns optimizations of the algorithm that are not taken into account in the semi-formal proof of section 6.3.5, because they complicate matters to a great extent.

### 6.3.1 Introduction to the algorithm

Similar to the algorithm of section 3.6, the algorithm integrates four activities:

1. Constructing the (extended) partial region graph by iteratively exploring new edges.
2. Stabilizing the partial region graph and ensuring that it respects fairness properties and atomic properties.
3. Keeping track which regions are known to be reachable
4. Deciding the values of regions, based on the decision procedure of section 6.2.

These procedures do not represent subsequent steps, but rather correspond to different aspects of the algorithm.

Like the algorithm of section 3.6, iteratively a region is selected that still requires further processing. However, in the parametric reachability algorithm, a newly encountered region was immediately split to ensure that all its subregions — reachable or not — satisfy invariants, respect atomic properties, and respect guards of outgoing edges. In terms of the original minimal model generation algorithm, one could say that the invariants, atomic properties and guards define the initial partition. As a consequence, unreachable regions will be split. Therefore, splitting is done in a more fine-grained manner, avoiding useless splitting of unreachable regions as much as possible. As a consequence, the main loop becomes more complex. With each region, a phase attribute is associated that keeps track of the progress in ensuring respect and stability.

The following categories of splitting are required:

1. splitting to ensure that regions respect the appropriate invariant;
2. splitting to ensure that regions respect the fairness properties;
3. splitting to ensure that regions respect the appropriate atomic properties;
4. splitting to ensure that regions respect the guards of edges (enabling stability);
5. splitting to ensure propagation stability with respect to outgoing time edges;
6. splitting to ensure propagation stability with respect to outgoing discrete edges.

In the algorithm, splitting of regions is always done in the order shown above, with the exception of the final two categories.

Consider for example a reachable region that satisfies the appropriate invariant and respects all fairness properties, and suppose that it is currently being split according to one of the atomic properties. If as a result of this splitting, subregions are created that are not reachable, then for these subregions splitting is not continued. The phase attribute of that region then remembers that this region still requires further splitting according to atomic properties. This information is needed in case the subregion becomes reachable again.
Now consider the integration of the TCTL decision procedure. First of all, the model checker will extend the XTG it is given as input with a property graph and the divergence graph. Furthermore, since the decision procedure is based on the decision rules of section 6.2, a valuation as defined in definition 6.6 will be maintained for the partial region graph. Integration of the local decision rules into the algorithm is relatively straightforward. Similar to the reachability algorithm, local decisions are taken as soon as possible, using the rules of figure 6.1 (type 1, 2 and 3). This means that the current valuation is always optimal with respect to local decisions. Whenever for a region new information has become available (caused by a type 1 decision), re-evaluation may be required (type 3 decision rules). This may initiate a sequence of successor-based decision making, which involves repeated application of type 2 and type 3 rules. This is referred to as value propagation, because typically decisions are propagated backwards through the graph. Note that the propagation of values has to be done after the propagation of splits, since values can only be decided on regions that are stable. Once the value of a region is decided, there is no use in splitting it again, so stabilization is only performed on undecided regions.

For global decisions a similar procedure could be adopted. This would mean that whenever an edge to an already explored region is added, one has to check whether the strongly connected component that region is part of satisfies one of the conditions of the evaluation rules of figure 6.1. Such a procedure would be optimal in the sense that it avoids unnecessary exploration and splitting, but it is expensive in terms of processing time. Checking for strongly connected components is expensive and will quite often not result in decisions because of the presence of not fully explored regions. Therefore, we will for now adopt a somewhat simplistic evaluation approach for global decisions similar to that of the parametric reachability algorithm and Sokolski’s [97] approach. This is based on the idea that global decisions are only taken once the state space is completely explored. In that case, all remaining undecided regions can be decided based on global criteria.

However, for our approach this does not have mean that global decisions are postponed until exploration is completed. As was discussed in section 4.2.3, the extension with a property graph results in a set of subspaces, each of which is associated with a subset of subproperties of the property to be checked. If for a subspace all regions are fully explored, then all undecided regions in that subspace can be decided. To that end we require that, over subspaces, exploration is done in depth-first order. Within a subspace we still do not impose any search order. To implement this idea, there is a stack of sets $S$, one for each subspace, rather than a single set $S$ as in the reachability algorithm. Finally, note that section 6.3.4 discusses an extension of the algorithm discussed here which does incorporate an on-the-fly cycle detection approach.

### 6.3.2 Algorithm infrastructure

This section considers the data structures and functions that are used in the algorithm.

First of all, regions now have a specific interpretation in terms of splitting trees.

$L_{\text{exp}}$: the set of (partially or fully) explored locations

$R$: The set of leaf regions that are present in the forest. $R$ forms a partition of the partial state space induced by $L_{\text{exp}}$.

$R^*$: The set of all leaf and internal regions that are present in the forest.

Note that $R$ and $R^*$ are not explicitly stored. They are only needed for descriptions, not for the algorithm itself. The forest is accessed using the functions of definition 5.5.
As in the reachability algorithm, \textit{unexp}, \textit{stab} and \textit{instab} attributes are used to store edges. However, the information stored for each edge is adapted to the need of the algorithm. For time edges, we need the separating constraint of the source and destination region. This is needed for the propagation operation (\textit{propT}, section 5.4.2). For discrete edges, we store the corresponding XTG edge, because it is needed for the \textit{propD} operation. This results in the following edge-related attributes of regions.

\textbf{stab} : \( R \rightarrow R^* \times (E \cup \tau) \times (\text{Bexpr}_V \cup \bot) \): \( \text{stab}(r) \) holds stable incoming edges of \( r \), such that for each \( \langle r', x, c \rangle \in \text{stab}(r) \), \( x = \tau \iff c \neq \bot \). Thus \( \langle r', e, \bot \rangle \in \text{stab}(r) \) means that there is a stable discrete edge \( r' \xrightarrow{e} r \), while \( \langle r', \tau, c \rangle \in \text{stab}(r) \) means that there is a stable time edge \( r' \xrightarrow{\tau} r \) where \( c \) is the separating constraint of \( r' \) and \( r \).

\textbf{instab} : \( R \rightarrow R^* \times (E \cup \tau) \times (\text{Bexpr}_V \cup \bot) \): \( \text{instab}(r) \) holds stable outgoing edges of \( r \), where for each \( \langle r', e, c \rangle \in \text{instab}(r) \), \( e = \tau \iff c \neq \bot \). Thus \( \langle r', e, \bot \rangle \in \text{instab}(r) \) means that there is a discrete stable edge \( r \xrightarrow{e} r' \), and \( \langle r', \tau, c \rangle \in \text{instab}(r) \) means that there is a stable time edge \( r \xrightarrow{\tau} r' \) where \( c \) is the separating constraint of \( r \) and \( r' \).

\textbf{unexp} : \( R \rightarrow E \): \( \text{unexp}(r) \) holds the unexplored outgoing edges of \( r \).

All explored edges are stored in either \textit{stab} or \textit{instab}, but an edge can never be stored in both \textit{stab} and \textit{instab}. The only exception is that self-looping time edges are not stored. These edges are kept implicit, since these are always present.

Furthermore, the upper bounds of regions are stored, for reasons that will become clear in the algorithm description.

\textbf{bound} : \( R \rightarrow (\text{Bexpr}_V \cup \bot) \): \( \text{bound}(r) \) holds the upper bound of \( r \), if it exists. Otherwise, \( \text{bound}(r) = \bot \).

The following variables are used to keep track of the exploration:

\textbf{S} : the set of reachable but potentially not yet stable regions that belong to the subspace that is currently under consideration.

\textbf{C} : the set of reachable stable regions of which all outgoing edges have been explored that belong to the subspace that is currently under consideration.

\textbf{StkS} : A stack of sets \( S \), each set corresponding to a different subspace.

\textbf{StkC} : A stack of sets \( C \), each set corresponding to a different subspace.

The boolean attributes below already appeared in the algorithm of section 3.6

\textit{reachable}(\( r \)) \( \in \mathbb{B} \): indicates whether or not \( r \) is reachable.

\textit{noaccess}(\( r \)) \( \in \mathbb{B} \): indicates whether or not \( r \) is inaccessible due to an invariant.

\textit{urgent}(\( r \)) \( \in \mathbb{B} \): indicates whether or not time can pass in \( r \).

To be able to efficiently check initial states, the following structures are used.

\textit{initial}(\( r \)) \( \in \mathbb{B} \): indicates whether or not \( r \) contains initial states.

\textit{init_count} : The number of undecided regions that have initial states. If \textit{init_count} turns zero, the model checking procedure is completed.

The following attributes, functions and variables are needed for the decision procedure.

\textit{val}(\( r \))(\( P \)) : the valuation as defined in definition 6.6.

\textit{fval}(\( r \)) \( \in \mathcal{P}(F') \): a valuation for the fairness predicates (\( F' \) being the relevant set of fairness predicates). It holds for each region, the set of fairness predicates it satisfies.

\textit{decided}(\( r \)) \( \in \mathbb{B} \): a function that indicates whether or not all subproperties that are relevant for \( r \) (i.e. \textit{props}(\( r \))) have been decided in \( r \). Thus

\[
\text{decided}(\( r \)) = \bigwedge_{P \in \text{props}(\( r \))} \text{val}(\( r \))(\( P \)) \neq \mathcal{U}
\]
\( \mathcal{V} \): holds the set of combinations of region and subproperty for which values still have to be propagated. Elements in \( \mathcal{V} \) are tuples \( \langle r, P \rangle \), where \( r \) is a region and \( P \in \text{props}(r) \). If a tuple \( \langle r, P \rangle \) is in \( \mathcal{V} \), this means that \( \text{val}(r)(P) \) has recently been set to \( T \) or \( F \), but that the effect of this decision has not yet been propagated to other values.

As was discussed already, we have to keep track of the progress of regions in terms of the kind of splitting that has already been done. To that end, a region has a \textit{phase} attribute.

\( \text{phase}(r) \in \{ \text{FAIR}, \text{ATOM}, \text{EDGES}, \text{GUARDS}, \text{TIME}, \text{ACTIVE} \} \): represents the phase of \( r \) with respect to the required splitting.

- \( \text{phase}(r) = \text{FAIR} \) indicates that for \( r \) still some splitting according to fairness predicates is required.
- \( \text{phase}(r) = \text{ATOM} \) indicates that \( r \) respects the fairness properties but that some splitting according to atomic properties may still be required.
- \( \text{phase}(r) = \text{EDGES} \) indicates that \( r \) respects the fairness and atomic properties but that the outgoing edges have not yet been loaded into \( \text{unexp} \).
- \( \text{phase}(r) = \text{GUARDS} \) indicates that fairness and atomic properties are respected and that all outgoing edges are in \( \text{unexp}(r) \), but that some splitting according to guards of outgoing edges may still be required. Thus there may still be not enabled outgoing edges in \( \text{unexp}(r) \).
- \( \text{phase}(r) = \text{TIME} \) indicates that \( r \) respects fairness and atomic properties, that \( r \) respects the guards of its outgoing edges, that \( \text{unexp} \) contains all enabled outgoing edges, that \text{urgent} has its intended meaning, but that \( r \) is not necessarily correct with respect to outgoing time edges.
- \( \text{phase}(r) = \text{ACTIVE} \) indicates that fairness and atomic properties are respected and that \text{urgent}, \text{stab}, \text{instab} and \( \text{unexp} \) have their intended meaning.

Note that the \textit{initial}, \textit{noaccess} and \textit{decided} attributes have their intended values, regardless of the phase. Splitting according to invariants is done immediately upon exploration of a new location.

Furthermore, the symbolic state space manipulation functions of definition 5.5 are assumed. Also, there are the reduction and propagation procedures.

\( \text{reduce}(c, r) \): the reduction state space manipulation function of section 5.5, illustrated in figure 5.10.

\( \text{reduce}_\text{par}(c, r) \): denotes a variation of the reduction procedure which performs the reduction test in the zone that is the result of filling in the initial values of the non-parameters in the zone of \( r \). It performs reduction on a value space in which all non-parameter variables have been given a fixed value (namely their initial values).

Note that if \( c \) does not contain any parameters, the result will always be \text{SAT} or \text{DISSAT}. This procedure is needed to test whether or not a subregion of a region that contains initial states, also contains initial states.

\( \text{propD}(c, u) \): the discrete propagation procedure of section 5.4.1

\( \text{propT}(c_1, c_2) \): the time propagation procedure of section 5.4.2

The algorithm starts from an extended XTG system \( X' \). The expansion of parallelism is done on-the-fly, the algorithm operates directly on the extended system \( X' \). The following data and functions form the interface to \( X' \).

\( l_0 \): the initial location of the expanded XTG.

\( \text{edges}(l) \): returns the outgoing edges of an expanded location \( l \).

\( U(e) \): returns a boolean value indicating whether or not the edge \( e \) is urgent, where \( e \) is an edge from the expanded XTG.
\(P(x)\): where \(x \in (E \cup \tau)\), returns a boolean value that is \textsc{true} if and only if \(x\) is an expanded XTG edge that originates from the property graph. Thus, it returns \textsc{false} if \(x\) is a XTG edge not from the property graph or if \(x = \tau\).

\(I(l)\): returns the invariant of an expanded location \(l\)

Thus the \textit{edges} function performs the actual expansion of parallelism. Given a expanded location \(l\), it inspects the XTG system description and returns the set of edges starting from \(l\) that would be in the expanded XTG.

### 6.3.3 The algorithm

Figure 6.2 shows the main loop of the algorithm. Lines 1 – 3 initialize the data structures and create the initial region. For each newly explored location, first an initial region is created. For the initial location this is done in line 1 \((r_i := \text{new\_tree}(l_0))\). Regions corresponding to newly explored locations are first split according to the appropriate invariant (for the initial location, in line 4), to avoid that inaccessible regions are further processed.

The model checking loop (lines 5 – 42) continues until all initial regions are decided, \textit{init\_count} keeping track of the number of undecided initial regions. As long as \(S\) is non-empty, a region is selected from \(S\) for further processing (line 7). Note that the implementation of \(S\) is left unspecified. It could be implemented as a stack, resulting in a depth-first search, but also as a queue, resulting in a "fair" selection method, resembling breadth-first search. Lines 8 – 9 make sure that only leaf regions are taken into account. Lines 10 – 11 make sure that decided regions are not considered. Thus only for undecided leaf regions the algorithm arrives at line 13, where depending on the phase of that region, appropriate processing is started. This can either be

1. \(\text{phase}(r) = \text{fair}\): Splitting according to fairness predicates \((\text{fair\_split}, \text{line } 14)\).
2. \(\text{phase}(r) = \text{atom}\): Splitting according to atomic properties \((\text{atom\_split}, \text{line } 15)\).
3. \(\text{phase}(r) = \text{edges}\): Adding outgoing XTG edges \((\text{line } 16)\).
4. \(\text{phase}(r) = \text{guards}\): Splitting according to guards of outgoing XTG edges and removing edges that are not enabled \((\text{guard\_split}, \text{line } 17)\).
5. \(\text{phase}(r) = \text{time}\): Adding time edges and splitting to ensure stability with respect to these time edges \((\text{time\_split}, \text{line } 18)\).
6. \(\text{phase}(r) = \text{active}\): Exploring a new edge and stabilizing this edge \((\text{lines } 26 – 37)\).
7. also \(\text{phase}(r) = \text{active}\): Splitting to stabilize an existing but instable outgoing edge \((\text{lines } 23, 24 \text{ and } 36, 37)\).

These activities are ordered. Generally, an activity will only be performed if previous activities have been completed. This is enforced by the phase attribute of regions. The only exception is that the two final steps are not ordered. This concerns regions that are in the \textit{active} phase. Only from these regions, new edges are iteratively explored.

As a result of splitting that is initiated in the main loop, splits will be propagated back through the region graph by three stabilization functions \(\text{prop\_split}_D, \text{prop\_split}_T\) and \(\text{prop\_split}_{TI}\).

