Computation of winning strategies for μ-calculus by fixpoint iteration

Christian Neukirchen

München 2014

Computation of winning strategies for μ-calculus by fixpoint iteration

Christian Neukirchen

Masterarbeit
am Institut für Informatik
der Ludwig-Maximilians-Universität München

vorgelegt von
Christian Neukirchen
geboren in Biberach am 16.08.1987

angemeldet am 20. Mai 2014
eingereicht am 11. November 2014

Betreuer: Prof. Martin Hofmann, PhD
Lehr- und Forschungseinheit für Theoretische Informatik

Contents

Abstract

We present an implementation of a model checker for μ-calculus that certifies its result as a winning strategy for a corresponding parity game. The winning strategy is computed by means of a fixpoint iteration, similar to the well-known set semantics of μ-calculus, and can be regarded as a instrumentation thereof. The computed certificates are compact and can be checked efficiently in low polynomial time by a separate routine.

Kurzzusammenfassung

Wir stellen eine Implementation eines Modellprüfers für den μ-Kalkül vor, die mithilfe von Gewinnstrategien für ein korrespondierendes Paritätsspiel ihr Rechenergebnis zertifiziert. Die Gewinnstrategie wird mittels Fixpunktiteration berechnet, in einem Prozess ähnlich der bekannten Mengensemantik des μ-Kalküls, und kann als Instrumentierung dieser aufgefasst werden. Die berechneten Zertifikate sind kompakt und können effizient in niedriger Polynomialzeit durch ein eigenständiges Programm überprüft werden.

Preface

Writing correct software is hard, and thus over the last decades, many techniques have been developed to assist programmers produce better software. Especially applications such as embedded systems, device control and concurrent systems have high demand for safety and correctness, because large sums or even human life are at hazard in case of a malfunction. For these applications, mere testing does not cut it, because tests cannot—in all but the simplest cases—be exhaustive, but we want the guarantee that things *cannot go wrong*.

Formal verification methods such as *deductive verification* allow us to *prove* generic propositions about our programs. However, these proofs generally require large effort and significant human assistance. Often, using these methods cannot be justified because it would take too long or is very expensive.

Model checking provides a middle ground: properties can be verified automatically, and the restrictions implied (such as finite models or less expressive calculi) are gratefully accepted as a trade-off. Thus, model checking became the probably most successful verification technique that is used in real-life industrial applications, spanning the wide range from network protocols and chip design to aircraft collision avoidance systems or self-destruction of sounding rockets.

However, one problem remains: akin to Juvenal's question *"Quis custodiet ipsos custodes?"*, we can ask "Who proves the model checkers correct?" Industrial grade model checkers are large and complex applications, and the more sophisticated and optimized their algorithms become, the more likely there is a bug somewhere which could yield a wrong result—with possibly fatal consequences.

But there is a resort: we can implement a model checker which *certifies its result*, that is, a second program—the *certificate checker*—can independently (and hopefully more efficiently than the model checker itself!) verify that the result of the model checker is correct. In case of a disproof, this is easy: you can just output a single counterexample that refutes the proposition. How do you certify that the formula holds, though?

Hofmann and Rueß [19] propose a solution: we can certify propositions in μ-calculus, a very general model checking calculus, using *winning strategies* for parity games corresponding to the μ-calculus formula. We have implemented a simple model checker (named *Micromu*) based on μ-calculus which computes these winning strategies while performing the model check. Verifying these winning strategies is comparatively easy and can be done completely independent. This allows us to satisfy the so-called *de Bruijn criterion* of "satisfying the possibility of independent checking by a small program" [1]. Now, we only need to trust this small program, the certificate checker, as it will notice when the (potentially large and complex) model checker made a mistake.

Thinking one step ahead, we can also *formally verify* the certificate checker (hopefully using a proof assistant that fulfills the *de Brujin criterion* itself), and it's almost as good as having the full model checker proved formally. You could even say it's better, because the proof that the certificate checker works does not tangent the model checker at all! You can freely improve the model checker without danger of undetected false results.

Overview. We progress in the following order: Chapter 1 addresses model checking and defines the models and formulae we use throughout, as well as their fundamental semantics. We strive through different temporal logics and approaches for tackling common problems in model checking. We also review existing tools.

In Chapter 2, we define parity games and associated objects, such as game strategies and explain how strategies can be used as certificates.

Next, Chapter 3 is the heart of this work: we show how to interpret model checking problems as parity games, and how to construct winning strategies for these games by means of fixpoint iteration. We also show how these winning strategies can be used as certificates, and how to check them. The chapter concludes with a complexity analysis of the algorithms presented.

Chapter 4 illustrates how the ideas of Chapter 3 have been implemented in our prototype implementation Micromu. We underline the feasibility of this implementation with a some benchmarks.

Subsequently, Chapter 5 summarizes the work and our contributions. We point out areas that deserve further attention and ideas for further research.

Appendix A is a short *user's manual* for people that want to work with Micromu.

Finally, Appendix B contains most of the code that is part of the implementation of Micromu.

Notation. The set of natural numbers $\mathbb{N}$ always includes zero. We use $\subset$ for proper subsets only and $\subseteq$ else. We use $\sqcup$ for disjoint unions. In μ-calculus notation, quantifiers bind as far as possible, that is, $\mu X.a \vee b$ is to be read as $\mu X.(a \vee b)$. Partial functions from a set S into a set T are denoted as $f : S \rightharpoonup T$.

Other notations are introduced when needed.

Acknowledgments. Thanks go to Martin Hofmann for the continuous supervision and very helpful discussions, and to Harald Rueß for pointing out some intricacies of μ-calculus and helping me to understand the big picture.

— Christian Neukirchen
Munich, November 2014

CHAPTER 1

Model checking and the modal μ-calculus

We understand *model checking* as the discipline of exhaustive proofs on finite structures. In particular, we want to verify whether formulae of temporal logic hold on infinite paths of given labelled transition systems.

1.1. Goals of model checking

Exhaustive proofs are both a blessing and a burden of this formal method: Obviously, exhaustive checks are easy to mechanize, work completely automatized and have no problems with soundness. Yet, the combinatorial explosion of states is easy to underestimate and requires a high degree of abstraction to make viable models.

Model checking is also popular for concurrent systems, where its exhaustiveness guarantees that no possible interleaving of programs can reach a failure state. However, also in this case the number of states and transitions grows exponentially in the amount of concurrent threads that are modelled.

Classically, model checking uses logical systems of limited expressiveness such as Linear Time Logic (LTL) or Computation Tree Logic (CTL). The logic CTL*, which unites LTL and CTL, is often the most rich logic investigated.

In contrast, μ-calculus subsumes all these. LTL and CTL formulae can be translated into μ-calculus in a relatively straight-forward way, and it can be shown that μ-calculus, limited to this subset, is (algorithmically) not less efficient than specialized model checking supporting LTL or CTL only. Obviously, more complex (and thus more expressive) formulae of μ-calculus will make the problems harder, but this price is only paid when necessary. Thus, studying μ-calculus is a viable approach for solving general model checking problems, even if the full power is not required for actual applications [14].