There are a number of reasons for structuring the processing steps in this way. First, it is part of the strategy to avoid unnecessary splitting as was discussed in section 6.3.1. A second reason is that adding edges after splitting according to fairness and atomic properties, avoids that outgoing edges have to be stored for regions that become decided
1. $r_1 := \text{new\_tree}(l_0)$; $\text{Stk}_S := \emptyset$; $\text{Stk}_C := \emptyset$; $S := \{r_1\}$; $C := \emptyset$; $L_{\exp} := \{l_0\}$
2. $\text{instab}(r_1) := \emptyset$; $\text{stab}(r_1) := \emptyset$; $\text{initial}(r_1) := \text{true}$; $\text{init\_count} := 1$; $\text{bound}(r_1) := \bot$
3. $\text{reachable}(r_1) := \text{false}$; $\text{urgent}(r_1) := \text{false}$; $\text{phase}(r_1) := \text{fair}$
4. $\text{invariant\_split}(r_1, I(l_0))$
5. while $\text{init\_count} \neq 0$ do
6. if $S \neq \emptyset$ then
7. choose some $r \in S$
8. if $\text{sub}(r) \neq \{r\}$ then
9. $S := (S \setminus \{r\}) \cup \{r' \in \text{sub}(r) \mid \text{reachable}(r')\}$
10. else if $\text{decided}(r)$ then
11. $S := S \setminus \{r\}$
12. else
13. cases $\text{phase}(r)$
14. case $\text{fair}$: $\text{fair\_split}(r, F)$
15. case $\text{atom}$: $\text{atom\_split}(r, \text{atoms}(r))$
16. case $\text{edges}$: $\text{unexp}(r) := \text{edges(location}(r)); \text{phase}(r) := \text{GUARDS}$
17. case $\text{guards}$: $\text{guard\_split}(r, \text{unexp}(r))$
18. case $\text{time}$: $\text{time\_split}(r)$
19. case $\text{active}$:
20. if $\text{instab}(r) = \emptyset$ and $\text{unexp}(r) = \emptyset$ then
21. $S := S \setminus \{r\}; C := C \cup \{r\}$
22. else
23. if $\text{instab}(r) \neq \emptyset$ then
24. choose some $\langle r', e, c \rangle \in \text{instab}(r)$; $\text{instab}(r) := \text{instab}(r) \setminus \{\langle r', e, c \rangle\}$
25. else
26. choose some $e = \langle l, -, u, l' \rangle \in \text{unexp}(r)$; $\text{unexp}(r) := \text{unexp}(r) \setminus \{e\}$
27. if $l' \in L_{\exp}$ then
28. $r' := \text{get\_tree}(l')$
29. else
30. $\text{new\_tree}; r' := \text{get\_tree}(l'); L_{\exp} := L_{\exp} \cup \{l'\}$
31. $\text{phase}(r') := \text{fair}; \text{instab}(r') := \emptyset; \text{stab}(r') := \emptyset; \text{initial}(r') := \text{false}$;
32. $\text{bound}(r') := \bot$; $\text{reachable}(r') := \text{false}$; $\text{urgent}(r') := \text{false}$
33. $\text{invariant\_split}(r', I(l'))$
34. if $P(e)$ then
35. push($\text{Stk}_S, S$); push($\text{Stk}_C, C$); $S := \emptyset$; $C := \emptyset$
36. if $e = \tau$ then $\text{prop\_split\_T}(r, c, r')$ else $\text{prop\_split\_D}(r, c, r')$
37. $S := S \cup \{\langle r'' \rangle \mid \text{reachable}(r'')\}$
38. $V := \{\langle r'', \phi \rangle \mid r'' \in \text{sub}(r') \land \phi \in \text{props}(r'') \land \neg \text{atom}(r'') \land$
39. $\text{reachable}(r'') \land \text{val}(r'')(\phi) \neq U\}$
40. $\text{prop\_agate\_values}()$
41. else
42. $\text{decide\_cycles}(C)$
43. if $\text{Stk}_S \neq \emptyset$ then $S := \text{pop}(\text{Stk}_S); C := \text{pop}(\text{Stk}_C)$

Figure 6.2: The main model checking loop
or unreachable during this splitting. Another reason has to do with urgency. The fourth step splits regions with respect to guards (corresponding to edges that were added in the previous step). As a result of this splitting one is able to distinguish between urgent regions (those having at least one outgoing urgent edge) and non-urgent regions (those having no outgoing urgent edges). The first are marked as urgent, and are special because these regions (and their subregions) will not have outgoing time edges.

Splits according to fairness predicates, atomic properties and guards have to be propagated back through incoming edges of the region under investigation, ensuring stability with respect to incoming discrete edges and enabling the tracing of reachability. However, time edges can not yet be considered in this phase, since it is yet unknown whether the region is urgent or not. This is why the time_split procedure in the fifth step is needed. It adds time edges to non-urgent regions and stabilizes with respect to these edges.

Activities 6 and 7 form the main body of the algorithm. Activities 1 to 5 can be considered as necessary preprocessing which is done only once for each region — in fact, it corresponds to the initialize procedure of the reachability algorithm of chapter 3. Once this is completed, the region will definitively be marked as active. If for such a region all outgoing edges have been explored and are stable, it is removed from \( S \) and added to \( C \) (lines 20 – 21), where it waits to be processed later in the decide_cycles procedure. Otherwise either an instable edge is chosen (line 24) or an unexplored edge is chosen (line 26). For the first case, the algorithm can immediately proceed with stabilizing the region with respect to this edge (line 36). In case of a newly explored edge, there are two possibilities. If the destination was visited already, the region presentation of that location is retrieved (line 28) and we can again proceed with stabilizing \( r \). Otherwise, a new region representation for the destination location is initialized and split into regions that respect the invariant of the location (lines 30 – 33). If the edge originates from the property variable automaton, then the algorithm shifts to a new "subspace" and pushes the current sets \( S \) and \( C \) to the stacks \( Stk_S \) and \( Stk_C \), and starts with fresh sets \( S \) and \( C \). (line 34, 35). In any case reachable destination regions of processed edges are always added to \( S \) (line 37), ensuring that these regions will be processed in a future iteration.

\[
\text{Figure 6.3: } \text{invariant_split}
\]

At the end of each iteration, the propagate_values procedure re-evaluates regions for which new information may have become available with respect to the satisfaction of subproperties (line 39). These are stored in \( V \). There are two situations which require re-evaluation of a region with respect to subproperties. The first situation occurs when atomic properties are decided, which is done in atom_split (line 15). In atom_split, newly
Chapter 6 Model checking TCTL

decided region, subproperty combinations are stored in $V$. The second situation occurs in case an edge is added to an existing location (line 36). The new destination location may have decided values that affect the source location. Therefore, at line 38, all decided $\langle r'', \phi \rangle$ combinations with $r'' \in \text{sub}(r')$ are stored in $V$ for processing in $\text{propagate}\_\text{values}$.

If $S$ becomes empty then there are no more regions to be processed for this subspace. In that case, the cycle decisions can be made ($\text{decide}\_\text{cycle}$, discussed later), and, if possible, the exploration backs up to a ‘parent’ subspace (lines 41 – 42).

The remainder of this subsection discusses the individual procedures for splitting and decision making, starting with $\text{invariant}\_\text{split}$. Every time a new location is explored, the corresponding initial region is immediately split to separate accessible states from inaccessible states (lines 4 and 33). The latter are marked as inaccessible (the noaccess attribute). This is the task of the $\text{invariant}\_\text{split}$ procedure in figure 6.3. Remember that it assumes that invariants are conjunctions of single constraints, represented as a set.

The $\text{invariant}\_\text{split}$ procedure splits according to each single constraint in the set. If some region is a candidate for splitting, it is first checked whether or not splitting is necessary (line 44). This is done using the reduction procedure introduced in chapter 5. As we will see, this pattern of using the $\text{reduction}$ procedure will occur in each splitting procedure. Subregions that do not satisfy one of the constraints are marked inaccessible and do not have to be split any further (lines 47 and 51). If for a region a constraint represents a "true split" for that region ($\text{reduce}$ returns $\text{SPLIT}$, line 45), then the split is performed (line 46), and the subregion dissatisfying $c(\text{left}(r))$ is marked inaccessible and requires no further processing. For the subregion satisfying $c(\text{right}(r))$ splitting is continued according to the remaining single constraints in the invariant. The cases in which the reduction procedure returns $\text{SAT}$ or $\text{DISSAT}$ should now speak for themselves.

Figure 6.4 shows the pseudocode descriptions of $\text{fair}\_\text{split}$ and $\text{atom}\_\text{split}$. $\text{fair}\_\text{split}$ splits a region $r$ into subregions that respect each fairness predicate $f$ in the fairness property $F$. It is a recursive procedure which takes one fairness predicate at the time, and splits according to each of the single constraints in that fairness predicate. If a subregion satisfies all single constraints in a fairness predicate, this is stored in the fairness valuation $fval$ (line 67). It is avoided that unreachable regions are unnecessarily split (lines 54 and 62). If a region respects all fairness predicates, it is promoted to the next processing phase (line 53).

The $\text{atom}\_\text{split}$ procedure is similar. In this case the valuation $val$ has to be updated whenever a subregion is found to satisfy or dissatisfy an atomic property (lines 77, 83 and 86). Decided regions are stored in $V$, to allow $\text{propagate}\_\text{values}$ to check if other values can be decided as a consequence of these decisions.

The $\text{guard}\_\text{split}$ procedure in figure 6.5 splits a region $r$ into subregions that respect the guards of edges in $E$. It follows the same pattern as the procedures above. If a region is found not to satisfy a guard, the edges to which the guard belongs are not enabled in that region and therefore removed from that region’s set of unexplored edges (lines 95 and 101). If a region is found to satisfy the guard of an urgent edge, the $\text{urgent}$ attribute of that region is set (line 104).

The $\text{time}\_\text{split}$ procedure in figure 6.6 splits a region into subregions that are stable with respect to time edges and adds these edges where appropriate. Obviously, it is only applied to non-urgent regions. It is based on the fact that a non-urgent region that is stable with respect to time edges, has at most one outgoing time edge, and also, one upper bound. A non-urgent region always has an outgoing time edge except if the passing of time in a region always leads to inaccessible regions, or if a region is unbounded. The constraints
\(\text{fair_split}(r, F)\):

53. if \( F = \emptyset \) then \( \text{phase}(r) := \text{ATOM} \); return

54. if \( \text{reachable}(r) \) then

55. choose some \( f \in F \)

56. for each \( c \in f \) do

57. cases \( \text{reduce}(c, r) \)

58. case \text{SPLIT}:

59. \( \text{split}(r, c) \)

60. \( \text{fair_split}(\text{left}(r), F \setminus \{f\}) \)

61. \( r := \text{right}(r) \)

62. if not \( \text{reachable}(r) \) then return

63. case \text{SAT}: (do nothing)

64. case \text{DISSAT}:

65. \( \text{fair_split}(r, F \setminus \{f\}) \)

66. return

67. \( \text{fval}(r) := \text{fval}(r) \cup \{f\} \)

68. \( \text{fair_split}(r, F \setminus \{f\}) \)

\(\text{atom_split}(r, P)\):

69. if \( P = \emptyset \) then \( \text{phase}(r) := \text{EDGES} \); return

70. if \( \text{reachable}(r) \) then

71. choose some \( p \in P \)

72. if \( \text{val}(r)(p) \neq \text{U} \) then \( \text{atom_split}(r, P \setminus \{p\}); \text{return} \)

73. for each \( c \in p \) do

74. cases \( \text{reduce}(c, r) \)

75. case \text{SPLIT}:

76. \( \text{split}(r, c) \)

77. \( \text{val}(\text{left}(r))(p) := \text{F}; \mathcal{V} := \mathcal{V} \cup \{(\text{left}(r), p)\} \)

78. \( \text{atom_split}(\text{left}(r), P \setminus \{p\}) \)

79. \( r := \text{right}(r) \)

80. if not \( \text{reachable}(r) \) then return

81. case \text{SAT}: (do nothing)

82. case \text{DISSAT}:

83. \( \text{val}(r)(p) := \text{F}; \mathcal{V} := \mathcal{V} \cup \{(r, p)\} \)

84. \( \text{atom_split}(r, P \setminus \{p\}) \)

85. return

86. \( \text{val}(r)(p) := \text{T}; \mathcal{V} := \mathcal{V} \cup \{(r, p)\} \)

87. \( \text{atom_split}(r, P \setminus \{p\}) \)

Figure 6.4: \text{fair_split} and \text{atom_split}

that define a region can be retrieved by traversing up the splitting tree. Some of these constraints will be upper bounds. The \text{time_split} procedure splits a region into subregions that each have a single upper bound, namely one of the upper bound constraints found when traversing up the tree.

\(\text{time_split}^*\) is the recursive body of \text{time_split}. It has three parameters:

- \( x \) is the region under investigation, \( \text{bound}(x) \) contains the current candidate for the
\begin{algorithm}
\textbf{guard\_split} \((r, E)\):
  \begin{algorithmic}[1]
    \State \textbf{if} \(E = \emptyset\) \textbf{then} \textbf{phase}(r) := \text{TIME}; \textbf{return}
    \State \textbf{if} \text{reachable}(r) \textbf{then}
    \State \hspace{1em} \textbf{choose some} \(e = \langle -, g, -, - \rangle \in E\)
    \State \hspace{1em} \textbf{for each} \(c \in g\) \textbf{do}
    \State \hspace{2em} \textbf{cases} \text{reduce} (c, r)
    \State \hspace{3em} \textbf{case} \text{SPLIT}:
    \State \hspace{4em} \text{split} (r, c)
    \State \hspace{4em} \text{unexp}(\text{left}(r)) := \text{unexp}(\text{left}(r)) \setminus \{e\}
    \State \hspace{4em} \text{guard\_split} (\text{left}(r), E \setminus \{e\})
    \State \hspace{4em} \textbf{r} := \text{right}(r)
    \State \hspace{3em} \textbf{if not} \text{reachable}(r) \textbf{then return}
    \State \hspace{3em} \textbf{case} \text{SAT}: \text{(do nothing)}
    \State \hspace{3em} \textbf{case} \text{DISSAT}:
    \State \hspace{4em} \text{unexp}(r) := \text{unexp}(r) \setminus \{e\}
    \State \hspace{4em} \text{guard\_split} (r, E \setminus \{e\})
    \State \hspace{4em} \textbf{return}
    \State \textbf{if} \text{U}(e) \text{ then} \text{urgent}(r) := \text{TRUE}
    \State \textbf{guard\_split} (r, E \setminus \{e\})
  \end{algorithmic}
\end{algorithm}

Figure 6.5: \textit{guard\_split}

upper bound of \(x\), if there is one (otherwise \(\text{bound}(x) = \bot\)).

\begin{itemize}
  \item \(d\) is the current candidate to be the destination of a time edge from \(x\). Initially it is set to \(\bot\).
  \item \(t\) is the region that is currently inspected to find alternative destination regions. Initially it is set to the parent of the region to which \textit{time\_split} is applied. At all times, \(x\) and \(d\) are subregions of \(t\).
\end{itemize}

The idea is to ensure that when only considering the partitioning defined by the subtree of which \(c\) is the top node, \(\text{bound}(x)\) holds the upper bound of \(x\) if there is one, and that \(d\) is the destination of an outgoing time edge, if there is one. Iteratively the subtree is enlarged by replacing \(t\) by its parent and checking if new bounds occur. At some point \(t\) will be equal to the top node of the tree, resulting in the correct values for the bound and the destination region for the outgoing time edge (if present).

Thus, at each invocation of \textit{time\_split}\(\ast\) the current subtree is enlarged by taking the constraint of the parent of \(t\) into account. Lines 112 – 118 check whether the new constraint is an upper bound, and if so, stores this constraint in \(\text{cons}\), and stores the region "on the other side" of the bound in \(\text{nd}\). If the constraint is not an upper bound, then that constraint is not relevant and the algorithm proceeds one level up in the splitting tree (line 117). Otherwise, if this is the first upper bound that is encountered (i.e. \(d = \bot\)), then it is stored in the \textit{bound} attribute of the current region \(x\) and the region \(\text{nd}\) is stored in \(d\). This reflects the fact that in the context of the subtree under \(t\), \(x\) has a time edge to \(d\) with separating constraint \(\text{bound}(x)\). The algorithm then proceeds one level up the splitting tree (lines 120, 121).

If on the other hand, we already had a bound for \(x\) (which is stored in \(\text{bound}(x)\)), then \(x\) has to be split to separate the states for which \(\text{bound}(x)\) is the valid bound, from those for which the new bound \(\text{cons}\) is relevant. The constraint that does this \((\text{ncons})\) is
time_split (r):  
106. phase(r) := ACTIVE  
107. if not urgent(r) then  
108.  bound(r) := ⊥  
109.  time_split*(r, ⊥, r)  

time_split* (x, d, t):  
110. [if not reachable(x) then return ]  
111. if parent(t) ≠ ⊥ then  
112. if type(t) = SAT and cons(parent(t)) is an upper bound then  
113.  cons := cons(parent(t)); nd := left(parent(t))  
114. else if type(t) = DISSAT and cons(parent(t)) is a lower bound then  
115.  cons := ¬cons(parent(t)); nd := right(parent(t))  
116. else  
117.  time_split* (x, d, parent(t))  
118. return  
119. if d = ⊥ then  
120.  bound(x) := cons  
121.  time_split* (x, nd, parent(t))  
122. else  
123.  ncons := propT (cons, bound(x))  
124. cases reduce (ncons, x)  
125. case SPLIT:  
126.  split (x, ncons)  
127.  bound(left(x)) := cons  
128.  time_split* (right(x), d, parent(t))  
129.  time_split* (left(x), nd, parent(t))  
130. case SAT:  
131.  time_split* (x, d, parent(t))  
132. case DISSAT: (cannot happen)  
133. else  
134. if d ≠ ⊥ then  
135.  prop_split_T (x, bound(x), d)  

Figure 6.6: time_split

computed by the propT procedure of chapter 5. Reduction has to be applied because it
could be that for all states in x the original bound still applies (line 130). In that case cons
is not a relevant upper bound, and the algorithm proceeds one level up the splitting tree
(line 131). In case ncons represents a "true split" for x (line 125), x is actually split. For
the subregion that does not satisfy ncons (i.e. left(x)) the upper bound is changed to cons
(line 127) and the candidate destination region is changed to nd. For both subregions the
algorithm proceeds one level up the tree (lines 127 – 129). Note that right(s) automatically
inherits the bound attribute from its parent (the split procedure).

The recursion ends when the top node of the splitting tree is reached (line 133). If
no bound was found, apparently time is not constrained in this region and no time edge
has to be added. Alternatively, if a bound was found then x has a stable time edge to
split \( r, c \) :

136. let pred = stab\( r \), initial = initial\( r \)

137. extend \( r, c \); copy phase, val, fval, urgent, unexp, instab and bound, attributes from

\( r \) to left\( r \) and right\( r \) and initialize reachable and initial attributes to false

138. for each \( \langle r', e, c \rangle \in \text{stab}\( r \) do

139. if \( e = \tau \) then prop\_split\_T \( r', c, r \) else prop\_split\_D \( r', e, c, r \)

140. if initial then

141. cases reduce\_par \( c, r \)

142. case split:

143. initial\( \text{left}(r) \):= \text{true}; reachable\( \text{left}(r) \):= \text{true}

144. initial\( \text{right}(r) \):= \text{true}; reachable\( \text{right}(r) \):= \text{true}

145. init\_count := init\_count + 1

146. case SAT:

147. initial\( \text{right}(r) \):= \text{true}; reachable\( \text{right}(r) \):= \text{true}

148. case DISSAT:

149. initial\( \text{left}(r) \):= \text{true}; reachable\( \text{left}(r) \):= \text{true}

150. if phase\( \text{left}(r) \) = active and not urgent\( \text{left}(r) \) then

151. prop\_split\_TI \( r \)

Figure 6.7: split

\( d \). However, \( d \) is likely to be a region that is split itself. This means that the time edge
cannot be simply added to stab\( d \), but that the splitting of \( d \) has to be propagated back
to \( x \) (line 135).

Figure 6.7 shows the split procedure, which is responsible for actually splitting a region
with respect to some constraint. It assumes that it is given a constraint that represents a
true split of the region (i.e. reduction was already applied). It starts with implementing
the split in the forest (line 137), copying the relevant attributes from \( r \) to its children
left\( r \) and right\( r \). Line 136 is needed to remember certain attributes of \( r \). The split is
likely to cause predecessors of \( r \) to become instable. Therefore the splitting is propagated
to each of its incoming edges (lines 138, 139). For time edges this is taken care of by
prop\_split\_T, while for discrete edges this is done by the prop\_split\_D procedure.

Lines 140 – 149 check whether a region holds initial states. This check is only needed
if the parent region \( r \) also held initial states (initial\( r \) = true). The reduce\_par\( c, r \)
procedure checks how the new constraint relates to the subset of initial states in a region.
If it returns split this means that there are initial states in \( r \) that satisfy \( c \) as well as initial
states that do not satisfy \( c \). The cases for SAT and DISSAT have the expected meaning. If
a region has initial states, then the reachable attribute as well as the initial attribute are
set for that region. Also, if both subregions have initial states, the number of undecided
initial regions is increased with one (line 145).

Lines 150 – 151 implement the effect of the implicit self-looping time edge. If \( r \) is not urgent
and it is split by an upper or lower bound, then one of the new subregions is likely
to have become instable with respect to time edges. This corresponds to the internal time
propagation case discussed in section 5.4.2. The prop\_split\_TI procedure is designed to
deal with this situation. The reason for the phase\( \text{left}(r) \) = active condition is that —
as discussed above — if the region is not yet in phase active, then time edges are not
yet taken into account. Note that the condition in line 150 checks left\( r \) rather than \( r \)
because attributes have only meaning on leaf regions.
prop_split_D (r, e, r'):
152. if sub(r) \neq \{r\} then
153. prop_split_D (left(r), e, r')
154. prop_split_D (right(r), e, r')
155. else
156. if reachable(r) then
157. if sub(r') \neq \{r'\} then
158. [if not decided(r) then]
159. c := propD (cons(r'), u) s.t. e = \langle -, -, u, - \rangle
160. cases reduce (c, r)
161. case SPLIT:
162. split (r, c)
163. prop_split_D (left(r), e, left(r'))
164. prop_split_D (right(r), e, right(r'))
165. case SAT:
166. prop_split_D (r, e, right(r'))
167. case DISSAT:
168. prop_split_D (r, e, left(r'))
169. else
170. if \neg \text{noaccess}(r') then
171. stab(r') := stab(r') \cup \{(r, e, \bot)\}
172. reachable(r') := \text{TRUE}
173. else
174. instab(r) := instab(r) \cup \{r', e, \bot\}

Figure 6.8: prop_split_D

Figures 6.8, 6.10, and 6.12 show the three procedures that are primarily responsible for stabilization. prop_split_D (r, e, r') stabilizes r with respect to an e-edge to r', while prop_split_T (r, c, r') stabilizes r with respect to a time edge to r', where c is the constraint that separates r and r'. These functions are called when either a destination of a stable edge is split (line 139) or a new edge is added (line 36 and line 135). Furthermore, prop_split_TI deals with internal time propagation.

\[ \text{invariant_split} \rightarrow \text{fair_split} \rightarrow \text{atom_split} \rightarrow \text{split} \rightarrow \text{prop_split_D} \rightarrow \text{prop_split_TI} \rightarrow \text{prop_split_T} \rightarrow \text{guard_split} \rightarrow \text{time_split} \]

Figure 6.9: Call graph for the splitting procedures

The three stabilization procedures together with the split procedure implement the propagation of splits, by recursive mutual calls. Figure 6.9 illustrates how the different splitting procedures invoke each other. The procedures on the left represent the "initial" splitting initiated in the main loop. As a consequence of such splits, there can be a sequence of propagated splits, implemented by the split and prop_split procedures. Also,
in case a new edge is added, splitting is initiated directly through the prop_split procedures (illustrated by the edge coming from the right).

First consider the recursive procedure prop_split_D. It is essential to note that a precondition of prop_split_D(r, c, r') is that r is stable with respect to the e edge to r'. Lines 152 – 154 ensure that the source region is a leaf region, by calling prop_split_D for each of the subregions (which are necessarily also stable with respect to the e edge to r'). If r is not reachable, stabilization should not continue, since refining unreachable states is a waste. Instead this edge is stored in instab(r) (line 174). It will be reconsidered only in case its source region r becomes reachable again. If r' is not split (line 157), then the edge can simply be added to stab(r'), provided that r' is not excluded by an invariant (line 170). r' must be marked reachable since it has an edge from the reachable region r. Furthermore, if r was decided already, then refining r also serves no purpose (line 158).

We then arrive at the body of the propagation procedure (lines 159 – 168), which means that propagation is really required. The propD function of chapter 5 is used to propagate the split of r' back to r. The resulting constraint is again submitted to the reduction procedure. If the constraint represents a true split of r (line 161), then the split is actually performed (line 162), resulting in edges from left(r) to left(r') and from right(r) to right(r'). However, left(r') and right(r') could very well also be split. Therefore, propagation continues by recursive calls to prop_split_D (lines 163, 164). In case the new constraint is not a true split for r, the algorithm recursively continues by considering either the left or right child of r', whichever is appropriate (lines 166 and 168). The recursion ends when regions become unreachable (line 174), or in case r' is a leaf region and an edge from r to r' can be added (line 171).

The prop_split_T procedure (figure 6.10) is similar to prop_split_D, but there are some differences. First of all, the computation of the propagated constraint is now performed by the propT function of chapter 5 rather than propD (line 184). Also, if the splitting constraint is not an upper bound or lower bound, propagation simply comes down to copying the constraint (line 186) (c could for example be a constraint on enumerated type variables). Another difference is that internal time propagation is not needed, since the propagated constraints are inherently non-bounds.

Finally, there is a subtlety in dealing with the case that r is split. Like in prop_split_D, if r is split, then the procedure is called for the subregions of r (lines 175 – 177). However, if r was split by an upper or lower bound, this would be incorrect, because the split should only be propagated to one of the two subregions, namely the one of which the bound is equal to the separating constraint c. This is why the check in line 179 is included (and also the sole reason for maintaining the bound attribute). If the constraint that separates r and r' is equal to the bound of r, then there is a time edge from r to r'. Figure 6.11 illustrates this. There is a time edge from r to r' stored in stab(r'). At some moment, r became split in subregions r1 and r2. Now, suppose r' becomes split, then this split will be propagated back to r. However, it should only be propagated to r1 and not to r2.

Finally, there is the prop_split_TI procedure. Given a region r which has been split, it is responsible for adding any time edges between the two subregions of r, splitting were necessary to ensure stability. This procedure is also responsible for maintaining the bound attributes. First, it checks whether the constraint that splits r is a bound (lines 202 – 207). If not (line 207), no time edge needs to be added. Otherwise, r1 is made to point to the subregion of r from which the (potential) time edge departs, while r2 is made to point to destination of the potential time edge. The splitting constraint is stored in c, as an upper bound. If r was unbounded (and thus bound(r1) = ⊥), we can add an edge from r1 to r2,
prop_split_T(r, c, r'):
175. if \( \text{sub}(r) \neq \{ r \} \) then
176. \( \text{prop_split_T}(\text{left}(r), c, r') \)
177. \( \text{prop_split_T}(\text{right}(r), c, r') \)
178. else
179. if \( c = \text{bound}(r) \) then
180. if \( \text{reachable}(r) \) then
181. [if not decided(r) then]
182. if \( \text{cons}(r) \) is an upper or lower bound then
183. \( c' := \text{propT}(\text{cons}(r), c) \)
184. else
185. \( c' := \text{cons}(r) \)
186. cases reduce\( (c', r) \)
187. case SPLIT:
188. \( \text{split}(r, c') \)
189. \( \text{prop_split_T}(\text{left}(r), c, \text{left}(r')) \)
190. \( \text{prop_split_T}(\text{right}(r), c, \text{right}(r')) \)
191. else
192. if \( \neg \text{noaccess}(r') \) then
193. \( \text{stab}(r') := \text{stab}(r') \cup \{ \langle r, \tau, c \rangle \} \)
194. \( \text{reachable}(r') := \text{TRUE} \)
195. else
196. \( \text{instab}(r) := \text{instab}(r) \cup \{ r', \tau, c \} \)
197. Figure 6.10: prop_split_T

because in that case all states in \( r_1 \) will have a transition to a state in \( r_2 \). This is done by the prop_split_T procedure that was discussed earlier, to make sure that the edge is added in the proper manner. If \( r \) was not unbounded, the algorithm has to separate the states that have the old bound of \( r \) as upper bound from those that have the splitting constraint \( c \) as upper bound. According to theorem 5.6 we can use propT for that (line 211). The resulting constraint \( c' \) identifies the subregion of \( r_1 \) for which the existing bound is still

\[
\begin{array}{c}
r' \\
| \\
| \\
\hline
r_1 \\
\hline
r \\
\hline
r_2
\end{array}
\]

Figure 6.11: Illustrating a time propagation subtlety
**prop_split_TI** \( (r) \):

202. if \( \text{cons}(r) \) is a lower bound then
203. let \( r_1 = \text{left}(r), r_2 = \text{right}(r), c = \neg \text{cons}(r) \)
204. else if \( \text{cons}(r) \) is an upper bound then
205. let \( r_1 = \text{right}(r), r_2 = \text{left}(r), c = \text{cons}(r) \)
206. else
207. return
208. if \( \text{bound}(r_1) = \perp \) then
209. \( \text{prop_split}_T(r_1, c, r_2) \)
210. else
211. \( c' := \text{prop}_T(\text{bound}(r_1)) \)
212. cases \( \text{reduce} (c', r_1) \)
213. case split:
214. \( \text{split}(r_1, c'); \text{bound}(\text{left}(r_1)) := c \)
215. \( \text{prop_split}_T(\text{left}(r_1), c, r_2) \)
216. case sat: (cannot happen)
217. case dissat:
218. \( \text{bound}(r_1) := c \)
219. \( \text{prop_split}_T(r_1, c, r_2) \)

Figure 6.12: prop_split_TI

valid. Thus, for the region that does not satisfy \( c' \) — if it exists — a new time edge to \( r_2 \) has to be added and the bound has to be set to the splitting constraint \( c \). Again, reduction is applied to avoid empty regions.

Together, split, prop_split_D, prop_split_T and prop_split_TI realize the backward propagation of splits through the state space. Every time a region is split, the propagation procedures are called to stabilize predecessor regions, which may again lead to more splitting. This splitting cannot go on forever, because only reachable regions are allowed to be split, and reachability attributes are set after backward propagation has been performed.

Now consider the aspects of the algorithm that deal with deciding values. Section 6.2.3 showed that there are four types of decisions. For the atomic properties (the type 1 decision) the decisions are made in atom_split. The propagate_values function deals with type 2 and 3 decisions — the successor-based and local decisions. These are the decisions that are made locally as a consequence of earlier decisions. The type 4 decisions require the inspection of strongly connected components. This is handled by the decide_cycles procedure discussed later.

Figure 6.13 shows the propagate_values procedure. It operates on the set \( V \) of decided \( \langle r, \phi \rangle \) combinations for which the effect on other values has not yet been evaluated. For each \( \langle r, \phi \rangle \) combination, it is first checked, if there are predecessor regions within the current subspace that could be affected. This can only be the case if \( \phi \) is a temporal operator. In that case \( Enext \) or \( Anext \) values of predecessors may have to be decided. This is done in lines 221 – 229, in accordance with the first two type 2 rules. Subsequently, it is checked if there is a predecessor region in another subspace. This can only be the case if the parent property of the current property is an update property. If so, values are decided in accordance with the final two type 2 rules (lines 230 – 234). Finally, there are the type 3 decisions which are relevant in case the current property does have a parent property (i.e.
propagate_values():
220. while \( V \neq \emptyset \) do
221.  choose some \( \langle r, \phi \rangle \in V \); \( V := V \setminus \{ \langle r, \phi \rangle \} \)
222.  if \( \text{val}(r)(\phi) = T \) and \( \phi \) is a \( EU \) or \( E\text{true} \) formula then
223.    for each \( \langle r', x, - \rangle \in \text{stab}(r) \) s.t. \( \neg P(x) \) do
224.      for each \( rr' \in \text{sub}(r') \) s.t. \( \text{val}(rr')(\text{Enext}(\phi)) = U \) do
225.        \( \text{val}(rr')(\text{Enext}(\phi)) := T \); \( V := V \cup \{ \langle rr', \text{Enext}(\phi) \rangle \} \)
226.  if \( \text{val}(r)(\phi) = F \) and \( \phi \) is a \( AU \) formula then
227.    for each \( \langle r', x, - \rangle \in \text{stab}(r) \) s.t. \( -P(x) \) do
228.      for each \( rr' \in \text{sub}(r') \) s.t. \( \text{val}(rr')(\text{Anext}(\phi)) = U \) do
229.        \( \text{val}(rr')(\text{Anext}(\phi)) := F \); \( V := V \cup \{ \langle rr', \text{Anext}(\phi) \rangle \} \)
230.  for each \( \phi' \in \text{par}(\phi) \) do
231.    if \( \phi' = u. \phi'' \) for some \( u \) and \( \phi'' \) then
232.      if \( \exists \langle r', x, - \rangle \in \text{stab}(r) \) s.t. \( P(x) \) then
233.        for each \( rr' \in \text{sub}(r') \) s.t. \( \text{val}(rr')(\phi'') = U \) do
234.          \( \text{val}(rr')(\phi'') := \text{val}(r)(\phi); V := V \cup \{ \langle rr', \phi'' \rangle \} \)
235.    else if \( \text{val}(r)(\phi') = U \) then
236.       apply type 3 rules for \( \text{val}(r)(\phi') \)
237.      if \( \text{val}(r)(\phi') \neq U \) then
238.        \( V := V \cup \{ \langle r', \phi' \rangle \} \)
239.    if \( \text{par}(\phi') = \bot \) and initial\( (r) \) then
240.      \( \text{init\_count} := \text{init\_count} - 1 \)

Figure 6.13: propagate_values

it does not coincide with the "top" property and if that parent property is not an update property. For this case the type 3 rules are applied (line 236). To keep the pseudocode simple, these rules are not explicitly included here.

In any case, if a new value is decided, it is stored in \( V \), such that the consequences of the new decision will also taken into account. The propagation of values will always terminate, since only \( U \) values are decided and the number of regions is finite. If the "top" property of an initial region is decided (line 240), \( \text{init\_count} \) has to be decreased (since \( \text{init\_count} \) keeps track of the number of undecided initial regions).

In an implementation, values would be propagated in a more efficient manner, by avoiding needless propagations of values. To keep descriptions simple, these improvements are not shown here. In general, the storage and propagation of values can be implemented quite efficiently. As a result, time and space consumption related to value storage and propagation is small compared to that of storage and manipulation of the symbolic state space.

Finally decide\_cycles, shown in figure 6.14, is still to be discussed. As was mentioned already, the approach chosen to deal with type 4 decisions, is to postpone these decisions until no more type 1 - 3 decisions can be taken. The reason is that the type 4 decisions are relatively expensive because they require searching for strongly connected components. This will result in a decision procedure that is not completely optimal, because some of the type 4 decisions could be taken at an earlier stage.

The decide\_cycles procedure is given a set \( C \) of fully explored regions, corresponding to the same subspace, on which the local decision rules were not able to decide. When decide\_cycles\( (C) \) is invoked, it is always the case that for all regions \( r \in C \), all regions
in a different subspace on which \( r \) depends, are already decided. This is enforced by the fact that with respect to the subspaces, the algorithm operates in a depth-first manner. Basically, \( \text{decide}\_\text{cycles} \) finds all fair strongly connected components in \( C \) (lines 241 – 245), and then uses this information to iteratively decide values of regions in \( C \) (lines 245 – 257).