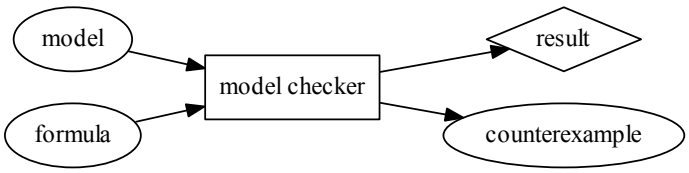


FIGURE 1.1. Standard model checking workflow.

We will try to explain the differences between different methods of model checking using workflow diagrams such as fig. 1.1: For classic model checking, we need to specify the model (more on how this looks like in the next section) and a formula stating the property we want to check. Both inputs are handed to the model checker (programs/algorithms are displayed as rectangles), which then computes a Boolean result, whether the formula is satisfied by the model or not (Boolean results are drawn as diamonds). Often, model checkers can compute counterexamples in case the formula does not hold, but when it does we only get a "that's right".

1.2. Labelled transition systems

Our models are *labelled transition systems* (short LTS, also known in the literature as *Kripke structures*).

We use a standard definition of labelled transition systems:

DEFINITION 1.1. *A* labelled transition system *is a quadruple* $\mathcal{L} = (S, \xrightarrow{a}, \mathcal{A}, T)$*, where* S *is a non-empty set of* states*,* $\xrightarrow{a} \subseteq S \times S$*, a family of left-total* transition *relations on* S*, indexed over* $\mathcal{A}$*, the set of* actions*. The function* $T : S \to 2^{\mathcal{P}}$ *assigns which* propositions $p \in \mathcal{P}$ *are considered to be true at each state* $s \in S$*.*

Note that the relation $\xrightarrow{a}$ is required to be left-total, i.e. there needs to be a possible further move from each state. Thus, all paths on the transition system are infinite paths.

We make no further assumptions about the actions, and will generally consider a generic action a only.

We mostly consider *finite* LTS later, i.e. all sets of the quadruple are finite.

1.3. μ-calculus syntax

DEFINITION 1.2. *A* μ-calculus formula $\phi \in \Phi$ *has the following syntax:*

$$
\begin{array}{lr}
\phi ::= X & \textit{(variables)} \\
\quad \mid p \mid \neg p & \textit{(atomic propositions)} \\
\quad \mid [a]\phi & \textit{(for all } a\textit{-transitions)} \\
\quad \mid \langle a \rangle \phi & \textit{(}a\textit{-transition exists)} \\
\quad \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 & \\
\quad \mid \mu X.\phi & \textit{(least fixpoint)} \\
\quad \mid \nu X.\phi & \textit{(greatest fixpoint)}
\end{array}
$$

Compared to other expositions such as, e.g. Emerson [11], we do not define $\langle a \rangle$, $\vee$ and $\mu X.\phi$ as syntactic sugar together with general negation: allowing only atomic propositions to be negated, we can ensure *monotonicity* of all syntactically valid formulae. Else, would need to ensure every variable only appears with the same *polarity*: either never negated or always negated. (Consider the formula $\neg X \vee \langle a \rangle X$ on a LTS $0 \rightleftarrows 1$; there is no fixed point.)

1.4. Set semantics, fixpoint iteration, Knaster-Tarski

Given a μ-calculus term, what does it actually mean? We define the following *set semantics* for μ-calculus as a function $sem : \Phi \times (\mathcal{X} \to 2^S) \to 2^S$, which given a formula ϕ and an environment $\eta : \mathcal{X} \to 2^S$, returns the set of states of the LTS where the formula is satisfied.

The set semantics of a μ-calculus formula is given recursively by:

$$\begin{aligned}
sem(X, \eta) &= \eta(X) \\
sem(p, \eta) &= T(p) \\
sem(\neg p, \eta) &= S \setminus T(p) \\
sem(\phi_1 \wedge \phi_2, \eta) &= sem(\phi_1, \eta) \cap sem(\phi_2, \eta) \\
sem(\phi_1 \vee \phi_2, \eta) &= sem(\phi_1, \eta) \cup sem(\phi_2, \eta) \\
sem([a]\phi, \eta) &= \widetilde{pre}(\xrightarrow{a})(sem(\phi, \eta)) \\
sem(\langle a \rangle \phi, \eta) &= pre(\xrightarrow{a})(sem(\phi, \eta)) \\
sem(\mu X.\phi, \eta) &= iter_X(\phi, \eta, \varnothing) \\
sem(\nu X.\phi, \eta) &= iter_X(\phi, \eta, S)
\end{aligned}$$

The first five cases are clear: a variable X holds exactly at the states it is set to, and where p and $\neg p$ are true is determined by T. In order for a conjunction $\phi_1 \wedge \phi_2$ to be true, both operands need to be true, we thus intersect the results. For the disjunction $\phi_1 \vee \phi_2$, it is enough for one side to be true, thus we take the union of the resulting sets.

Next, *pre* is the *preimage* and $\widetilde{pre}$ the *weakest precondition*, defined as follows:

$$\begin{aligned}
s \in \widetilde{pre}(\xrightarrow{a})(U) &\Leftrightarrow \forall t \in S.\, s \xrightarrow{a} t \implies t \in U \\
s \in pre(\xrightarrow{a})(U) &\Leftrightarrow \exists t \in S.\, s \xrightarrow{a} t \wedge t \in U
\end{aligned}$$

A state s is included in the weakest precondition of U, exactly when all possible moves $s \xrightarrow{a} t$ lead into U. A state s is part of the preimage of U, if there is at least one possible move $s \xrightarrow{a} t$ that leads into U. We therefore get the asked-for semantics of $[a]\phi$ meaning "take any a-transition, then fulfill ϕ", respectively $\langle a \rangle \phi$ meaning "there is an a-transition, take it, then fulfill ϕ."

$\mu X.\phi$ and $\nu X.\phi$ are defined by *fixpoint iteration*. This means we apply a rule of computation until the result stays the same. Here, we stop when further application of *sem* does not change the result:

$$\begin{aligned}
iter_X(\phi, \eta, U) = \text{let } U' &:= sem(\phi, \eta[X := U]) \text{ in} \\
&\text{if } U = U' \text{ then } U \text{ else } iter_X(\phi, \eta, U')
\end{aligned}$$

In order to see why this definition is well-defined, we need the following definitions:

Definition 1.3. *A function f from sets to sets is* monotone*, if*

$$x \subseteq y \implies f(x) \subseteq f(y).$$

THEOREM 1.1 (Knaster-Tarski). *If f is a monotone function from sets to sets, then*

$$\mathit{lfp}(f) = \bigcap_{f(P)\subseteq P} P \qquad\qquad \mathit{gfp}(f) = \bigcup_{P\subseteq f(P)} P$$

exist, such that

$$f(\mathit{lfp}(f)) = f \qquad\qquad f(\mathit{gfp}(f)) = g.$$

We say lfp is the least fixed point*, while gfp is the* greatest fixed point*.*

PROOF. A proof for arbitrary partial orders on sets can be found in [26, Theorem 10.29]. For a general, constructive proof on complete lattices, see Cousot and Cousot [6]. □

THEOREM 1.2 (Finite Knaster-Tarski). *Let S be a finite set of size n and $f : S \to S$ a monotone function, then*

$$\mathit{lfp}(S) = f^n(\varnothing) \qquad\qquad \mathit{gfp}(S) = f^n(S)$$

where f^n is the n-times iterated application of f.