To find the fair strongly connected components, it uses a recursive procedure called \( \text{find}\_\text{SCCs} \), that, given a region \( r \), returns the set of undecided, reachable regions from which \( r \) can be reached, that are part of a fair SCC. After some initialization (lines 241 and 242), repeatedly \( \text{find}\_\text{SCC} \) is called, until for each region its SCC has been found. \( \text{find}\_\text{SCC} \) stores all regions that have been found to be part of a fair SCC in \( FC \). Thus, at line 245, \( FC \) will contain all regions that are part of a fair subgraph.

Type 4 decisions only concern temporal operators. The procedure proceeds by first collecting all temporal subproperties that are in the scope of the subspace, and then deals
with one temporal subproperty at the time, in such an order that each time a temporal subproperty is chosen for which all temporal subproperties have already been handled (line 247). Each time values are decided, it could be that type 2 or 3 rules again apply, and therefore \texttt{propagate_values} is called. As a consequence, whenever dealing with a temporal subproperty \( \phi \), all subproperties of that subproperty have already been decided.

According to the second and third type 4 rule, regions in fair components can be decided to \texttt{true} for \( AU \) properties, while they can be decided to \texttt{false} for \( Etrue \) properties. For the latter it is evident from the rules that this is correct. For \( \phi_1 AU \phi_2 \) this follows from the fact that all regions in the fair components do not satisfy \( \phi_2 \), because otherwise a decision would already have been forced by a type 3 decision rule. Lines 249 – 252 implement the decisions based on fair components.

Subsequently, decided values are propagated (line 253). After that, all remaining undecided regions can be decided too, using the other three type 4 rules. According to these rules this means that \( AU \) and \( EU \) properties are set to \texttt{false} and that \( Etrue \) properties are set to \texttt{true} (lines 255, 256). The correctness of these decisions follows from the rules taking into account that all subproperties are already decided, and also that the yet undecided regions are not part of a fair component. Again, values have to be propagated (line 257). Line 258 is explained by the fact that if the property that is decided coincides with the top property, then all initial regions will become decided and the algorithm terminates.

\texttt{find_SCC} is based on Tarjan’s algorithm for finding strongly connected components in a graph [100]. It starts from a single region, and progresses backwards through the graph — rather than forward like Tarjan’s algorithm. Only regions that are reachable and not yet decided are considered (line 262). If a region is visited, it is pushed to a stack \( CS \) and marked as \texttt{current}. Also it is given a fresh \texttt{curroot} value, which is indicative for order in which regions are visited (\texttt{curroot} is a local variable in \texttt{find_SCCs}). Each region is visited only once, which is enforced using the \texttt{current} attribute of regions, which is set whenever a region is visited. While backtracking, the smallest root value encountered thus far, is maintained in \texttt{root}. \( \texttt{root}(r) \) holds the lowest \texttt{curroot} value encountered thus far in the exploration of the predecessors of \( r \). Thus, if a root value is returned from a predecessor that is smaller than the root value of the current region, then the latter is updated to hold the new value.

Now if for some region \( r \), all root values received from predecessors are equal or larger than the original root value of \( r \) (stored in the local variable \texttt{curroot}), this must mean that \( r \) is part of a (possibly singleton) strongly connected component, which is residing at the top of the stack. This is the case when arriving at line 266. Then, the SCC that has been found is transferred from the stack to \( CC \). The obtained strongly connected component is checked for fairness (line 269), and if fair, its members are stored in \( FC \). All regions that are found to be part of a strongly component are marked as \texttt{visited} (line 268), to avoid that these regions are investigated again (line 262). The condition in line 261 ensures that only edges that remain in the current subspace are taken into account. Furthermore, for obvious reasons, reachable as well as already decided regions are not taken into account (line 262). Finally, note that in line 262 the algorithm also deals with the peculiarity that a predecessor \( r' \) of a region \( r \) may be split into subregions. Then, the algorithm acts as if all subregions of \( r' \) are predecessors of \( r \). This assumption is not always correct when dealing with time edges (see the discussion earlier that was illustrated by figure 6.11). However, for these cases the validity of the algorithm is not violated because all subregions at least are indirect predecessors of \( r \). When checking for SCC’s this is good enough.
6.3.4 Extensions of the algorithm: optimal decision making

The algorithm presented here can be improved in several ways. Many improvements can be seen as further refinements of the algorithm, that become relevant when actually implementing the algorithm in a model checking tool. Also, extensions of the symbolic state space representation approach were already discussed in the previous section.

However, there is one issue that is worth discussing. The decision rules of section 6.2.3 were shown to be optimal. However, the algorithm applies simplified decision criteria for type 4 decisions, for reasons of simplicity and performance. Here, we consider how more optimal decision criteria could be included in the algorithm.

In the algorithm above, type 4 decisions are made only once a subspace has been completely explored. Part of these decisions can potentially be made at an earlier stage. The advantage of doing so is that the decision procedure becomes more optimal, and thus avoids unnecessary state space exploration. On the other hand, checking for strongly connected components is relatively expensive and will therefore also have a negative impact on the performance of a model checking algorithm.

Two cases can be discerned, on-the-fly fair cycle detection and on-the-fly interference-free cycle detection. Starting with the first, the decisions for fair SCS’s can be performed on-the-fly. This concerns the second and third type 4 rules. During exploration — for example each time a new edge is added — the algorithm can check whether there are new fair SCS’s. Obviously, this is only useful if there are still undecided AU or Etrue properties. Whenever a fair SCS is found, all Etrue properties can be decided, while for a φ1 AU φ2 property it has to be checked whether all regions in the SCS satisfy φ2. If so, this property can be decided for all regions in the SCS.

Figure 6.15 shows some cases that illustrate the practical advantage of on-the-fly fair SCS detection. Three cases are shown, for each a fairness condition, a TCTL property and an XTG is depicted. These are extremely simple examples, having no data, only some (abstractly represented) atomic properties. First consider the verification problem on the left side, where the dashed arrow indicates that the edge leads to another (potentially large part) of the XTG that is not shown here. One would expect that verification of this system would be fast and easy, especially if the algorithm would employ a breadth-first
strategy in exploring new edges. There is a run from the initial location to the location satisfying $f \land p$, where it can stay forever, since that location represents a fair cycle.

However, the model checking algorithm as it was presented above will first explore the part of state space that is behind the dashed arrow, because to decide $EF \ p$ the algorithm has to verify that $Etrue$ is satisfied, which requires an SCS-based decision. The latter decisions are made only after the whole (sub)space is explored. However, by checking for fair cycles on-the-fly, this problem can be decided very easily. The case in the middle of figure 6.15 shows another case for which on-the-fly fair SCS detection would be very efficient. The three locations, result in a fair SCS in which $p$ is never satisfied. Using the third type 4 rule, this means that $AF \ p$ is not satisfied in the initial states. Again, an algorithm without on-the-fly fair SCS detection would first have to explore the parts of the system that are behind the dashed arrows. Worst case, this may mean that verification will not succeed because the irrelevant parts of the state space are too large. The right-side case is an extreme form of the middle case.

For the other three type 4 rules, it is much harder to make a case for more optimal decision making — both with respect to implementation and justification. These require checking for strongly connected components rather than strongly connected subgraphs. In fact, one has to find so-called interference-free cycles, i.e. strongly connected components of fully explored regions of which all external successors have already been decided.

Mostly, on-the-fly checking for these decisions does not result in any gain because often these decisions require complete exploration of the state (sub)space anyway. However, when temporal operators are nested, examples can be found that would benefit from more optimal decision making. Figure 6.16 shows such a case. In the second location, $AF \ q$ can be decided using the first rule. As a consequence, the complete property can be decided for the initial location and the verification is finished. The part of the state space behind to dashed edge does not have to be completely explored.

$$F = \{ f \}$$

$$EF \ (p \land AF \ q)$$?

![Figure 6.16: illustrating the use of on-the-fly interference-free cycle detection](image)

Somewhat related to the example above, there is another potential benefit of checking type 4 decisions on-the-fly. When performing parametric verification, it may very well be that for a subset of the parameter values, verification is feasible (because of time or data performance), while for the rest of the parameter values it is not. An on-the-fly decision making approach supports this, while the straightforward implementation of the basic algorithm implies that all type 4 decisions are taken after the relevant part of state space has been completely explored.

Finally, there is another possible optimization when checking simple $EU$ properties — reachability properties like $EF \ p$ or $AG \ p$ — without fairness constraints. When checking $EF \ p$ without fairness this is not exactly the same as checking reachability as discussed in
prop noaccess plain reachability analysis. That this is a very specific case, but one that occurs quite often, since it covers the case of definitions and the semantics of XTG it follows that if \( S \)
\[
\text{Where } \rho \text{ of the algorithm.}
\]
decided is needed to show the correctness of the decision making aspects (of the algorithm and that the edges represented by the data structures are stable. This
\[
\text{the induced partial region graph are indeed correctly represented in the data structures}
\]
initial splitting procedures are dealt with (\( \text{atom\_split}, \text{fair\_split}, \text{guard\_split}, \text{prop\_split\_D}, \text{prop\_split\_T}, \text{prop\_split\_TT}, \text{time\_split} \) and \( \text{split} \)). This is used to show that all edges in the induced partial region graph are indeed correctly represented in the data structures of the algorithm and that the edges represented by the data structures are stable. This is needed to show the correctness of the decision making aspects (\( \text{propagate\_values} \) and \( \text{decided\_cycles} \)). Finally, this section ends with a brief discussion of termination aspects of the algorithm.

For notational convenience we will write \([r]\) instead of \(F[r]\). Also, sometimes notation will be abused by writing \([r] \cap [c]\) to denote the set of states \((l, \rho)\) in \([r]\) of which the valuation \(\rho\) satisfies \(c\). Note that now we use \(r\) and \(r', r_1, \text{etc.}\) to denote region representations rather than regions itself, although we will often use the term region. As was discussed, given a region representation \(r\), \([r]\) denotes the actual region. Furthermore, we will use two shorthands \(\text{stable}\) (\(\text{reg}, x, \text{reg}'\)) and \(\text{stable}^*\) (\(\text{reg}, x, \text{reg}'\)) where \(\text{reg}\) and \(\text{reg}'\) are regions and \(x \in (\{r\} \cup E)\), with the following meaning,
\[
\begin{align*}
\text{stable}^* (\text{reg}, x, \text{reg}') & \iff \text{reg} \cap S_{\text{urg}} = \emptyset \land \text{pre}_x (\text{reg}, \text{reg}') = \text{reg} \\
\text{stable}^* (\text{reg}, (-, g, u, -), \text{reg}') & \iff \text{reg} \subseteq [g] \land \text{pre}_u (\text{reg}, \text{reg}') = \text{reg} \\
\text{stable} (\text{reg}, x, \text{reg}') & \iff \text{reg} \xrightarrow{x} \text{reg}' \text{ is stable}
\end{align*}
\]
Where \(S_{\text{urg}}\) denotes the set of states from which an urgent edge is enabled. From these definitions and the semantics of XTG it follows that if \(\text{stable}^* (\text{reg}, x, \text{reg}')\) and all states in \(\text{reg}\) and \(\text{reg}'\) satisfy the invariants, then \(\text{stable} (\text{reg}, x, \text{reg}')\).
Partial state space. The set of regions \( R \) should define a partial state space corresponding to locations in \( L_{\text{exp}} \). We require that, when arriving at line 39,

\[
\forall (l, \rho) \in S . (l \in L_{\text{exp}} \iff \exists r \in R : (l, \rho) \in \{r\}) \tag{6.13}
\]

For each location that is added to \( L_{\text{exp}} \), an initial region is created that covers all states corresponding to that region (new_tree in lines 1 and 30). Also, from the definition of splitting trees it follows that splitting a region results in subregions of which the union corresponds to the original region.

Also, from the definition of splitting trees it follows that

\[
\forall r, r' \in R . r \neq r' \Rightarrow \{r\} \cap \{r'\} = \emptyset \tag{6.14}
\]

This means that \( R \) is a true partition of the partial state space.

Correctness of initial attribute and init_count value. The following must hold

\[
\forall r \in R . (\text{initial}(r) \iff \exists s \in \{r\} . s \in S_0) \tag{6.15}
\]

\[
\text{init}_{\text{count}} = \vert \{r \in R \mid \text{initial}(r) \land \text{val}(r)(\phi_t) = U\} \vert \tag{6.16}
\]

where \( \phi_t = \text{init} U(\psi) \) is the "top" property. Initially, the initial region contains all initial states. Therefore the initial attribute of that region is initially set to true (line 2), while for new regions, it is initialized to false (line 31). The only occasion at which the initial attribute is affected is when a region is split, which only occurs in the split procedure. When splitting, initial attributes of the new subregions are first set to false. If the initial attribute of the region that was split was set, then it is checked using the reduce_par procedure, which of the two subregions contain initial states (lines 141 – 149).

If reduce_par\((r, c)\) returns split, then \( c \) represents a "true" split of the set of initial states in \( r \), because it splits the set of parameter valuations represented by \( r \). Therefore, in that case both subregions are marked as initial. The cases for sat and dissat are analogous.

Also init_count is initialized to 1, which is obviously correct. From the correctness of the split procedure it again follows directly that splitting preserves the validity of equation 6.16. Only if splitting resulted in two initial subregions, then the number of initial regions will increase with one. Furthermore, the validity of equation 6.16 could be affected whenever \( \phi_t \) is decided for an initial region. It is easy to verify that in propagate_values (line 240) and decide_cycles (line 258), init_count is appropriately decreased whenever this is the case.

Induced partial region graph. The partial partition induces a partial region graph (as in definition 6.1). Let \( R_0 = \{r \in R \mid \text{initial}(r)\} \). Let \( D^* \) be a set of edges such that

\[
\forall r, r' \in R . \langle (\text{phase}(r) = \text{active} \land \exists s \in \{r\}, s' \in \{r'\}, e \in E \setminus \text{unexp}(r). s \xrightarrow{e} T s' \rangle \iff r \xrightarrow{e} D^* r' \rangle
\]

\[
\wedge \langle (\text{phase}(r) = \text{active} \land \exists s \in \{r\}, s' \in \{r'\}, e \in E . s \xrightarrow{e} T s' \rangle \iff r \xrightarrow{\tau} D^* r' \rangle
\]

\[
\tag{6.17}
\]

Let \( R_U = \{r \in R \mid \text{phase}(r) \neq \text{active} \lor \text{unexp}(r) \neq \emptyset\} \). Then

\[
RG^* = \langle R, R_0, D^*, R_U \rangle \text{ is a correct partial region graph for } M \tag{6.18}
\]

The correctness of \( R \) (with respect to definition 6.1) follows from equation 6.13. The correctness of \( R_0 \) follows from the correctness of the initial attribute in equation 6.15. Also, from the definition of \( D^* \) it follows immediately that \( D^* \) satisfies the conditions for \( E_R \) in definition 6.1. Finally, for \( R_U \), the condition on \( R_U \) in definition 6.1 means that a region \( r \) has to be in \( R_U \) whenever there is a state \( s \in \{r\} \) which has an outgoing transition to a state \( s' \), such that there is no region \( r' \) with \( s' \in \{r'\} \) such that \( r \rightarrow D^* r' \). From equation 6.17 and the definition of \( R_U \) it follows that this is the case.

Self-looping time edges are not stored. For that reason, we will use a transition relation
$D$, which is defined as follows.

$$D = \{ (r, x, r') \in D^* \mid x = \tau \Rightarrow r \neq r' \}$$

**Correctness of the urgent attribute.** Urgency is important whenever time edges are taken into account, which is only for time and active regions. Therefore, the following must hold

$$\forall r \in R . \ phase(r) \in \{ \text{time}, \text{active} \} \Rightarrow
\exists e \in E . (e \in \text{unexp}(r) \lor \exists r' . r \xrightarrow{e} r') \iff \text{urgent}(r)$$

(6.19)

Regions can only be given phase time in the guard_split procedure (line 88). This only happens if such regions respect the guards of all enabled outgoing XTG edges. Whenever in this procedure a region turns out to respect and satisfy the guard of an XTG edge, it is checked whether this edge is urgent and if so, the region is marked as urgent (line 104). Thus a region $r$ that enters the time phase is marked urgent if and only if there is urgent edge in $\text{urgent}(r)$. Also, a region can only enter the active phase when coming from the time phase. Furthermore, when splitting a region, the urgency attribute is inherited by all children (line 137), which is correct because all time and active regions respect the guards of all outgoing edges. As a consequence, the urgent attribute identifies exactly those time and active regions in which all states in which an urgent XTG edge is enabled.

**Correctness of the noaccess attribute.** It is the sole task of the invariant split procedure to ensure that the noaccess attribute is correctly filled for each new region. This attribute is needed to avoid edges to inaccessible regions. Therefore, whenever edges are added to the graph, the following equation must hold:

$$\forall r \in R . (\llbracket r \rrbracket \cap \llbracket I(location(r)) \rrbracket \neq \emptyset \iff \text{noaccess}(r))$$

(6.20)

Immediately when a new region is introduced, it is subjected to the invariant_split procedure (line 4 and 33). It is easy to show that as a result of invariant_split$(r, I(location(r)))$, all subregions respect the invariant of location$location(r)$ and that for exactly those subregions in which the invariant is violated noaccess$\text{noaccess}(r) = \text{true}$.

**Correctness of the reachable attribute.** The following must hold.

$$\forall r \in R . \ (\text{reachable}(r) \Rightarrow \exists r_i \in R . (\text{initial}(r_i) \land r_i \xrightarrow{\text{D}} r))$$

(6.21)

$$\forall r \in R . \ (\text{reachable}(r) \iff \text{initial}(r) \lor \text{stab}(r) \neq \emptyset)$$

(6.22)

The first equation is proved by induction. Initially, for all regions, the reachable attribute is set to false. We then prove by induction that assigning reachable$\text{reachable}(r)$ is correct assuming that currently for all regions, equation 6.21 is satisfied. In split, the reachability attribute is set in case for a region initial$\text{initial}(r) = \text{true}$ (lines 140 – 149), which is correct according to equation 6.21. Furthermore, the reachable attribute is set in prop_split_D and prop_split_T (lines 172 and 199). Using the induction hypothesis, it follows immediately that there the reachable attribute is set correctly. The validity of equation 6.22 follows immediately, since the reachable$\text{reachable}(r)$ attribute is only set for initial regions or in case an incoming edge is added to stab$\text{stab}(r)$, and edges are never removed from a region (as long as it is a leaf region).

**Correctness of the bound attribute.** Correctness of the bound attribute is expressed as follows.

$$\forall r \in R . \ (\text{phase}(r) = \text{active} \land \neg \text{urgent}(r)) \Rightarrow
\text{bound}(r) = \bot \iff r \text{ is unbounded}) \land
\text{bound}(r) = c \iff r \text{ has upper bound } c)$$

(6.23)

Regions become active only as a result of completing the time_split procedure. From the correctness of that procedure it follows that on becoming active, the bound attribute of a non-urgent region is correct. When splitting an active region, the bound attribute is
copied to the subregions (line 137). This is correct unless the region is split by a bound. Therefore, each active non-urgent region $r$ that is split, is submitted to $prop\_split\_T I(r)$. From correctness of that procedure it follows that as a result all subregions of $r$ again satisfy equation 6.23.

**Correctness of $atom\_split$ and $fair\_split$.** After calling $atom\_split(r, P)$ the following holds.

$$\forall r' \in sub(r) . (reachable(r') \iff phase(r') = EDGES)$$

$$\land (reachable(r') \Rightarrow \forall p \in P . \forall s \in \{r'\} . ((val(r')(p) = T \iff s \models p) \land (val(r')(p) = F \iff s \not\models p)))$$

(6.24)

Thus all reachable subregions of $r$ respect the atomic properties, and the valuation is adapted to reflect the values of the atomic properties. Also, all such regions are promoted to the $EDGES$ phase. Similarly, after calling $fair\_split(r, F)$ the following holds

$$\forall r' \in sub(r) . (reachable(r') \iff phase(r') = ATOM)$$

$$\land (reachable(r') \Rightarrow \forall f \in F . \forall s \in \{r'\} . ((f \in fval(r') \iff s \models f) \land (f \not\in fval(r') \iff s \not\models f)))$$

(6.25)

The above properties follow relatively simply from the algorithm descriptions.

**Correctness of $guard\_split$.** After calling $guard\_split(r, E)$ the following holds.

$$\forall r' \in sub(r) . (reachable(r') \iff phase(r) = TIME) \land (reachable(r') \Rightarrow (\forall e \in E . \forall s \in \{r'\} . (e \in unexp(r') \iff e \models P))$$

(6.26)

Again the proof is relatively simple.

**Characterizing correctness of edges.** Before going into the correctness of edges, some equations are defined that will be used for characterizing correctness of edges.

First of all, the following two equations express correctness and stability of discrete edges, respectively, for $e$-edges from subregions of $r$ to subregions of $r'$.

$$\forall rr \in sub(r) . \forall rr' \in sub(r') . (rr \xrightarrow{D} rr' \Rightarrow$$

$$\exists rrr . rr \in sub(rrr) \land \langle rrr, e, \bot \rangle \in stab(rr') \lor$$

$$\exists rrr' . rr' \in sub(rrr') \land \langle rrr', e, \bot \rangle \in instab(rrr))$$

(6.27)

$$\forall rr \in sub(r) . \forall \langle rr', e, \bot \rangle \in instab(rr) . stab^*([rr], e, [rr']) \land$$

$$\forall rr' \in sub(r') . \forall \langle rr, e, \bot \rangle \in stab(rrr) . stab ([rr], e, [rrr'])$$

(6.28)

The first equation states that all edges in the induced partial region graph between $r$ and $r'$ should somehow be represented in the data structures of the algorithm, while the second equation states that all edges that are represented should correspond to stable edges — where for edges in $instab$ an exception is made for inaccessible states. Analogously, the following two equations do the same for time edges from subregions from $r$ to $r'$.

$$\forall rr \in sub(r) . \forall rr' \in sub(r') . (rr \xrightarrow{T} rr' \Rightarrow$$

$$\exists rrr . rr \in sub(rrr) \land \langle rrr, \tau, c \rangle \in stab(rrr') \lor$$

$$\exists rrr' . rr' \in sub(rrr') \land \langle rrr', \tau, c \rangle \in instab(rrr))$$

(6.29)

$$\forall rr \in sub(r) . \forall \langle rr', \tau, c \rangle \in instab(rr) . stab^*([rr], \tau, [rr']) \land$$

$$\forall rr' \in sub(r') . \forall \langle rr, \tau, c \rangle \in stab(rrr) . stab ([rr], \tau, [rrr'])$$

(6.30)

We will use the following equation to express the correctness of bound attributes of subregions of $r$.

$$\forall rr \in sub(r) . bound(rr) = \begin{cases} \bot & \text{if } rr \text{ is unbounded} \\ c & \text{if } rr \text{ has upper bound } c \end{cases}$$

(6.31)

**Correctness of $time\_split$.** Let $r_{top} = get\_tree(location(r))$. After calling $time\_split(r)$, equations 6.29 and 6.30 hold for $r$ and $r_{top}$. In other words, all subregions of $r$ have
the correct outgoing time edges. Furthermore, equation 6.31 holds for \( r \), and also \( \forall rr \in \text{sub}(r), \text{phase}(rr) = \text{active} \).

For the proof, some dedicated notation is needed. Given two region representations \( x = \langle l, n_x \rangle \) and \( t = \langle l, n_t \rangle \), such that \( x \in \text{sub}(t) \), we write \( x^t \) to denote the region that is interpreted by only taking into account the subtree under \( t \), thus not including the constraints that are above \( t \). Formally \( F[x^t] = \langle l, S[x^t] \rangle \). Remember that \( S[x]^n_t \) denotes the constraint associated with \( x \) when only taking the subtree under \( n_t \) into account (see definition 5.2).

We first prove the following property of \( \text{time\_split*} \). For region representations \( x, d \) and \( t \) such that \( x \in \text{sub}(t) \) and \( d \in (\text{sub}(t) \cup \{ \bot \}) \), if

\[
\begin{align*}
(d \neq \bot & \Rightarrow \text{stable}^* \left( \left[ x^t \right],\tau,[d^t] \right) \land \text{bound}(x) \text{ is upper bound of } x^t \right) \land \\
(d = \bot & \Rightarrow \left[ x^t \right] \text{ is unbounded} \land \text{bound}(x) = \bot) \\
\end{align*}
\]

(6.32)

then after calling \( \text{time\_split*}(x,d,t) \), equations 6.29 and 6.30 hold for \( x \) and \( r_{top} \) and equation 6.31 holds for \( x \).

This is proved by induction, which is done by showing that the above property holds in case \( \text{parent}(t) = \bot \) (base case), and that it holds in case \( \text{parent}(t) \neq \bot \) if we assume that it holds already for \( \text{time\_split}^*(x',d',\text{parent}(t)) \) for any \( x' \) and \( d' \) such that \( x' \in \text{sub}(t) \) and \( d' \in \text{sub}^*(\text{parent}(t)) \cup \{ \bot \} \) (induction case). The induction ends because splitting trees are finite.

- We start with the base case. If \( \text{parent}(t) = \bot \) (line 133), then \( t = r_{top} \), which means that \( x = x^t \) and \( d = d^t \). If \( d \neq \bot \) then it follows from equation 6.32 that \( \text{stable}^* \left( \left[ x^t \right],\tau,[d^t] \right) \) and that \( \text{bound}(x) \) is correct. From stability follows that \( x \) cannot have time edges to other regions than \( x \). \( \text{prop\_split\_T}(x,\text{bound}(x),d) \) is called (line 135) and from the correctness of that procedure it follows that equations 6.29 and 6.30 hold for \( x \). On the other hand, if \( d = \bot \) then \( \left[ x^t \right] \) is unbounded (equation 6.32), in which case there are no outgoing time edges, and the equations are trivially satisfied. Furthermore, equation 6.31 follows from the assignments in lines 120 and 127, and equation 6.32.

- We continue with the induction case (\( \text{parent}(n) \neq \bot \), line 112). Lines 112 – 115 check whether or not \( \text{cons}(\text{parent}(t)) \) represents an upper bound for \( x^t \). If so, then the upper bound is stored in \( \text{cons} \) and the region that is the other child of \( \text{parent}(t) \) is stored in \( \text{nd} \). First consider the case that \( c = \text{cons}(\text{parent}(t)) \) does not represent an upper bound for \( x^t \) (line 117). Then if \( d = \bot \), then \( \left[ x^t \right] \) is unbounded (equation 6.32), which means that \( \left[ x^{\text{parent}(t)} \right] = \left[ x^t \right] \cap [c] \) is also unbounded (since \( c \) is not an upper bound). If \( d \neq \bot \) then it easy to show that from \( \text{stable}^* \left( \left[ x^t \right],\tau,[d^t] \right) \) it follows that \( \text{stable}^* \left( \left[ x^t \right] \cap [c],\tau,[d^t] \cap [c] \right) \), which means that \( \text{stable}^* \left( \left[ x^{\text{parent}(t)} \right],\tau,[d^{\text{parent}(t)}] \right) \). For both cases this means that equation 6.32 also holds for \( x, d \), and \( \text{parent}(t) \), since also \( \text{bound}(x) \) remains the same. From the induction hypothesis then follows that calling \( \text{time\_split*}(x,d,\text{parent}(t)) \) (line 117) has the effect that required equations hold for \( x \).

- We continue with the case that there is an upper bound \( \text{cons} \) for \( x^t \) (line 119). If \( d = \bot \) (line 120), then \( \left[ x^t \right] \) is unbounded (equation 6.32), which means that \( \text{cons} \) is the upper bound of \( x^{\text{parent}(t)} \) and thus that \( \text{stable}^* \left( \left[ x^t \right] \cap \{ \text{cons} \},\tau,[\neg \text{cons}] \right) \). This means that \( \text{stable}^* \left( \left[ x^{\text{parent}(t)} \right],\tau,[\text{nd}^{\text{parent}(t)}] \right) \) and that equation 6.32 holds for \( x, \text{nd} \), and \( \text{parent}(t) \), \( \text{time\_split*}(x,\text{nd},\text{parent}(t)) \) is called (line 121), and the proof of our equations again follows from applying the induction hypothesis.

- If \( d \neq \bot \) (line 123) then we know from equation 6.32 that \( \text{stable}^* \left( \left[ x^t \right],\tau,[d^t] \right) \).
Then it follows from theorem 5.5 that \textbf{stable*} \((\mathbf{r'}, \mathbf{e}, \mathbf{r'})\) then after calling \textit{prop_split}* \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\), equations 6.27 and 6.28 hold for \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\). This is proved using an inductive argument. We prove that the equations hold for \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\) if we assume that for any \((\mathbf{r}_x, \mathbf{e}, \mathbf{r}'_x)\) such that \(r_x \subseteq \text{sub}(r), r'_x \subseteq \text{sub}(r')\) but not both \(r_x = r\) and \(r'_x = r'\), the property already holds. Because splitting trees are finite, our property then follows by induction. This means we have to prove that equations 6.27 and 6.28 hold after \textit{prop_split}* \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\) with \textbf{stable*} \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\) assuming that for any recursive call to \textit{prop_split}* \((\mathbf{r}_x, \mathbf{e}, \mathbf{r}'_x)\) with \textbf{stable*} \((\mathbf{r}_x', \mathbf{r}_x')\), the equations already hold.

- If \(r\) is not a leaf region (lines 152 – 154), then \textit{prop_split}* is called for subregions of \(r\). From \textbf{stable*} \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\) it follows using the definition of stability that \textbf{stable*} \((\mathbf{left(r)}, \mathbf{e}, \mathbf{r'})\) and \textbf{stable*} \((\mathbf{right(r)}, \mathbf{e}, \mathbf{r'})\). From the induction hypothesis and the fact that \(\text{sub}(r) = \text{sub}(\text{left}(r)) \cup \text{sub}(\text{right}(r))\) it follows that after line 154, equations 6.27 and 6.28 hold.

- If \(r\) is a leaf region and not reachable (line 174), then the stable edge \(r \rightarrow r'\) is added to \textit{instab}(r). It follows from \textbf{stable*} \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\), that both equations hold.

- If \(r\) is a reachable leaf region (line 155), then if \(r'\) is not split, the stable edge is added to \textit{stab}(r') provided that \(r'\) is accessible (line 171). From \textit{reachable}(r) it follows that \textit{noaccess}(r), and since \textit{noaccess}(r') it follows that \textbf{stable} \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\), and thus equation 6.28 holds. Equation 6.27 follows immediately. If \textit{noaccess}(r), then the edge is illegal and nothing needs to be done.

- If \(r'\) is split, we arrive at line 159. From \textbf{stable*} \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\) and theorem 5.2 we know that \textbf{stable*} \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\) and that \textbf{stable*} \((\mathbf{r}, \mathbf{e}, \mathbf{r'})\)
[\neg \text{cons}(r')] \). From the definition of reduce it follows that if reduce(r, c) = split then both \([r] \cap [c]\) and \([r] \cap \neg [c]\) are feasible regions. From the definition of split it follows that right(r) implements \([r] \cap [c]\) and left(r) implements \([r] \cap \neg [c]\). As a consequence, stable\(([[\text{left}(r)], e, [[\text{left}(r')]])\) and stable\(([[\text{right}(r)], e, [[\text{right}(r')]])\), which means that by calling prop\_split\_D for these edges, the induction hypothesis applies. Equation 6.28 then follows immediately, while equation 6.27 can be shown to hold using the induction hypothesis and the fact that there can be no edges from left\(r') to right\(r') and from right\(r') to left\(r). The cases for sat and dissat follow from similar reasoning.

**Correctness of prop\_split\_T.** If \(\forall rr \in \text{sub}(r) . (\text{bound}(rr) = c \Rightarrow \text{stable\(([[rr], \tau, [rr']])))\) then after calling prop\_split\_T\((r, c, r')\), equations 6.29 and 6.30 hold for \(r\) and \(r'\).

This is again proved using an inductive argument. We prove that equations 6.29 and 6.30 hold for \((r, \tau, r')\) if we assume that for any \(r_x, \tau, r'_x\) such that \(r_x \in \text{sub}(r), r'_x \in \text{sub}(r')\) but not both \(r_x = r\) and \(r'_x = r'\), these equations already hold.

- If \(r\) is not a leaf region (lines 175 – 177), then prop\_split\_T is called for subregions of \(r\). From the precondition it follows directly that \(\forall rr \in \text{sub}(\text{left}(r)) . \text{bound}(\text{left}(r)) = c \Rightarrow \text{stable\(([[\text{left}(r)], \tau, [\text{left}(r')]])\)) and \(\forall rr \in \text{sub}(\text{left}(r)) . \text{bound}(\text{right}(r)) = c \Rightarrow \text{stable\(([[\text{right}(r)], \tau, [\text{right}(r')]])\)). From the induction hypothesis and the fact that \(\text{sub}(r) = \text{sub}(\text{left}(r)) \cup \text{sub}(\text{right}(r))\) it follows that after line 177, equations 6.29 and 6.30 hold.

- Arriving at line 179, an outgoing edge from \(r\) to a region in \(r'\) must have \(c\) as separating constraint, which means that \(c\) must be the bound of \(r\). Thus, if this is not the case, then there are no time edges in \(D\) from \(r\) to subregions of \(r'\).

- If \(r\) is not reachable, then the representation of the stable edge \(\overrightarrow{r}D\) is added to instab\((r)\) (line 201). Since \(\text{bound}(r) = c\) and thus \(\text{stable\(([[rr], \tau, [rr']])))\), the equations hold.

- We continue at line 181, where we know that \(r\) is reachable and not split. If \(r'\) is not split either, then the stable edge is added to \(\text{stab}(r')\) (line 198). Analogous to the case for prop\_split\_D, it follows that the two equations hold.

- Arriving at line 183, from theorem 5.5 we know that propagation of a non-bound constraint results in a split with the same constraint (line 186). For a bound constraint, prop\(T\)(cons\((r), c\)) gives the desired propagated constraint (line 184). Thus \(\text{stable\(([[r] \cap [e'], \tau], [r'] \cap [\text{cons}(r')])))\) and \(\text{stable\(([[r] \cap \neg [e'], \tau], [r'] \cap [\neg \text{cons}(r')])))\).

For lines 187 – 195 the line of reasoning is analogous to that for prop\_split\_D.