Therefore, the fixpoints can be computed in finitely many steps.

PROOF. Since there are only n elements in S, consider the chain $\varnothing \subseteq f(\varnothing) \subseteq f(f(\varnothing)) \subseteq \cdots \subseteq f^{n-1}(\varnothing)$. If one $\subseteq$ actually is $=$, we have found the fixpoint already. It is the least fixpoint by construction. Else, there are $n-1$ different elements in $f^{n-1}(\varnothing)$, thus $f^n(\varnothing) = f(f^n(\varnothing))$ since the set cannot grow further. Analogously, we prove the statement for *gfp*. □

1.5. μ-calculus systems of equations

There is an alternate representation of μ-calculus formulae as in definition 1.2 which uses a *system of equations*. A formula ϕ gets converted to an ordered list of formulae $(X = \phi_X)_X$, with one formula ϕ_X for each variable X occurring in ϕ. For simplicity, we assume variable names don't clash, in doubt α-renaming needs to be performed first.

Every quantifier introduces a new equation: $\mu X.\cdots X\cdots$ turns into $X \stackrel{\mu}{=} \cdots X \cdots$, likewise for $\nu X.\phi$. The formulae are ordered such that subformulae come after the formula containing them, for example:

$$\nu Z.(\neg t_1 \vee (\mu X.c_1 \vee [a]Z)) \wedge [a]Z$$

turns into

$$Z \stackrel{\nu}{=} (\neg t_1 \vee X) \wedge [a]Z$$
$$X \stackrel{\mu}{=} c_1 \vee [a]Z$$

The order of equations is relevant if we want to restore the original formula. If we had started with

$$X \stackrel{\mu}{=} c_1 \vee [a]Z$$
$$Z \stackrel{\nu}{=} (\neg t_1 \vee X) \wedge [a]Z$$

we would retrieve

$$\mu X.c_1 \vee [a](\nu Z.(\neg t_1 \vee X) \wedge [a]Z).$$

In case the formula does not start with a qualifier, a fresh variable needs to be introduced and bound to the formula in first place.

This representation is advantageous for our implementation as it allows cheap substitution of variables for concrete sets of states. To look up a X, we can first check the environment whether $\eta(X)$ is defined, and fall back to looking up X in the system of equations. We therefore don't need to rewrite μ-terms later, which would be very expensive when they nest deeply.

1.6. Expressivity of model checking logics

μ-calculus can be seen as a generalization of other temporal logics used in model checking, such as LTL, CTL or CTL*. CTL* can express everything that can be specified in both LTL and CTL, but μ-calculus is strictly more powerful than CTL* [36].

In fact, every CTL formula can be embedded in μ-calculus with a quantifier alternation depth of 1 and every LTL or CTL* formula with an alternation depth of 2 [14]. Since μ-calculus becomes strictly more expressive with more alternations [20], it is therefore more expressive than CTL*.

1.7. A few examples

To the reader familiar with Computation Tree Logic, it is instructive to see how we can rewrite CTL formulae in μ-calculus [11]:

$$\begin{aligned}
\mathbf{EF}\phi &\equiv \mu Z.\phi \vee \langle a\rangle Z \\
\mathbf{AG}\phi &\equiv \nu Z.\phi \wedge [a]Z \\
\mathbf{AF}\phi &\equiv \mu Z.\phi \vee [a]Z \\
\mathbf{EG}\phi &\equiv \nu Z.\phi \wedge \langle a\rangle Z \\
\mathbf{A}(\phi\,\mathbf{U}\,\psi) &\equiv \mu Z.\psi \vee (\phi \wedge [a]Z) \\
\mathbf{E}(\phi\,\mathbf{U}\,\psi) &\equiv \mu Z.\psi \vee (\phi \wedge \langle a\rangle Z)
\end{aligned}$$

For example, "q holds everywhere along any path" can be written in CTL as "$\mathbf{AG}q$", and would translate into μ-calculus as

$$\nu Z.q \wedge [a]Z$$

that is, "the greatest fixpoint of Z, where q holds and any possible step leads into Z again". Thus, this formula holds exactly at the states where q is true and stays true for every possible path of transitions.

For a more complex example, we'll look at

$$\nu X.\mu Y.(q \wedge \langle a\rangle X) \vee \langle a\rangle Y$$

This means: "the greatest fixpoint of X, then the least fixpoint of Y, such that q holds and we get with a single step into X again, or else we get with a single step back into Y". To fulfill this, we need satisfy the condition leading into X, that is, visit

q infinitely often (but not necessarily in each step). Thus, this is a *fairness condition* for q: we never reach a state where we can not visit q ever again. Conditions like these are popular properties to verify using model checking, for example to avoid livelocks or starvation.

As an explicit argument to show that μ-calculus is more expressive than CTL*, we look at the formula

$$\nu X.p \wedge \langle a \rangle \langle a \rangle X$$

which means "p is true at every second step", a concept which cannot be expressed in CTL* [36].

We can see that raw μ-calculus can be quite difficult to write properties in. It is however practical to use μ-calculus as a target language for more "user friendly" model checking calculi (It is not for nothing that some call μ-calculus "the assembler of model checking").

1.8. Certified model checking

The computation of counterexamples is a standard feature for most model checking systems. Certifying model checkers have been proposed by Namjoshi [25]. For CTL, *witnesses* can be used to certify valid propositions [29].

These witnesses often are rather concrete: for example $\mathbf{EF}\phi$ ("there exists a path such that ϕ holds finally") could be witnessed by the *prefix of the run* such that ϕ holds at the last position.

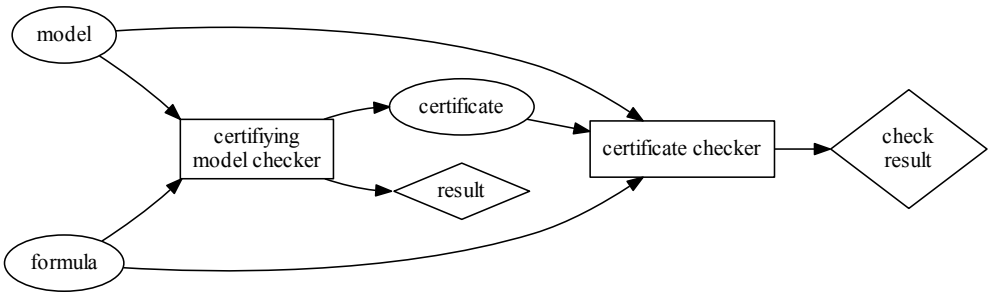


FIGURE 1.2. Certified model checking workflow.