**Correctness of prop\_split\_TI.** If a region \(r\), with \(\neg \text{urgent}(r)\) is split into two leaf subregions then after calling prop\_split\_TI\((r)\), equations 6.29 and 6.30 hold for \(r\) and \(r'\). In other words, all time edges between subregions of \(r\) are correct and stable. Also, if equation 6.31 holds for \(r\), then after calling prop\_split\_TI\((r)\), this equation will still hold for \(r\).

This is proved as follows.

- Lines 202 – 207 investigate the type of constraint that splits \(r\). If this constraint is a non-bound then there will be no time edges between the subregions of \(r\) (theorem 5.6), and nothing needs to be done (line 207). Otherwise, temporary variables \(r_1, r_2\) and \(c\) are given values such that \(c\) contains the separating constraint of \(r_1\) and \(r_2\) in the form of an upper bound, and \(r_1\) holds the subregion of \(r\) that satisfies \(c\), and \(r_2\) holds the subregion that dissatisfies \(c\). Thus, there are potential time edges from \(r_1\) to \(r_2\).

- If \(r\) was an unbounded region (i.e. it has no upper bound, expressed by the fact \(\text{bound}(r_1) = \bot\), line 208) then there is a stable time edge from \(r_1\) to \(r_2\) (theorem 5.6).
Depending on reachability of \( r_1 \) this edge has to be added to \( \text{instab}(r_1) \) or \( \text{stab}(r_2) \). This is achieved by calling \( \text{prop}\_\text{split} \_T(r_1, c, r_2) \) (line 209). Because \( r \) and thus \( r_1 \) and \( r_2 \) are non-urgent, it follows that \( \text{stable}^* (r_1, \tau, r_2) \). From the correctness of \( \text{prop}\_\text{split} \_T \) it then follows that equations 6.29 and 6.30 are satisfied after line 209.

- Continuing with line 211, if \( r \) does have an upper bound (stored in \( \text{bound}(r_1) \)), then according to theorem 5.6, \( \text{stable}^* ([r_1, \tau, r_2]) \). Reduction is performed to check whether or not splitting is really needed. If reduction returns \( \text{split} \), \( r_1 \) is split and as a consequence, \( \text{stable}^* ([\text{left}(r_1), \tau, r_2]) \). Then \( \text{prop}\_\text{split} \_T(\text{left}(r_1), c, r_2) \) ensures that equations 6.29 and 6.30 are satisfied. The case for \( \text{dissat} \) follows from similar reasoning. Reduction cannot return \( \text{sat} \) because that would mean that \( c \) was not a splitting constraint for \( r \), which cannot happen.

- The correctness of the bound attribute (equation 6.31) can easily be verified (lines 214 and 218).

**Correctness of \( \text{split} \).** Given a (leaf) region \( r \) and a constraint \( c \), as a result of \( \text{split}(r, c) \),\( r \) is split into subregions such that all states in all subregions of \( \text{left}(r) \) dissatisfy \( c \) and all states in all subregions of \( \text{right}(r) \) satisfy \( c \). This follows directly from line 137.

First of all, splitting an (active) non-urgent region may introduce new time edges between the new subregions. In that case always \( \text{prop}\_\text{split} \_T \) is called. It then follows from the correctness of that procedure that all time edges between subregions are correctly added.

This leaves us to deal with existing incoming or outgoing edges of the region that is split. We want to show the following properties. If for some \( r \rightarrow_{\tau} r' \), equations 6.27 and 6.28 hold, then after calling \( \text{split}(r) \) or \( \text{split}(r') \), this is still the case. Also, if for some \( r \rightarrow_{\tau} r' \), equations 6.29 and 6.30 hold, then after calling \( \text{split}(r) \) or \( \text{split}(r') \), this is still the case. Thus, splitting a region, retains the correctness of incoming and outgoing edges.

To show this, we start with incoming edges. Suppose \( r \) is split, resulting in subregions \( r_1 \) and \( r_2 \). There are two cases for an incoming edge.

- If before splitting \( \exists rr . r \in \text{sub}(rr) \land \langle rr, x, \bot \rangle \in \text{instab}(r') \) then the property will still hold since also \( r_1, r_2 \in \text{sub}(rr) \).
- If before splitting \( \exists rr' . r' \in \text{sub}(rr') \land \langle rr', c, \bot \rangle \in \text{stab}(r) \) then in the \( \text{split} \) procedure \( \text{prop}\_\text{split} \_D(r', x, r) \) or \( \text{prop}\_\text{split} \_T(r', x, r) \) will be called. The property then follows from the correctness of these procedures.

Now for outgoing edges, if before splitting there was an edge \( r \rightarrow_{\tau} D r' \), then there are two cases, since the current edge can be stored in \( \text{instab}(r) \) or \( \text{stab}(r') \):

- If before splitting \( \exists rr' . r' \in \text{sub}(rr') \land \langle rr', c, \bot \rangle \in \text{instab}(r) \) then the same will hold for the new subregions of \( r \), since when splitting, the \( \text{instab} \) attribute is copied to the subregions (line 137).
- If before splitting \( \exists rr . r \in \text{sub}(rr) \land \langle rr, c, \bot \rangle \in \text{instab}(r') \) then the property will still hold for the new subregions \( r_1 \) and \( r_2 \) of \( r \) since \( r_1, r_2 \in \text{sub}(rr) \).

**Stability of discrete edges.** All discrete edges stored in \( \text{stab} \) and \( \text{instab} \) attributes are stable:

\[
\forall r \in R \land \forall \langle r', e, c, \bot \rangle \in \text{instab}(r) . \text{stable}^* ([r], e, [r']) \\
\land \forall \langle r', e, c, \bot \rangle \in \text{stab}(r) . \text{stable} ([r'], e, [r]) \tag{6.33}
\]

Initially, for each region \( r \), \( \text{stab}(r) \) and \( \text{instab}(r) \) are empty. Now consider each occasion at which discrete edges are added to \( \text{stab}(r) \) or \( \text{instab}(r) \). In \( \text{prop}\_\text{split} \_D \) discrete edges are added to \( \text{stab} \) and \( \text{instab} \). From the correctness of \( \text{prop}\_\text{split} \_D \) it follows that this
procedure adds only edges that satisfy equation 6.33. Furthermore, in \( split(r, c) \), \( instab(r) \) is copied to its subregions. From the definition of stability it follows immediately that (for discrete edges) stability is retained.

**Stability of time edges.** All \( stab \) and \( instab \) attributes define stable edges:

\[
\forall r \in R . \forall \langle r', \tau, c \rangle \in instab(r) . \{(bound(r) = c \Rightarrow stab\{[r], \tau, [r']\})
\wedge \forall \langle r'', \tau, c \rangle \in stab(r) . \{(bound(r) = c \Rightarrow stab\{[r''], \tau, [r']\})
\]  

(6.34)

Initially, for each region \( r \), \( stab(r) \) and \( instab(r) \) are empty. Only in \( prop\_split\_T \) time edges are added to \( stab(r) \) and \( instab(r) \). From the correctness of \( prop\_split\_T \) it follows that each edge that is added satisfies equation 6.34. Furthermore, in \( split(r, c) \), \( r \) is split, which may result in subregions that are instable with respect to time edges. This is solved by the call to \( prop\_split\_TI(r) \), which ensures that subregions of \( r \) become stable with respect to time edges that are internal to \( r \). Also, existing outgoing time edges in \( instab(r) \) are copied to the subregions. For these regions, the validity of equation 6.34 is retained (thanks to the \( bound(r) = c \) conditions in that equation).

**Presence of discrete edges.** We have to show that each edge in the partial region graph is represented in the data structures of the algorithm. At the end of each iteration (line 39), the following equation always holds.

\[
\forall r, r' \in R . \forall e \in E . \langle r, \tau, c \rangle \rightarrow_D r' \Rightarrow
\begin{align*}
& \exists rr' . r' \in sub(rr') \land \langle rr', e, \bot \rangle \in instab(r) \lor \\
& \exists rr . r \in sub(rr) \land \langle rr, e, \bot \rangle \in stab(r'))
\end{align*}
\]  

(6.35)

Thus, each edge in \( D \) is represented in either the \( instab \) attribute of the source or the \( stab \) attribute of the destination.

Initially, equation 6.35 holds since \( D \) is empty as no active regions exist. We can now assume that at the beginning of the iteration the above equation holds, and investigate the events in the iteration that might cause the equation to become temporarily violated.

- First of all, a region can become active, which means that its outgoing edges are taken into account in \( D \) (equation 6.17). Due to the way the \( phase \) attribute is administrated, a region always first enters the \( GUARDS \) phase. From there it may go to \( TIME \) and eventually to \( ACTIVE \). When a region \( r \) enters the \( GUARDS \) phase (line 16), all edges leaving from the location associated with \( r \) are added to \( unexp(r) \). For such a region, \( guard\_split(r) \) is then called. From the correctness of that procedure it follows that for each subregion \( r' \in sub(r) \) that goes to \( TIME \), \( unexp(r') \) contains all XTG edges that are enabled in states of \( r' \). Regions in the \( TIME \) phase are subjected to the \( time\_split \) procedure which results in subregions \( r' \) which have the same set of discrete edges in \( unexp(r') \) as \( r \) had. In the \( time\_split \) procedure, the phase of resulting regions is set to \( ACTIVE \). Thus when a region \( r \) enters the \( ACTIVE \) phase, all outgoing and enabled XTG edges are stored in \( unexp(r) \). This means that a region that becomes \( ACTIVE \) initially has no outgoing edges in \( D \).

- In line 26, an edge \( e \) is removed from \( unexp(r) \), which means that potentially outgoing \( e \)-edges from \( r \) are added to \( D \). Then \( prop\_split\_D(r, e, r') \) is called (line 36), where \( r' \) is the initial region for the destination location of \( e \) (lines 28 and 30). From the correctness of \( prop\_split\_D \) it then follows that equation 6.35 is again satisfied. Also, this might cause further propagated splitting. Assuming that such propagation terminates it follows from the correctness of the \( split \) and propagation procedures that equation 6.35 remains valid.

- The only occasion at which discrete edges are removed from \( instab(r) \) is in line 24. In that case again \( prop\_split\_D \) is called, and the same argument as above holds.
Finally, a region \( r \) (with \( \text{phase}(r) = \text{ACTIVE} \)) may become split as a consequence of ensuring respect and enabling stability (in \text{atom\_split}, \text{fair\_split} or \text{guard\_split}). From the correctness of the \text{split} procedure it follows that as a result equation 6.35 will be satisfied. The same argument as above applies to the propagation of splits.

**Presence of time edges.** At the end of each iteration (line 39), the following equation always holds.

\[
\forall r, r' \in R. ((r \rightarrow_D r') \Rightarrow ((\exists r'' . r'' \in \text{sub}(rr') \land (rr', \tau, -) \in \text{instab}(r)) \lor (\exists rr . r \in \text{sub}(rr) \land (rr, \tau, -) \in \text{stab}(r'))) \quad (6.36)
\]

Initially, there are no regions that are \text{ACTIVE}. Again, we can assume that at the beginning of the iteration the above equation holds, and investigate the events in the iteration that might cause the equation to become temporarily violated.

- Only in \text{time\_split} regions are assigned \text{ACTIVE}. From the correctness of \text{time\_split} it follows that when regions become \text{ACTIVE}, they satisfy equation 6.36.
- The only occasion at which for an \text{ACTIVE} region \( r \) time edges are removed from \( \text{instab}(r) \) is in line 24. In that case \text{prop\_split\_T} is called for that edge. From correctness of \text{prop\_split\_T} it then follows that in this case equation 6.36 still holds.
- An active region may become split, in which case a similar argument as for discrete edges can be made. There is an additional complexity because when splitting a region, time edges between subregions may have to be added, but this also follows from the correctness of \text{split}.

**The sets \( S \) and \( C \).** The set \( S \) is needed to keep track of all regions that still need processing. The set \( C \) keeps track of fully explored regions that could not yet be decided. This means that the two sets should meet the properties stated below.

A subspace is uniquely identified by the set of extended subproperties that should be decided for regions in that subspace — any two regions \( r_1 \) and \( r_2 \) are in the same subspace if and only \( \text{props}(r_1) = \text{props}(r_2) \). Each subspace is dealt with using dedicated sets \( S \) and \( C \). Let \( R_C \subseteq R \) denote the set of regions that belong to the current subspace. Then

\[
\forall r \in R_C . ((\text{reachable}(r) \land \neg \text{decided}(r) \land (\text{unexp}(r) \neq \emptyset \lor \text{instab}(r) \neq \emptyset)) \Rightarrow (\exists rr . (r \in \text{sub}(rr) \land rr \in S)) \quad (6.37)
\]

Initially \( S \) is empty. Each region that becomes reachable is always added to \( S \) (lines 1 and 37). Furthermore, regions are removed from \( S \) if they are decided (line 11), or fully explored — meaning that there are no unexplored edges and no edges in \( \text{instab} \) (line 21). Also, in line 10 a region is replaced by its reachable subregion. It is easy to see that all these events do not influence the validity of equation 6.37.

For termination issues, we need a progress property: *If all reachable and undecided regions in \( S \) are fully explored and stable, then ultimately \( S \) will become empty.* This property follows immediately when inspecting the main loop — in particular lines 20 – 21.

For \( C \) the following condition holds.

\[
\forall r \in R_C . ((\text{reachable}(r) \land \neg \text{decided}(r) \land \text{unexp}(r) = \emptyset \land \text{instab}(r) = \emptyset)) \Rightarrow (\exists rr . (r \in \text{sub}(rr) \land rr \in (S \cup C))) \quad (6.38)
\]

Initially \( C \) is empty. Each fully explored region (i.e. \( \text{unexp}(r) = \emptyset \land \text{instab}(r) = \emptyset \)) is transferred from \( S \) to \( C \) (line 21), while regions are never removed from \( C \). From this fact it follows, together with equation 6.37, that equation 6.38 must hold. Note that splitting a region does not change the validity of the equations.

For regions that correspond to a subspace of which the sets \( S \) and \( C \)'s are stored on the stacks \( \text{Stk}_S \) and \( \text{Stk}_C \), the analogous properties can be shown to hold.
Local decision making (propagate_values). We have to show that decisions are correct and that decisions are made as soon as possible. Starting with type 1, whenever a new location is explored, its regions are submitted to the atom_split procedure. From the correctness of that procedure it follows that for each reachable subregion, all atomic properties are correctly decided. Exactly those subregions are promoted to the edges phase. Since any further processing of regions requires that these regions go through the edges phase, it follows that these decisions are made as soon as possible. Continuing with the type 2 and 3 decisions, the propagate_values procedure checks if, as a consequence of decisions stored in $\mathcal{V}$, other $\langle r, \phi \rangle$ combinations can be decided. It is easy to show that the decisions in that procedure are correct with respect to the rules in figure 6.1.

Global decision making (decide_cycles). Global decision making builds on the fact that the set $\mathcal{C}$ is correct, once $\mathcal{S}$ is empty. In particular it requires that all regions in $\mathcal{C}$ are fully explored and that $\mathcal{C}$ holds the complete set of regions that could not yet be decided. This follows from equations 6.37 and 6.38.

Lines 241 – 244 take care of finding the fair SCC’s in $\mathcal{C}$. The correctness of this procedure is not discussed here since it is a minor variation of Tarjan’s algorithm [100]. Thus, at line 245, $FC$ holds the set of fair SCC’s in $\mathcal{C}$. Subsequently, the loop in lines 247 – 257 decides the values of all regions in $FC$, one temporal subproperty at the time. The correctness of the decisions in lines 249 – 252 then follows directly from the appropriate type 4 decision rules. After that, values are propagated (line 253). Having done that, for the remaining regions, the interference free-cycle rules apply (the first, second and final type 4 rules). For example, consider the case for $EU$. Let $D$ denote the remaining undecided regions. Because all regions are explored and the local decision rules did not force a decision, it follows that for each $r \in D$, $r \notin R_U$ and $val(r)(\phi_2) = T$, and $\forall r \in D : \forall r' \in Suc_S(r) \setminus D, val(r')(\phi_1 \land EU \phi_2) = F$. To see that all regions in $D$ can be decided $F$, we could find the set of strongly connected components in $D$ and decide these components using the first type 4 rule, one at the time. Because of the definition of strongly connected components, each time a strongly connected component can be taken from $D$ which has no undecided external successors in $D$ and thus can be decided.

Termination. As was discussed already, termination can usually not be guaranteed. At best, the following property can be proven to hold: If a finite partition exists in which all reachable regions are stable and in which all regions respect all atomic properties and fairness predicates, then the algorithm will terminate.

In principle, this can be proved by showing that the algorithm does not perform unnecessary splits. Most of the time, this is the case, but there are some cases that do not adhere to that rule.

- In prop_split_TI$(r)$, $r$ is split when needed, regardless of its reachability. This always concerns a single split. Avoiding this would introduce complexity in the algorithm, while this concerns only a sporadic case.
In guard_split, enabling stability is enforced for each outgoing edge, before exploring the edges itself. It may be that outgoing edges cannot become enabled at all, and that splitting to respect the guards of these edges turns out to be redundant. This can only partly be avoided, since some splitting will be needed to be able to determine if a region is urgent or not. In principle a little efficiency could be won by delaying splitting on guards to the point at which edges are explored.

In general, the meaning of guaranteed termination is very limited. Systems that have a finite partition may be far too large to verify, while for other systems that do not have a finite partition, verification may turn out to be feasible. Note that if a finite stable partition can be reached by the algorithm, then it will terminate thanks to the progress property for $S$ discussed in the paragraph addressing $S$ and $C$.

Finally, a subset of XTG can be identified for which decidability can be proven for a subset of TCTL. Basically, this would mean that expressiveness is restricted to traditional timed automata. See for example [63] for more information on this topic.

## 6.4 Practical results

The model checking approach described in this thesis has been implemented in two prototype tools called PMC and LPMC [11, 79, 78]. This section briefly discusses these tools and reports on their performance in comparison with other model checking tools.

### 6.4.1 Two model checkers

PMC and LPMC are two similar implementations of the model checking algorithm discussed in this chapter. They only differ in the type of constraints that are allowed to occur in the symbolic state space. PMC is restricted to deal with difference constraints on integer and real variables and constraints on enumerated variables, while LPMC differs from PMC in that it can deal with arbitrary linear constraints on real variables. The distinction between the two versions originates from the implementation of the reduction routine. Reduction is the key aspect of our model checking approach in terms of performance — it is by far the most time-consuming operation. As was shown in section 5.5, a computationally simple procedure exists for reduction on difference constraints, while for linear constraints a more complex procedure is needed. As a consequence, the two versions of the tool are very different in terms of performance.

Obviously, the type of constraints that are allowed in the symbolic state space determines the type of verification problems that can be handled by the model checker. This means that for PMC system specifications are limited to only single XTG specifications, while for LPMC, linear XTG specifications are allowed (see definition 2.18 for the definition of these XTG subsets). The restriction on mixing clocks and discrete real variables comes from the fact that such mixing will result in non-difference constraints in symbolic states. As a result of this restriction, most parametric real-time verification problems will need a model checker that is based on linear constraints, since parameters will often be compared with clocks.

Both tools allow the generation of error or evidence traces, although current versions support this for safety/reachability properties only. The user can specify whether a depth-first or breadth-first strategy should be applied. The latter strategy is particularly useful for generating traces. Also, the tools can be instructed to assume that the system is non-zeno, resulting in more efficient verification for non-fair safety/reachability properties.
For LPMC, a linear reduction module was developed that uses linear programming techniques [15]. The choice was made not to use an existing linear program solver, because these solvers tend to focus on efficiently solving a single large linear programming problem. What is needed is a solver that is able to solve large sets of relatively simple linear programming problems efficiently.

Initially, the implementation of PMC was based on work reported in [11]. Since then, LPMC was introduced and some adaptations were made to the tools to bring them in line with the approach discussed in this thesis. The current versions of the tools mostly originate from around 2000. The tools are implemented in C and primarily targeted at Linux platforms. The two model checkers are compiled from the same source code, using compiler switches to select the appropriate type of constraints that are allowed. This illustrates the ease of changing the tools to allow for other data types. The tools take an ASCII file containing both a system and a property specification. If the system or property contains parameters then the model checkers will — in case of termination — produce a characterization of the set of parameter values for which the property holds. If the verification problem is not parametric, then \( \text{TRUE} \) or \( \text{FALSE} \) is returned, as appropriate. The tool can be instructed to report on decided parameter values on-the-fly, which can be useful in case only for a subset of the parameter values verification turns out to be feasible.

In [17, 36], reports on case studies with PMC and LPMC can be found.

### 6.4.2 Performance comparison

The time and space performance of PMC and LPMC have been compared to performances of some of the currently available model checking tools, namely Uppaal, HyTech and Kronos. Uppaal [74] is currently the most mature model checking tool for verifying real-time systems. Its development started in 1995 and since then many people have worked on it, resulting in a much used and relatively mature verification tool. It is based on the restriction to difference constraints — like PMC. Its input language is an expressive variation of timed automata. It includes an environment for convenient modelling, simulation and verification. HyTech [60] is one of the earliest model checking tools for real-time systems. Unlike Uppaal, it is not a tool that is actively improved. However, HyTech is one of the few tools that is able to deal with linear constraints rather than difference constraints. It is therefore comparable with LPMC. HyTech allows the verification of hybrid automata, an extension of timed automata with continuous variables that have arbitrary progress rates. Kronos [110] was developed around the same time as HyTech, and — like Uppaal and PMC — relies on the restriction to difference constraints. All three tools allow the verification of properties defined in a variation of TCTL. Also, all three tools are based on symbolic backward or forward analysis. Kronos also includes a partition refinement approach. However, verification is not done on-the-fly. First an abstract model is generated, which is subsequently checked using an explicit-state model checker.

For comparison, we took two well-known scalable benchmark problems, namely the Fischer mutual exclusion protocol [1] and the CSMA/CD protocol [110]. The experiments were performed on a 1GHz Pentium III with 512 Mb internal memory on Linux. The following version of the tools were used: Uppaal version 3.4.2 (October 2003); Kronos version 2.5i.2 (September 2002); HyTech version 1.04f (January 2002).

Figure 6.17 compares performances of PMC, Uppaal and Kronos for the two benchmark problems. A dash indicates that verification was infeasible in terms of required resources.
— either the system ran out of memory or verification was terminated after 15 minutes. The fact that Kronos fails already at Fischer-6 and CSMA-6 has a specific reason. Kronos performs the construction of the global control structure as a preprocessing step, unlike PMC and Uppaal. Kronos was not able to construct the global graph given the available space resources. For Uppaal we performed the test for two cases, without and with approximation enabled. The latter is enabled by the -A option. Kronos was also applied in two ways. Without any options, Kronos performs standard symbolic analysis, while the -ta options has the effect that Kronos applies ta-bisimulation-based partition refinement. Note that for the latter case the performance measurements only include the construction of the minimal model, not the subsequent verification which is to be performed by a separate tool. Thus, unlike in PMC, the verification is not done on-the-fly, but only after the minimal model is completely generated.

Figure 6.17 compares LPMC to HyTech using the same two benchmark problems. LPMC and HyTech are different from the tools in the previous comparison in that these allow arbitrary linear constraints in state space. For HyTech, the tests were performed with both forward and backward analysis.

The figures first of all show that the relative performance of tools is strongly related to the type of verification problem. For other verification problems, similar figures were obtained. In general one can say that the performance of PMC and LPMC is roughly comparable to other available tools. However, an exception has to be made for Uppaal when using its approximation option, which seems to lead to a significant better performance. Finally, note that one has to be careful in comparing performances of the tools. In

<table>
<thead>
<tr>
<th>problem</th>
<th>time (PMC)</th>
<th>mem (PMC)</th>
<th>time (Uppaal)</th>
<th>mem (Uppaal)</th>
<th>time (Uppaal -A)</th>
<th>mem (Uppaal -A)</th>
<th>time (Kronos)</th>
<th>mem (Kronos)</th>
<th>time (Kronos -ta)</th>
<th>mem (Kronos -ta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fischer-2</td>
<td>0.04</td>
<td>1.3</td>
<td>0.02</td>
<td>2.4</td>
<td>0.04</td>
<td>2.0</td>
<td>0.00</td>
<td>1.3</td>
<td>0.01</td>
<td>1.3</td>
</tr>
<tr>
<td>fischer-3</td>
<td>0.04</td>
<td>1.3</td>
<td>0.03</td>
<td>3.3</td>
<td>0.04</td>
<td>2.0</td>
<td>0.01</td>
<td>1.3</td>
<td>0.02</td>
<td>1.3</td>
</tr>
<tr>
<td>fischer-4</td>
<td>0.05</td>
<td>1.3</td>
<td>0.05</td>
<td>3.3</td>
<td>0.06</td>
<td>2.0</td>
<td>0.12</td>
<td>2.6</td>
<td>0.18</td>
<td>1.3</td>
</tr>
<tr>
<td>fischer-5</td>
<td>0.17</td>
<td>2.8</td>
<td>0.68</td>
<td>4.7</td>
<td>0.13</td>
<td>3.3</td>
<td>0.90</td>
<td>12</td>
<td>2.02</td>
<td>4.5</td>
</tr>
<tr>
<td>fischer-6</td>
<td>1.4</td>
<td>6.8</td>
<td>143</td>
<td>26</td>
<td>0.41</td>
<td>4.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fischer-7</td>
<td>17</td>
<td>32</td>
<td>-</td>
<td>-</td>
<td>1.7</td>
<td>5.8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fischer-8</td>
<td>430</td>
<td>243</td>
<td>-</td>
<td>-</td>
<td>7.2</td>
<td>16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fischer-9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>30</td>
<td>41</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>fischer-10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>125</td>
<td>145</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSMA-2</td>
<td>0.03</td>
<td>1.3</td>
<td>0.02</td>
<td>1.3</td>
<td>0.02</td>
<td>2.0</td>
<td>0.00</td>
<td>1.3</td>
<td>0.01</td>
<td>1.3</td>
</tr>
<tr>
<td>CSMA-3</td>
<td>0.06</td>
<td>1.3</td>
<td>0.02</td>
<td>1.3</td>
<td>0.02</td>
<td>3.3</td>
<td>0.01</td>
<td>1.3</td>
<td>0.05</td>
<td>1.3</td>
</tr>
<tr>
<td>CSMA-4</td>
<td>0.21</td>
<td>3.2</td>
<td>0.02</td>
<td>3.3</td>
<td>0.03</td>
<td>3.3</td>
<td>0.02</td>
<td>1.3</td>
<td>0.81</td>
<td>1.3</td>
</tr>
<tr>
<td>CSMA-5</td>
<td>1.8</td>
<td>7.1</td>
<td>0.16</td>
<td>3.7</td>
<td>0.04</td>
<td>3.3</td>
<td>0.11</td>
<td>2.0</td>
<td>14.1</td>
<td>1.3</td>
</tr>
<tr>
<td>CSMA-6</td>
<td>19</td>
<td>35</td>
<td>2.3</td>
<td>7.8</td>
<td>0.10</td>
<td>3.4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSMA-7</td>
<td>404</td>
<td>227</td>
<td>239</td>
<td>34</td>
<td>0.30</td>
<td>4.8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSMA-8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2.1</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSMA-9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6.1</td>
<td>21</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSMA-10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>18.5</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSMA-11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>53</td>
<td>141</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 6.18: Comparing PMC, Uppaal and Kronos

Practical results
particular, Uppaal is a relatively complete and mature tool in which many optimization techniques have been implemented, while the other tools — including PMC and LPMC — have a more "academic" and experimental character.

### 6.5 Discussion

This chapter combined the results from the previous three chapters resulting in a model checking algorithm for TCTL. This was done in two steps. First the abstract decision approach defined in chapter 4 was put into a more concrete form of decision rules operating on valuations. These rules were still generic in the sense that the embedding in an actual algorithm can be done in different ways. In particular, the decision rules define an optimal decision approach, while a model checking algorithm may chose to implement a non-optimal interpretation of these rules for reasons of performance. This is also the case for our algorithm, which was the result of the second part of this chapter. Here, the decision approach chosen is non-optimal in the sense that for global (type 4) decisions a simple and computationally efficient approach was taken. However, this may be at the cost of unneeded exploration of parts of a state space. In practice, one has to find a balance between these two aspects of efficiency.

Besides extending the algorithm of chapter 3 towards TCTL verification, it also introduces splitting trees as an underlying state space representation. Rather than abstracting from state space representation, this aspect of model checking is now made explicit. Also a more refined splitting approach was introduced. Rather than choosing a brute-force (implicit) initial partition $\rho_0$ that ensured that all invariants, guards and atomic properties were respected, splitting for these aspects is done only if really needed. Only the invariants are seen as part of the initial partition. Related to this is the improved granularity of splitting induced by the splitting tree approach. Rather than splitting to complete constraints, splitting is done on the level of single constraints, avoiding unneeded splitting.

Furthermore, the algorithm description shown here is less abstract in the sense that it

<table>
<thead>
<tr>
<th>Problem</th>
<th>LPMC</th>
<th>HyTech (forward)</th>
<th>HyTech (backward)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>mem</td>
<td>time</td>
</tr>
<tr>
<td>fischer-2</td>
<td>0.05</td>
<td>8.5</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.3</td>
</tr>
<tr>
<td>fischer-3</td>
<td>0.05</td>
<td>8.5</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.8</td>
</tr>
<tr>
<td>fischer-4</td>
<td>0.11</td>
<td>8.6</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>fischer-5</td>
<td>2.0</td>
<td>9.3</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>fischer-6</td>
<td>41</td>
<td>13</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>fischer-7</td>
<td>728</td>
<td>39</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>CSMA-2</td>
<td>0.13</td>
<td>2.4</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.0</td>
</tr>
<tr>
<td>CSMA-3</td>
<td>0.40</td>
<td>8.6</td>
<td>0.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.2</td>
</tr>
<tr>
<td>CSMA-4</td>
<td>13.2</td>
<td>9.3</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.6</td>
</tr>
<tr>
<td>CSMA-5</td>
<td>200</td>
<td>14</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7.6</td>
</tr>
<tr>
<td>CSMA-6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>CSMA-7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>CSMA-8</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>81</td>
</tr>
</tbody>
</table>

Figure 6.18: Comparing LPMC and HyTech
avoids constructs that cannot directly be translated to a model checking tool implementation. Implicit definitions are replaced by concrete descriptions. Also, the ever-present self-looping edges are not represented in the algorithm, although obviously their presence is taken into account. One aspect that was deliberately not made explicit is the exact form of the set $S$. Here different choices can be made, which could for example result in a depth-first exploration approach, or a mostly breadth-first exploration approach. Experiences show that for some cases, depth-first analysis performs better, while for others, breadth-first analysis is a better solution. For discussions of related algorithms, refer to the discussion sections of chapters 3 – 5. To our knowledge, there are no algorithms that build on a similar combination of partition refinement, splitting trees and TCTL verification.

Initially we based part of our approach on the region product graph approach of Sokolsky [98, 97]. This was described in [11]. Like a region graph, a region product graph is a partition of a more fine-grained, usually infinite, structure, induced by an equivalence relation. This underlying structure, called timed product structure is a product graph of a timed transition system and a graph model of the property of interest, called formula graph. A region product graph is defined as a finite quotient structure of a timed product structure, like the way a region graph is a quotient of a timed transition system. The region product graph approach can be used to lead to the same results as the approach that is chosen in this thesis. In fact, the implementation discussed in chapter 6.4 was initially based on the region product graph approach. One of the advantages of the current approach is that different aspects of a TCTL verification approach are more explicitly addressed using an intermediate step in terms of standard CTL verification. In the region product graph approach, no evident semantics could be given to timed product structures and region product structures. Note that roughly, the role of the formula graph in the region product graph approach is in the current approach played by a property automaton.

There are several possibilities to extend the model checking algorithm presented here with other techniques that are often used in (real-time) model checking. Many of these techniques are orthogonal to our model checking approach — they can be included because they do not directly interfere with our algorithm. An example of this is the clock reduction approach discussed in [45]. Also, several other techniques used in other model checking approaches can be adapted to also fit with our approach, for example partial order reduction techniques [92, 52, 25].

$$c := 2c$$

$$c := 0$$

![Figure 6.19: Example of non-termination](image)

Also, several specific optimizations can be applied to the general algorithm that was described here. For example, for each clock $c$, a standard invariant $c \geq 0$ would be added to avoid that negative clock values are taken into account. This can be realized by a single subtree with such a constraint for each clock and add this tree on top of each tree in the forest, thus avoiding that these standard invariants are repeated in each splitting tree.

Another aspect that is not mentioned here, but which is essential in a model checking tool is the generation of evidence traces. The algorithm can be extended to produce such
traces. The model checker discussed in chapter 6.4 is able to produce traces.

Finally, the issue of termination. It is a fact that for the type of systems the algorithm is focussed at, there are generally no guarantees for feasibility of verification. The reasons for infeasible verification problems can be diverse. Often, verification problems will simply lead to untractable state spaces, but for some cases surprisingly simple causes can be found. Consider for example figure 6.19. One can easily verify that our partition refinement approach will not terminate as a result of endless splitting, while it is clear that the second location can never be reached.

\[
\begin{align*}
c &:= c - 1 \\
c &:= p & [c < 1]
\end{align*}
\]

Figure 6.20: Example of non-termination in parametric analysis

Figure 6.20 shows another case of non-termination, this time due to the presence of parameters. Regions corresponding to the first location will be endlessly split. This is a phenomenon that can easily be avoided by putting a lower bound on the parameter \( p \).

An extreme example is shown in figure 6.21, where \( p \) is again a parameter. Here verification of reachability of the third location would not terminate, because the algorithm is not able to stabilize the regions associated with the first location. Here, more optimal decision making based on the general rules of figure 6.1, that also allows global decision making of non-fully explored regions could lead to termination.

\[
\begin{align*}
x &:= 0 \\
x &:= 0 & y := 0 & [x \geq p]
\end{align*}
\]

Figure 6.21: Another example of non-termination in parametric analysis
Chapter 7

Conclusions

This thesis presented a model checking approach for real-time systems based on partition refinement. It was developed by subsequently considering the three key aspects identified in the introduction: exploration, representation and evaluation of the state space. For each aspect, a solution was developed, with the basic assumption that the exploration aspect is based on partition refinement, using the minimal model generation algorithm as a starting point. Thus, one could say that the aim was to find a partition-refinement based exploration approach with matching solutions for the representation aspect and for the evaluation aspect of symbolic model checking.