Schematically (see fig. 1.2) certified model checking works as follows: again, we have a model and a formula we want to verify. However, instead of outputting the Boolean result only, the certifying model checker also outputs a certificate of the result. This certificate then can be passed to a separate *certificate checker*, together with the model and the formula. The certificate checker verifies the result of the model checker, and outputs whether the check has passed or not. When the certificate checker says the certificate is valid, the result of the model checker can be trusted. Else, an error has been found—the formula may or may not be satisfied.

For model checking logics with higher quantification, it's not immediately clear what a certificate is. The use of game strategies has been proven fruitful:

The relationship between μ-calculus and game theory has been known for a long time, e.g. as parity games [13] or as Ehrenfeucht-Fraïssé games [30]. However, the equivalence between parity games and μ-calculus is often shown using signatures, requiring infinite ordinals and transfinite induction [25, 33]. These constructions of winning strategies happen in a global manner, which makes proofs and implementation quite complicated.

The main contribution of the presented algorithm are that it is defined inductively on the μ-formula and how close the strategy computation is to the fixpoint iteration of set semantics. Strategy computation is straight-forward and can be seen as augmenting the computation of set semantics using fixpoint iteration (see section 3.3).

1.9. Model checking and proof assistants

Model checking is also interesting when it can be used inside larger, deductive proofs. When (sub-)problems can be abstracted as model checking problems, an automated solver for μ-calculus can be a useful tool for proof development. The probably most well-known implementation of this is the μ-calculus model checker part of PVS [28], which has been implemented as a *primitive proof rule*. Predicates which only depend on finite state can be verified automatically by the system, using model checking techniques.

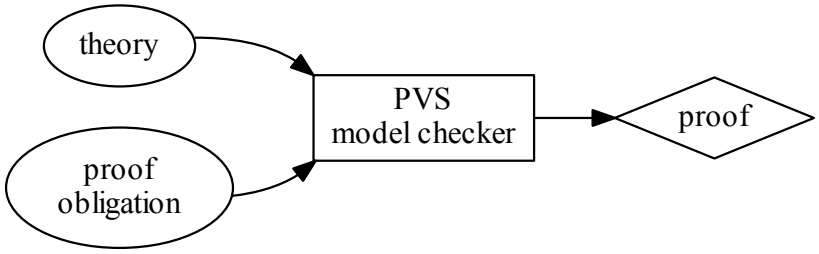


FIGURE 1.3. PVS workflow.

The PVS model checker is used as in fig. 1.3: during development of our proof, we have a *proof obligation* we need to work on that is essentially a μ-calculus formula. The formula refers to finite logic variables and predicates only, thus it can be solved by the model checker. Instead of an explicit LTS construction, the model is given by predicates of the theory. Applying the model checker, we get a Boolean result of whether or not the μ-formula was true. PVS will regard this as a proof for the obligation and we can continue with other parts of the proof.

Since this outcome is not certified, the PVS approach is suspect to bugs which can result in proving false properties, as discussed in the preface.

There have also been efforts to provide a verified implementation of μ-calculus in Coq, named SMC [35]. In contrast to the former, the latter implementation is actually verified using Coq itself and its results can be checked independently using the Coq proof

verifier `coqchk`. It is also possible to generate a standalone, verified μ-calculus checker using Coq's *code extraction* features. Since the whole model checker (including a library for binary decision diagrams) has been implemented in Coq, this was a quite elaborate development.

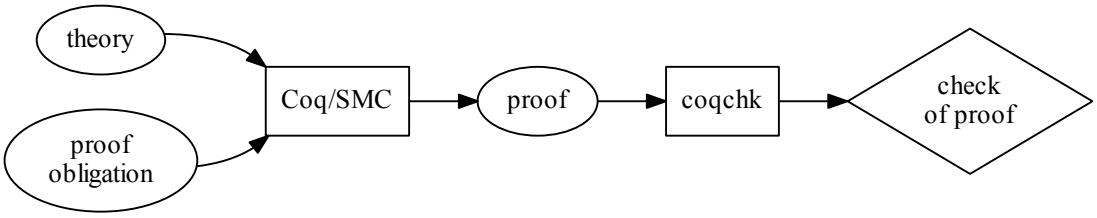


FIGURE 1.4. Coq/SMC workflow.

The Coq/SMC workflow (fig. 1.4) starts quite similar to the PVS one. Again, a proof obligation in the shape of a μ-calculus formula (for a model given by the theory) requires proof. However, the output of SMC is a *proof term*, a term in the Calculus of Inductive Constructions, the foundation of Coq. We can continue proving, and in the end the whole proof can be checked using the standalone verifier `coqchk`. It will then tell us whether our proof is correct, or whether there is a bug in Coq (or SMC!).

We also want to mention CAVA [15], a model checker for LTL implemented and verified using Isabelle/HOL. Since it uses a more restricted temporal logic, it cannot be compared directly to PVS or SMC. It can be extracted to Standard ML and has considerable performance.

1.10. Addressing state space explosion

As stated in the introduction to this chapter, state space explosion can be a serious problem when trying to use model checking. Considering that a C program with six 32-bit `int` variables has more possible states than there atoms in the universe, it is easy to see that tractable problems only have a few bits of possible state that should be explored. However, this is often enough for hardware designs or building blocks of protocols.

There are multiple ways to counter it, in particular two different aspects, which both try to reduce the state space that is visited:

In *symbolic model checking* [24] we avoid building the state graph and describe it using Boolean formulae instead. The transition relation is given by a Boolean implication that is true for allowed moves. States are not merely enumerated but given as a vector of Boolean variables. By using binary decision diagrams (BDDs), these formulae can be stored and manipulated effectively.

For *local model checking* [31], we want to verify that a particular formula is satisfied for a given single state, and are not interested in a global solution. Often, the possible

transitions are determined *on the fly*, thus precomputation or storage of the whole state graph is not required.

Our approach is consistent with symbolic model checking, but it is not yet clear whether the strategy computation can be adopted for local model checking. The strategy checkers we have implemented already operate *on the fly*.

CHAPTER 2

Parity games and strategies

Since our certificates for μ-calculus are winning strategies for parity games, we will quickly review the relevant definitions.

2.1. Parity games

DEFINITION 2.1. *A* parity game *is given by a quintuple $G = (V, V_0, V_1, \to, \Omega)$, where $V = V_0 \sqcup V_1$ is a non-empty set of vertices (called* positions*), precisely the disjoint union of V_0 and V_1. The left-total relation $\to \subseteq V \times V$ defines the possible* moves *between positions. Every position has a* priority *given by $\Omega : V \to \mathbb{N}$ with finite range.*

Parity games are played like this: There are two players, numbered 0 (the proponent) and 1 (the opponent). Players move a token on the game graph along the move relation $\to$. The destination position decides who moves next, depending on whether it is in V_0 and V_1.

A play takes infinitely many turns. In order to find out who wins, we need to take a look at the positions which are reached infinitely often (so-called "recurrent positions") and compute their priorities using Ω. When the *maximum* priority is *even*, player 0 wins, when it is *odd*, player 1 wins.

2.2. Strategies and positional strategies

To analyze how parity games can be won, we study *strategies*:

DEFINITION 2.2. *A* strategy *σ for player i of a parity game is a function $\sigma : V^n \to V, n \in \mathbb{N}$ such that for every $\vec{v} \in V^n, i < n \implies v_i \to v_{i+1}$ is a valid move, $v_n \in V_i$ and $\sigma(\vec{v}) = v'$ with $v_n \to v'$. That is, given a finite prefix of a series of valid moves, it determines the next valid move the player takes.*

We say that player i wins from position $v_0 \in V$ if there exists a strategy σ_i such that *any* play starting from v_0 results in player i winning, no matter which strategy σ_{1-i} the other player pursues.

DEFINITION 2.3. *We call W_i the* winning set of player *i when a strategy σ_i exists such that player i wins from every $v \in W_i$.*

THEOREM 2.1. *Every position in the game is either in the winning set of player 0 or player 1: $V = W_0 \sqcup W_1$.*

PROOF. This follows from a more general result on Borel Determinacy, see Martin [23].

A detailed proof using only elementary game theory an be found in Hofmann and Lange [18, Theorem 15.5]. □

It turns out that knowing the positions before the current one is not required:

DEFINITION 2.4. *A strategy σ is* positional *if it only depends on v_n.*

THEOREM 2.2. *There is a positional strategy such that player i wins from every $v \in W_i$.*

PROOF. C.f. Emerson and Jutla [13]. □

2.3. Certificates for winning strategies

Given a parity game and a suspected positional winning strategy, how can we determine whether the winning strategy really works?

We need to play the parity game and find the recurrent positions. For proponent, we have a winning strategy that says how to move at each turn (if it is not defined, the strategy was bad and we have lost automatically). For opponent, we need to consider every legal move that could happen. Evaluating these moves, we need to check the cycles (i.e. series to moves that return to the same position) we encounter: positions part of a cycle can be visited infinitely often and are therefore relevant to the winning condition. We compute the largest priority in the cycle and check that it has even parity, else there is a way the strategy loses. Once we have confirmed *all* cycles have even parity, the strategy is indeed a winning strategy.

A more efficient approach will be presented in section 3.8.

CHAPTER 3

Certification for μ-calculus

Given these preparations, we can now put them together and certify μ-calculus formulae.

3.1. Model checking as parity game

We will now model the semantics of a μ-calculus formula as a parity game. The proponent (player 0) tries to prove the formula, the opponent (player 1) tries to find a counterexample.

The positions of the parity game are given by (ϕ, s), where ϕ is a subformula of the formula we want to prove and $s \in S$ is a state of the LTS. We use the notation $(\phi, s) \rightsquigarrow (\phi', s')$ for the *possible move* relation.

For the disjunction $\phi_1 \vee \phi_2$, proponent is allowed to choose which side of the formula we consider true:

$$(\phi_1 \vee \phi_2, s) \rightsquigarrow (\phi_1, s)$$
$$(\phi_1 \vee \phi_2, s) \rightsquigarrow (\phi_2, s)$$

In the case of a diamond $\langle a \rangle \phi$, proponent needs to pick a transition $s \xrightarrow{a} s'$ in the LTS:

$$(\langle a \rangle \phi, s) \rightsquigarrow (\phi, s')$$

For the conjunction $\phi_1 \wedge \phi_2$, the opponent chooses which side to play (since both sides need to be satisfiable):

$$(\phi_1 \wedge \phi_2, s) \rightsquigarrow (\phi_1, s)$$
$$(\phi_1 \wedge \phi_2, s) \rightsquigarrow (\phi_2, s)$$

Likewise, for the box $[a]\phi$, opponent can chose any transition $s \xrightarrow{a} s'$ of the LTS (since all possible successor states s' need to be satisfiable):

$$([a]\phi, s) \rightsquigarrow (\phi, s')$$

For quantified formulae, there is only one possible move. Strictly speaking, the proponent moves for $\mu X.\phi$ and opponent moves for $\nu X.\phi$. We assume the "system of equations" interpretation, and thus X inside the formula refers to the quantified formula.

$$(\mu X.\phi, s) \rightsquigarrow (\phi, s)$$
$$(\nu X.\phi, s) \rightsquigarrow (\phi, s)$$

Finally, for variables X and (possibly negated) propositions p, the game loops endlessly:

$$(X, s) \rightsquigarrow (X, s)$$
$$(p, s) \rightsquigarrow (p, s)$$
$$(\neg p, s) \rightsquigarrow (\neg p, s)$$

What remains to properly define the parity game is the priority function $\Omega : \Phi \times S \to \mathbb{N}$.

$$\Omega(\mu X.\phi, s) = 2 \cdot nd(\mu X.\phi) + 1$$
$$\Omega(\nu X.\phi, s) = 2 \cdot nd(\nu X.\phi)$$
$$\Omega(p, s) = \begin{cases} 0 & \text{if } p \in T(s) \\ 1 & \text{if } p \notin T(s) \end{cases}$$
$$\Omega(\neg p, s) = \begin{cases} 1 & \text{if } p \in T(s) \\ 0 & \text{if } p \notin T(s) \end{cases}$$
$$\Omega(X, s) = \begin{cases} 0 & \text{if } s \in \eta(X) \\ 1 & \text{if } s \notin \eta(X) \end{cases}$$
$$\Omega(\phi, s) = 0 \quad \text{otherwise}$$

For X, p and $\neg p$, this definition is intuitive: in the "good" case it should be even, while in the "bad" case it is odd. The outer quantifiers are more powerful than the inner ones, thus their priority depends on $nd(\phi)$, the *nesting depth* of the formula ϕ, defined recursively by:

$$nd(\mu X.\phi) = 1 + nd'(X, \phi)$$
$$nd(\nu X.\phi) = 1 + nd'(X, \phi)$$
$$nd(\phi) = 0 \quad \text{otherwise}$$

$$nd'(X, \phi) = 0 \quad \text{if } X \notin FV(\phi)$$
$$nd'(X, X) = 0$$
$$nd'(X, \phi_1 \vee \phi_2) = \max\{nd'(X, \phi_1), nd'(X, \phi_2)\}$$
$$nd'(X, \phi_1 \wedge \phi_2) = \max\{nd'(X, \phi_1), nd'(X, \phi_2)\}$$
$$nd'(X, [a]\phi) = nd'(X, \phi)$$
$$nd'(X, \langle a \rangle \phi) = nd'(X, \phi)$$
$$nd'(X, \mu Y.\phi) = \max\{nd(\mu Y.\phi), nd'(X, \phi)\}$$
$$nd'(X, \nu Y.\phi) = \max\{nd(\nu Y.\phi), nd'(X, \phi)\}$$

Since the nesting depth is bound by the length of the formula ϕ, Ω has finite range.