The described model checking approach is aimed at parametric verification of real-time temporal logic properties (fair TCTL) on an extension of timed automata (XTG). At an abstract level, our partition refinement approach is based on a mixture of existing approaches. It is made more concrete by taking into account specific solutions for state space representation and evaluation. A key quality of the state space representation solution (splitting trees) is that it is dedicated to state spaces generated by partition refinement techniques. This sets our approach apart from other partition refinement approaches. The evaluation aspect was approached from a quite general perspective. A decision-making approach was described for arbitrary partial symbolic state spaces and fair TCTL properties. These decision rules are optimal in the sense that decisions are made as soon as enough information is available. Given these rules, a specific choice has to be made concerning the application of these rules, in which efficiency considerations have to be taken into account. The work on the three aspects of symbolic model checking cumulated in a relatively concrete model checking algorithm. This algorithm is to some extent specific for the splitting tree representation and makes specific choices in terms of the generic decision rules.

Model checking is to some extent a heuristic technique — worst-case performance is usually quite bad, but for concrete applications often good results can be achieved. The actual implementation of a proposed technique is therefore of great importance to be able to judge the potential. Therefore, the algorithm has been implemented in two prototype model checking tools, each having a different balance between expressiveness and performance. Results of experiments with these tools indicate that their performance is roughly comparable with other tools. From the performance measurements, it seems that partition refinement is a valid approach to symbolic model checking. More work would be needed to be able to more precisely relate partition-refinement approaches to symbolic forward and backward analysis approaches. It could very well be that the optimal choice
of algorithm depends on the type of verification problem at hand.

Looking back at the objectives stated in the introduction, we have first of all shown that partition refinement is a suitable approach for model checking real-time systems. It was shown that it can be applied in TCTL verification of an expressive timed automaton variant. Secondly, we have also shown that a partition refinement approach is valid in terms of performance — the efficiency of solving verification problems seems to be at least comparable to that of other tools.

Our second objective was met by choosing a rich verification framework — based on XTG and TCTL — of which it is worthwhile to mention that (for LPMC) it allows arbitrary linear constraints. Also, the approach can be readily extended to deal with more complex verification problems. Extension with new data types is realized by creating appropriate reduction and propagation procedures, while extension towards more complex temporal logic properties, would require the introduction of additional decision rules. An additional advantage of having such a convenient separation of data-type specific aspects is that one can conveniently produce several model checker instances for different categories of verification problems.

Concerning the third objective, we succeeded in finding a state space representation that is fit for use in a partition refinement approach. Although improvements may be possible, it seems that splitting trees form a state space representation approach that fits naturally with partition refinement. From the work presented, it seems likely that for partition refinement algorithms, a splitting tree representation leads to a better performance than standard canonical approaches. However, for definitive answers one would have to build another version of our tools using for example DBM’s.

Concerning the objective for fair TCTL verification, we investigated the problem of deciding TCTL properties on partial systems, separated from a specific exploration approach. For many model checking approaches, the evaluation solution is closely related to the chosen exploration approach — usually symbolic forward or backward analysis. For partition refinement, we took a more general perspective. A two-step approach was adopted in which first general and optimal decision rules are described, which are subsequently used to define a specific evaluation procedure.

Our final objective was met by realizing a concrete model checking approach and the corresponding implementations of two model checkers. Thus, our model checking approach was pursued to a concrete level. The implementation of concrete algorithms allowed us to show the validity of the approach in terms of performance.

Several possibilities for further work can be identified. Chapter 5 mentioned several suggestions for directions towards improving the efficiency of splitting trees. It seems likely that more elaborate splitting-tree approaches can be developed that are more efficient, both in terms of time and data. This could involve smarter reduction techniques as well as smarter variations of the splitting trees themselves.

Furthermore, the work in this thesis focused on symbolic representation of the data aspect of state spaces. The control aspect is still explicitly represented. In particular, a symbolic structure is associated with each global location. However, experience shows that there is a lot of similarity between the symbolic states associated with the different global locations. Therefore, it seems that there is a potential for a truly symbolic approach that also involves the control component, for example in the spirit of [46]. More concretely, an interesting experiment would be to directly compare the splitting tree representation with the widely used canonical representations, by building two versions of the same tool, but with different underlying state space representation.
Concerning the tool implementations, a re-implementation focusing on efficiency would certainly result in a considerably more efficient model checking tool. Also, note that for the current implementations there is no further formalization besides the high-level descriptions as presented in this thesis. In principle, the design and implementation of any model checker should be completely formalized. Although this may be infeasible given the current state of the art, for a mature model checking tool one would expect maximal usage of formal methods to ensure that the tool produces highly reliable results. Besides improvements on the implementation level, also many known techniques like for example clock reduction [45] and symmetry reduction [65, 25] could be included. Furthermore, the approach can relatively easily be extended towards more complex data types. The introduction of a new data type (for example, sets), would mean that for that data type, propagation and reduction operations are defined.

At a more fundamental level, it would be interesting to investigate the possibility of applying approximate techniques like [66] in the context of partition refinement. Approximate verification techniques have proven to be successful in symbolic model checking approaches — as became clear from the performance figures of section 6.4.2.

Finally, for model checking in general there is still much work to do apart from improving the capabilities of model checkers. In particular, an important line of further work would be the embedding of model checking techniques in software engineering practices. This would require building connections to current mainstream engineering methods and tools, for example in the context of the Unified Modelling Language. Another issue in this context is that due to the limited scalability of model checking techniques, model checkers are typically applied to models that abstract away from every detail that is irrelevant for proving the property. This means that when trying to integrate model checking approaches in engineering processes, support is needed for generating appropriate models from the models that are constructed in the engineering process. In fact, what is needed here are solutions that facilitate the constructions of correct abstractions aimed at specific verification problems. One could apply theorem provers for this purpose, but this would typically destroy the automatic aspect of model checking approaches. Ideally, one would like to have dedicated tools that given for example a UML (state-transition) model, construct more abstract models that can be used as input for verification, perhaps with limited guidance from humans.
Bibliography


Bibliography


Bibliography
Summary

Model checking Real-Time Systems based on Partition Refinement

The problems involved in systematic production of complex software are numerous and hard to tackle. Many improvements have been proposed over the years, some of which have proved to be very successful while many others have been silently discarded. One approach that has been around for a long time is formal methods. Formal methods potentially enable the development of complex systems at a higher level of correctness than can be reached with conventional, informal methods. Despite much research, formal methods have not yet been adopted by the software industry. However, the steady progress in formal methods research, as well as the growing emphasis in industry on modelling and verification seems to lead to increasing importance of mathematically-based development approaches. A particularly useful view concerning practical application of formal methods is that formal methods do not have to represent a solution for the complete development of systems. Rather, formal methods can be seen as a tool that provides support for a specific part of the development process, applied to — for some reasons critical — selected parts of a system.

In recent years, model checking has emerged as an approach in formal methods that seems to have the potential to play a significant role in industrial software and hardware development practice. It is a verification approach that fits quite well into the idea of selective application of formal methods. Model checking is an algorithmic technique for verifying formal specifications of hardware or software systems against formal specifications of properties. It is based on exhaustive state space exploration. By examining all of a system’s behaviour, one can decide whether or not it behaves conform specified properties. Its strength comes from the fact this can be done automatically — without any complex interference of humans. The major limitation is in its scalability — state spaces tend to explode as systems become more complex. Therefore, the major challenge in model checking is the search for techniques that allow efficient exploration of large state spaces.

This thesis develops a model checking approach for real-time concurrent systems. For this category of systems, formal verification is particularly relevant. Even very simple protocols or algorithms can already lead to intricate behaviour, making it hard to certify correctness.

Our modelling framework is that of timed automata and real-time temporal logic. Focus is on model checking approaches for “complex” real-time model checking problems — more complex than reachability analysis on standard timed automata. In particular, this concerns parametric systems that allow more complex manipulations on clocks than resets, other data types besides clocks, arbitrary linear constraints, urgency, and real-time temporal logic properties with fairness constraints. For modelling, a rich extension
of timed automata, called XTG, is chosen, along with the often-used real-time temporal logic called TCTL.

In the area of real-time systems, so-called symbolic model checking techniques are typically applied. Symbolic model checking techniques operate on implicitly represented sets of states, rather than explicit states. In this thesis, three key aspects of such a symbolic model checking approach are distinguished, namely the state space exploration algorithm, the symbolic state space representation and the approach to evaluate properties on the state space. In this thesis, the development of a symbolic model checking approach is defined in terms of these three aspects.

- A basic starting point is the focus on a particular state space exploration approach called partition refinement. Partition refinement is based on iterative refinement of a partition of the state space, eventually resulting in a partition that can be used as an abstract model of the complete state space. This abstract model is then used to verify the original property of interest. Currently, successful techniques are all based on variations of symbolic forward or backward analysis. In this thesis it is shown that partition refinement is a valid alternative to these techniques.

- In this thesis, a novel approach to symbolic state space representation, called splitting trees, is described. It is a relatively simple approach to representing symbolic states identified by constraints on clocks and variables. It fits naturally with partition refinement based model checking approaches. Usually, canonical states space representations are used in symbolic model checking. However, for partition refinement approaches, such canonical representations seem to be relatively inefficient because for partition refinement only a very limited set of operation on the symbolic state space is needed.

- We apply partition refinement to TCTL verification. A systematic approach to evaluating such properties is defined. An important quality of this approach is that it is optimal, meaning that it decides satisfaction or dissatisfaction of a property as soon as this is possible. Evaluation of TCTL properties is investigated from a generic perspective, which can be applied to any partial state space, regardless of how it was derived. This allows one to reason about the decision approach that is to be actually implemented in a model checking technique. This is important because early decision making can be used to avoid unnecessary construction of parts of the symbolic state space.

Our model checking approach was pursued to a concrete level. A model checking algorithm is described at a relatively detailed level. Furthermore, this thesis reports on the implementation of this algorithm in the form of two prototype model checking tools (PMC and LPMC). This allowed us to show the validity of the approach in terms of feasibility and performance.

Summarizing, a systematic approach is taken to develop a partition refinement model checking approach for TCTL and XTG. For each of the three above identified key problems of symbolic model checking, solutions are developed. Subsequently, these three aspects are joined to produce a concrete model checking approach. Also, it is shown that the resulting model checking approach is roughly comparable with other approaches in terms of data and time performance. This means that partition refinement approaches can be regarded as a valid model checking approach.
Samenvatting

Verificatie van tijdsgebonden systemen met behulp van model checking technieken gebaseerd op partitie-verfijning

Systematische software ontwikkeling brengt een groot aantal moeilijk op te lossen problemen met zich mee. Vele verbeteringen zijn voorgesteld, waarvan sommige succesvol bleken, maar waarvan ook vele in stilte verdwenen zijn. Een al lang bestaande aanpak gericht op de ontwikkeling van betere software, is formele methoden. Formele methoden hebben de potentie om een hoger niveau van correctheid te bewerkstelligen dan traditionele, informele methoden. Ondanks vele jaren van onderzoek, zijn formele methoden nog geen onderdeel van industriële software ontwikkel processen. Echter, de gestage vooruitgang in onderzoek naar formele methoden en de groeiende nadruk in de industrie op modellering en verificatie, lijkt te leiden tot toenemende mogelijkheden van meer formele aanpakken. Een bruikbare kijk op de toepassing van formele methoden is dat deze niet een oplossing bieden voor een compleet ontwikkelprocess, maar dat formele methoden selectieve ondersteuning kunnen bieden in de ontwikkeling van specifieke, om verschillende redenen kritische, aspecten van een systeem.

In het afgelopen decennium heeft model checking zich gemanifesteerd als een aanpak in formele methoden die de potentie heeft een belangrijke rol te kunnen spelen in industriële hardware en software ontwikkeling. Model checking is een aanpak die zich goed leent voor selectieve toepassing in een ontwikkelproces. Het is een algoritmische techniek voor formele verificatie van formeel beschreven modellen en eigenschappen. Het basis idee achter model checking is het uitputtend doorzoeken van de toestandsruimte van een systeem. Door alle gedragingen van een systeem in beschouwing te nemen kan een uitspraak gedaan worden omtrent de correctheid van het systeem. De kracht van model checking zit in het automatische karakter van de resulterende verificatie gereedschappen. De belangrijkste beperking van model checking daarentegen, zit in de schaalbaarheid — toestandsruimtes van complexe systemen zijn doorgaans erg groot. De belangrijkste uitdaging op het gebied van model checking zit dan ook in het vinden van efficiënte aanpakken voor het evalueren van grote toestandsruimtes.

In dit proefschrift wordt een model checking aanpak ontwikkeld voor real-time, gedistribueerde systemen. Voor dit type systemen is formele verificatie zeer relevant. Zelfs simpele protocollen en algoritmen kunnen al leiden tot complex en onnauwkeurig gedrag, waardoor het vaststellen van correctheid vaak zeer moeilijk is.

Voor de modellering van systemen en eigenschappen is gekozen voor het veelgebruikte raamwerk timed automata en real-time temporele logica. De nadruk ligt daarbij op het komen tot oplossingen voor relatief "complexe" verificatie problemen — meer complex dan reachability analysis op standaard timed automata. Specifiek betreft dit meer com-
plexe manipulaties op klokken, andere data typen dan klokken, het toestaan van lineaire
caractericte, het modelleren van urgency en het gebruiken van een real-time temporele
logica op basis van fairness. Voor het modelleren van systemen wordt XTG gebruikt, een
uitbreiding van timed automata, terwijl voor het modelleren van eigenschappen de logica
TCTL gebruikt wordt.

Voor de verificatie van real-time systemen worden doorgaans zogenaamde symbolic
model checking technieken toegepast. Deze hebben als belangrijkste eigenschap dat ze op
impliciet gepresenteerde toestandsverzamelingen werken, in plaats van op expliciete toes-
tanden. In dit proefschrift wordt het onderscheid gemaakt tussen drie belangrijke aspecten
van een symbolic model checking techniek. Dit betreft het doorzoeken van de toestands-
ruimte, het representeren van de toestandsruimte, en het evalueren van eigenschappen op
basis van een toestandsruimte.

• Een belangrijk uitgangspunt is de keuze voor zogenaamde partie-verfijnings (partition
refinement) technieken voor het doorzoeken van toestandsruimten. Deze zijn gebaseerd
op iteratieve verfijning van een partitie van de toestandsruimte. Het doel daarbij is om
uit te komen op een abstract model dat gebruikt kan worden om de te onderzoeken
eigenschap van het systeem te evalueren. De meeste nu gebruikte technieken zijn
gebaseerd op symbolische reachability analysis. In dit proefschrift wordt aannemelijk
gemaakt dat partition refinement een valide alternatief is voor de huidige aanpakken.

• Een nieuwe aanpak voor de representatie van symbolische toestandsruimten wordt
beschreven. Deze representatie, genaamd splitting trees, vormt een relatief eenvoudige
aanpak voor het representeren van abstracte toestanden die gekenmerkt zijn door con-
straints op klokken en data. Deze aanpak sluit op natuurlijke wijze aan op exploratie-
aanpakken gebaseerd op partition refinement. Vaak worden zogenaamde kanonieke
aanpakken gebruikt in symbolic model checking. Echter voor partition refinement
zijn zulke aanpakken relatief inefficient omdat voor partition refinement een zeer
beperkte verzameling van operaties op toestandsruimtes nodig is.

• Een systematisch aanpak voor de verificatie van TCTL specificaties is gedefinieerd.
Een belangrijke eigenschap van deze aanpak is optimaliteit, wat betekend dat beslissin-
gen omtrent het wel of niet voldoen aan de gespecificeerde eigenschap zo snel mogelijk
genomen worden. Evaluatie van TCTL specificaties wordt op een generieke manier be-
naderd, zodat de resulterende evaluatie aanpak toegepast kan worden op elke partiele
gegenereerde toestandsruimte, onafhankelijk van het algoritme dat gebruikt is om de
toestandsruimte te genereren. Dit maakt het mogelijk om keuzes omtrent de evaluatie
aanpak expliciet te maken. Dit is belangrijk omdat het zo snel mogelijk nemen van
beslissingen onnodige exploratie van de toestandsruimte kan voorkomen.

De beschreven model checking aanpak is concreet gemaakt in een relatief gedetailleerd
beschreven algoritme. Daarnaast is dit algoritme geïmplementeerd in een tweetal model
checking tools. Dit maakt het mogelijk om de haalbaarheid en efficiëntie van het algoritme
to evalueren.

Samenvattend is een systematische aanpak gekozen om te komen tot een op partition-
refinement gebaseerde model checking aanpak op basis van XTG en TCTL. Voor elk van de
hierboven geïdentificeerde aspecten zijn oplossingen aangedragen, welke zijn samengevoegd
teneinde te komen tot een concreet model checking algoritme. Daarnaast laten we zien dat
de resulterende aanpak in termen van performance grofweg vergelijkbaar is met andere
aanpakken. Daarmee lijkt partition refinement een valide aanpak voor symbolic model
checking te zijn.
Curriculum Vitae

9 juli 1969 geboren te Enschede

1981–1987 Atheneum B
Jacobus College, Enschede

1987–1992 Studie Informatica, Universiteit Twente
Vakgroup Tele-Informatica en Open Systemen

1993 Militaire dienst

1994–1999 Assistent in Opleiding
Technische Universiteit Delft
Vakgroup Technische Informatica

Sinds 1998 System designer
Thales Naval Nederland