3.2. Partial winning strategies

As we saw above, proponent can only make a choice in a few cases of game play. We need to chose which side of disjunction we follow and which state to move to in case of a diamond. The other moves are predetermined or out of proponent's control.

We will define the winning strategy as a partially defined strategy, whose domain will increase during construction. The domain of the partial winning strategy corresponds to the winning set of the player.

DEFINITION 3.1. *A* partial winning strategy *for μ-calculus is a partial function*

$$\begin{aligned} \Sigma : \Phi \times S \rightharpoonup\ & s && \textit{(move to state } s \in S\textit{)} \\ \mid\ & 1 && \textit{(take the left formula)} \\ \mid\ & 2 && \textit{(take the right formula)} \\ \mid\ & * && \textit{(take the only move)} \end{aligned}$$

We denote the function with the empty domain as $\{\}$, and use a set comprehension like syntax $\{x \mapsto y\}$ to construct the partial function with value y at point x. In order to incrementally build strategies, we need to define how to combine them into a bigger one:

DEFINITION 3.2. *Given two winning strategies Σ and Σ', we define the partial winning strategy $\Sigma + \Sigma'$ as*

$$\begin{aligned} (\Sigma + \Sigma')(\phi, s) = \textit{if } & (\phi, s) \in \textit{dom}(\Sigma) \\ & \textit{then } \Sigma(\phi, s) \\ & \textit{else } \Sigma'(\phi, s) \end{aligned}$$

Note that this definition "prefers" the strategy on the left hand side.

3.3. Strategy semantics

We can now describe how to build the strategy with a fixpoint iteration akin to section 1.4.

$$\begin{aligned} \textsc{sem}(X)_\eta &= \{(X, s) \mapsto * \mid s \in \eta(X)\} \\ \textsc{sem}(p)_\eta &= \{(p, s) \mapsto * \mid p \in T(s)\} \\ \textsc{sem}(\neg p)_\eta &= \{(p, s) \mapsto * \mid p \notin T(s)\} \\ \textsc{sem}(\phi \wedge \psi)_\eta &= \textsc{sem}(\phi)_\eta + \textsc{sem}(\psi)_\eta \\ &\quad + \{(\phi \wedge \psi, s) \mapsto * \mid (\phi, s) \in \mathrm{dom}(\textsc{sem}(\phi)_\eta) \\ &\qquad \wedge (\psi, s) \in \mathrm{dom}(\textsc{sem}(\psi)_\eta)\} \\ \textsc{sem}(\phi \vee \psi)_\eta &= \textsc{sem}(\phi)_\eta + \textsc{sem}(\psi)_\eta \\ &\quad + \{(\phi \vee \psi, s) \mapsto 1 \mid (\phi, s) \in \mathrm{dom}(\textsc{sem}(\phi)_\eta)\} \\ &\quad + \{(\phi \vee \psi, s) \mapsto 2 \mid (\psi, s) \in \mathrm{dom}(\textsc{sem}(\psi)_\eta)\} \end{aligned}$$

$$
\begin{aligned}
\textsc{sem}([a]\phi)_\eta &= \textsc{sem}(\phi)_\eta \\
&\quad + \{([a]\phi, s) \mapsto * \mid (\phi, s) \in \mathrm{dom}(\textsc{sem}(\phi)_\eta)\} \\
\textsc{sem}(\langle a\rangle\phi)_\eta &= \textsc{sem}(\phi)_\eta \\
&\quad + \{(\langle a\rangle\phi, s) \mapsto s' \mid s \xrightarrow{a} s' \wedge (\phi, s') \in \mathrm{dom}(\textsc{sem}(\phi)_\eta)\} \\
\textsc{sem}(\nu X.\phi)_\eta &= \textsc{sem}(\phi)_{\eta[X:=sem(\phi,\eta)]} \\
\textsc{sem}(\mu X.\phi)_\eta &= \textsc{iter}_X(\phi, \eta, \{\}) \\
\\
\textsc{iter}_X(\phi, \eta, \Sigma) &= \text{let } \Sigma' := \textsc{sem}(\phi)_{\eta[X:=\mathrm{dom}(\Sigma)]} \text{ in} \\
&\qquad \text{if } \Sigma = \Sigma' \text{ then } \Sigma \text{ else } \textsc{iter}_X(\phi, \eta, \Sigma')
\end{aligned}
$$

Most cases are straight-forward: for true variables X, as well as propositions p which hold, or propositions $\neg p$ which do not hold, we are done.

The conjunction $\phi \wedge \psi$ is built recursively, taking the strategies for the subformulae, and letting opponent decide where to move to iff both operands are in the domain of the partial winning strategy.

Analogously, for a disjunction $\phi \vee \psi$ we can decide which side to choose. After computing the strategy for the subformulae, we just take the first formula that can win.

For the modal operators, we let opponent move however it wants in the case of $[a]\phi$, but we need to grow the domain of the strategy for the fixpoint iteration to work. Thus we add $[a]\phi$ to the strategy when we can win for ϕ, even if this move is never requested. For $\langle a\rangle\phi$, we can pick any fortunate move into our winning domain.

More interesting are $\nu X.\phi$ and $\mu X.\phi$: In the case of $\mu X.\phi$, we apply fixpoint iteration again, just like for the set semantics. By construction, the domain of the partial winning strategy only grows. We can thus start with the empty strategy, and grow its domain by assuming the bound variable X to be true where the strategy already wins. Since our formulae and the transition systems are finite, this fixpoint iteration always terminates (Theorem 1.2).

For $\nu X.\phi$, we cannot use fixpoint iteration, since we would need to start with a "strategy defined everywhere" and remove all cases where it actually does not win, which would not work with this construction. However, we can cheat and actually compute the set where $\nu X.\phi$ holds using the set semantics, then substitute X appropriately and compute the winning strategy under this assumption. Since opponent moves in $\nu X.\phi$ case, this is outside of the control of our winning strategy anyway.

3.4. An example strategy

To make these abstract definitions clearer, let's investigate an example. We are given the LTS in fig. 3.1.

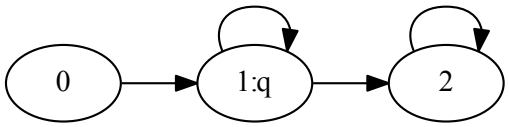


FIGURE 3.1. Example LTS.

The formula we want to check is "q is reachable infinitely often", a fairness condition as presented in section 1.7:

$$\nu X.\mu Y.(q \wedge \langle a \rangle X) \vee \langle a \rangle Y$$

We rewrite this into a system of equations which looks like:

$$\begin{aligned} X &\stackrel{\nu}{=} Y \\ Y &\stackrel{\mu}{=} (q \wedge \langle a \rangle X) \vee \langle a \rangle Y \end{aligned} \tag{1}$$

Our model checker will then compute that the formula is satisfied for states 0 and 1 (obviously, we would be stuck forever in 2 if we ever entered it), and emits this certificate:

```
X 0 -> * |
X 1 -> * |
Y 0 -> * |
Y 1 -> * |
q 1 -> * |
(<a>X) 0 -> X 1 |
(<a>X) 1 -> X 1 |
(<a>Y) 0 -> Y 1 |
(<a>Y) 1 -> Y 1 |
q /\ (<a>X) 1 -> * |
(q /\ (<a>X)) \/ (<a>Y) 0 -> #2 |
(q /\ (<a>X)) \/ (<a>Y) 1 -> #1
```

FIGURE 3.2. Certificate for eq. (1) on fig. 3.1.

The variables `X` and `Y` correspond exactly to X and Y of eq. (1). In order to check this strategy, we need to look at the cycles.

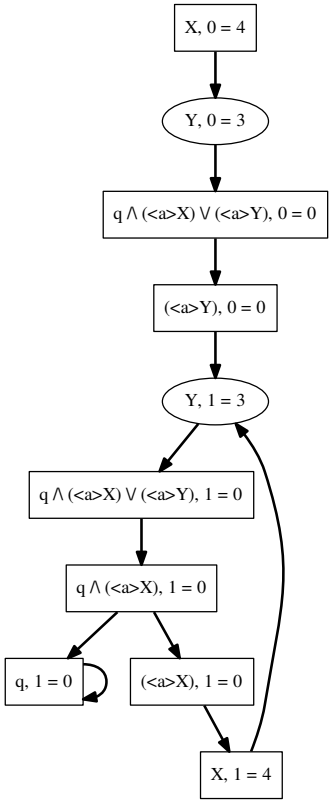


FIGURE 3.3. Run of the strategy shown in fig. 3.2.

It is instructive to trace the run of this strategy, which can be seen in fig. 3.3. Say, we want to prove the formula holds for the state 0. We start at $(X, 0)$ (which has priority 4). Since X is a ν-formula, it's opponent's turn. There is only one move, to $(Y, 0)$ with priority 3. We keep playing the game until we reach the node $(q \wedge (\langle a \rangle X), 1)$, where opponent decides how to move next. If he chooses $(q, 1)$, we have won immediately since the only move from $(q, 1)$ is to $(q, 1)$ again, and $(q, 1)$ has priority 0, which is even. What happens when opponent moves to $(\langle a \rangle X, 1)$ instead? We continue playing until we reach $(Y, 1)$ again, finding a cycle. The node with the largest priority in this cycle is $(X, 1)$, which has priority 4. It is even as well, thus we win the game, no matter how opponent plays. We have shown that fig. 3.2 is indeed a winning strategy.

Since we reached $(X, 1)$ as well during this trace, we have also shown that the formula holds for state 1.

3.5. Checking strategies

Given a partial winning strategy Σ, we want to verify whether it is correct: that is, it wins the parity game starting from the game positions (ϕ, s) for all states $s \in S$ where ϕ actually holds.

Therefore, we need to check whether the parity game winning conditions are true: the highest priority of every position visited infinitely often must be even.

3.6. The move relation

We consider the relation $(\phi, s) \rightsquigarrow (\phi', s')$ of moves in the parity game that are taken by the strategy Σ. In particular, Σ decides which side to take when seeing $\phi_1 \vee \phi_2$ and where to move to when seeing $\langle a \rangle$:

$$
\begin{aligned}
(\mu X.\phi, s) &\rightsquigarrow (\phi, s) \\
(\nu X.\phi, s) &\rightsquigarrow (\phi, s) \\
(X, s) &\rightsquigarrow (X, s) \\
(p, s) &\rightsquigarrow (p, s) \\
(\neg p, s) &\rightsquigarrow (\neg p, s) \\
(\phi_1 \vee \phi_2, s) &\rightsquigarrow (\phi_{\Sigma(\phi_1 \vee \phi_2, s)}, s) \\
(\langle a \rangle \phi, s) &\rightsquigarrow (\phi, \Sigma(\langle a \rangle \phi, s)) \\
(\phi_1 \wedge \phi_2, s) &\rightsquigarrow (\phi_1, s) \\
(\phi_1 \wedge \phi_2, s) &\rightsquigarrow (\phi_2, s) \\
([a]\phi, s) &\rightsquigarrow (\phi, s') \qquad \forall s \xrightarrow{a} s'
\end{aligned}
$$

Due to the nature of the strategy, $\rightsquigarrow$ is uniquely defined for the moves of the proponent, and contains all possible moves for the opponent.

In order to simplify the arguments below, whenever Σ is not defined, we move to a distinguished node $\bot$ for admitting failure ($\bot$ can be modelled as a variable corresponding to the empty set):

$$
\begin{aligned}
(\phi_1 \vee \phi_2, s) &\rightsquigarrow (\bot, s) \qquad \text{if } (\phi_1 \vee \phi_2, s) \notin \mathrm{dom}(\Sigma) \\
(\langle a \rangle \phi, s) &\rightsquigarrow (\bot, s) \qquad \text{if } (\langle a \rangle \phi, s) \notin \mathrm{dom}(\Sigma) \\
(\bot, s) &\rightsquigarrow (\bot, s) \\
\Omega(\bot) &:= 1
\end{aligned}
$$

Thus, $\rightsquigarrow$ is a left-total relation for every partial winning strategy.

In order to verify the strategy, we need to compute the single source reflexive-transitive closure $\rightsquigarrow^*$, starting from the formula and state (ϕ, s) to be checked. Then, we need to look at all cycles $(\phi_i, s_i) \rightsquigarrow (\phi_{i+1}, s_{i+1}) \rightsquigarrow \cdots \rightsquigarrow (\phi_i, s_i)$ and compute their maximum priority $\max_i \Omega(s_i)$. When all cycles have even priority, the formula ϕ holds at s, when one cycle has odd priority, ϕ does not hold at s.

3.7. Simple, recursive checking of strategies

A simple, albeit inefficient way to check this is to enumerate all possible cycles that can happen in the game, and check their highest priority. This can be implemented as a simple recursive traversal, as can be seen in Algorithm 1.

This algorithm has the benefit of being *on the fly*, i.e. no precomputation of the graph is required. We only need to keep track of the current position and be able to generate the possible next moves (which are given by $\rightsquigarrow$). This saves memory usage during checks, and is more efficient if we find a counterexample quickly.

Algorithm 1 Pseudo-code for simple, recursive checking.

function check($\phi : \Phi, s : S$) $\to$ **boolean**:
  findloop($\phi, s, \varnothing$)

function findloop($\phi : \Phi, s : S, V \subseteq S$) $\to$ **boolean**:
  if $(\phi, s) \in V$ **then**
    checkloop($\phi, s, \phi, s, \Omega(\phi, s), \varnothing$)
  else
    for all $(\phi, s) \rightsquigarrow (\phi', s')$:
      findloop($\phi', s', V \cup \{(\phi', s')\}$)

function checkloop($\phi : \Phi, s : S, \phi^* : \Phi, s^* : S, h :$ **integer**$, V \subseteq S$) $\to$ **boolean**:
  if $(\phi, s) \in V$ **then**
    if $\phi = \phi^* \wedge s = s^*$ **then**
      return $h \equiv 0 \mod 2$
    else
      return true *(Not the loop we are interested in.)*
  else
    for all $(\phi, s) \rightsquigarrow (\phi', s')$:
      checkloop($\phi', s', \phi^*, s^*, \max\{h, \Omega(\phi', s')\}, V \cup \{(\phi', s')\}$)

However, all approaches trying to enumerate the cycles cannot run in polynomial time since there can be exponentially many cycles, e.g. consider the "braid graph" shown in fig. 3.4. With each additional "twist", the number of cycles doubles.

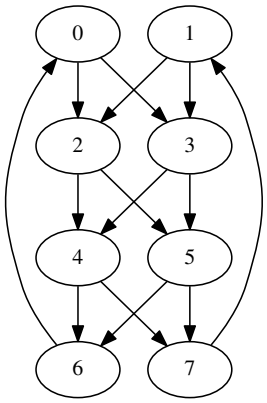


FIGURE 3.4. Braid graph with 3 "twists".

3.8. Efficient checking of strategies with strongly connected components

For an efficient strategy checker, we can use *strongly connected components* (SCCs). In a strongly connected component of a graph, each node is reachable from every other node. Thus every node is on a cycle, and therefore reachable infinitely often. If the highest priority of a node in a SCC is odd, the strategy failed. If it is even, we need to check further, because it could be that we don't visit *this* even node infinitely often, but instead a lower, odd one. Thus, we remove this even node from the graph, recompute the SCCs and recurse.

In particular, the problem is similar to determining non-emptiness of a Streett automaton. Generally, a Streett automaton is a non-deterministic ω-automaton with a finite set of states Q, an alphabet Σ, a transition relation $\Delta : Q \times \Sigma \times Q$, initial states $Q_0 \subseteq Q$ and an acceptance condition, a family of pairs of sets (E_i, F_i). The acceptance condition holds when for every element of E_i that appears infinitely often in the word, an element of the according F_i occurs infinitely often as well.

We shall consider $Q = V$, $\Sigma \subset \mathbb{N}$, Δ according to the possible move relation $\rightsquigarrow$ and $Q_0 = \{(\phi, s) \mid \phi \text{ holds at } s\}$. Then, together with the acceptance condition $\{(\{n\}, \{m \mid m \text{ odd}, n < m < \Omega_{\max} + 1\}) \mid n \text{ even}\}$ this matches all runs that satisfy the winning condition: When the highest recurrent priority h is even, $(\{h\}, \{h+1, \dots\})$ must be violated and the automaton does not accept. Thus, if the automaton is non-empty, its acceptance condition is violated and the strategy has been wrong.

One particularly suitable algorithm for this purpose is described in Duret-Lutz, Poitrenaud, and Couvreur [9] (originating from [8]), which is *on the fly* as well. It uses a variant of Dijkstra's algorithm for detecting strongly connected components [7, Chapter 25].

This algorithm works as follows [27]: During a depth-first search of the graph, we keep a stack of strongly connected components that have been found. Upon finding an edge back into a SCC that closes a cycle, we merge all SCC that are part of the cycle, since using the cycle we can now move from every SCC into any other, i.e. their union is actually one SCC.

This approach is beneficial for emptiness checks, because it traverses the SCC in a depth-first manner as well, allowing for an *on the fly* emptiness check by keeping a list of nodes to be "avoided" when counting down the priorities. Due to the avoidance (which is, effectively, the removal of vertices from the SCC), a single SCC can decompose into multiple, smaller SCC that used to be connected via the avoided node. Due to the nature of the algorithm, we know that these "sub"-SCC are unaffected by nodes visited later, and thus can be visited right away, and then again be forgotten.

3.9. Generating counterexamples

When model checking is applied in the real world, one benefit during development is that the user not only sees whether the specifications hold or not, but it is also possible to compute a specific counterexample which refutes the formula. Often, these counterexamples can be used to find bugs—either in the program, or in the specification.

Winning strategies can also be used to generate counterexamples: Given a formula ϕ which does not hold at a state s of the LTS, we can compute the dual formula ϕ^* (as defined below), which *does hold* at s (because of the determinacy of parity games), and

the winning strategy for ϕ^* tells us *why*. The dual formula can be seen as the negation of ϕ computed using an extension of De Morgan's laws for μ-calculus.

$$\begin{aligned} p^* &= \neg p \\ (\neg p)^* &= p \\ X^* &= X \\ (\phi_1 \wedge \phi_2)^* &= \phi_1^* \vee \phi_2^* \\ (\phi_1 \vee \phi_2)^* &= \phi_1^* \wedge \phi_2^* \\ ([a]\phi)^* &= \langle a \rangle \phi^* \\ (\langle a \rangle \phi)^* &= [a]\phi^* \\ (\mu X.\phi)^* &= \nu X.\phi^* \\ (\nu X.\phi)^* &= \mu X.\phi^* \end{aligned}$$

3.10. Verification of counterexamples

When the partial strategy is undefined for a certain formula at a given position, checking fails immediately. This implies that a formula can hold, but the winning strategy is incomplete (for which reasons whatsoever), and therefore the checker thinks it does not hold. Thus, by default a check only verifies that when a formula holds, it really holds. In order to check a formula *really does not hold*, we need to check that the negation of the formula really holds.

3.11. Complexity analysis

Generally, satisfiability for μ-calculus is EXPTIME-complete [12].

Given a **fixed** formula ϕ and state space S, the strategy can be computed in polynomial time. The size of a single partial winning strategy is bound by $O(|S| \cdot |\phi| \cdot \log(|S|))$, since we have for every state $s \in S$ at most $|\phi|$ subformulae where the strategy can be defined. In case the strategy needs to choose a move, we need $\log(|S|)$ bits to specify the successor state. The recursion depth of the fixpoint iteration is bounded by $O(|\phi|)$, due to the number of quantifiers. Since fixpoint iteration does not require keeping more than the last iteration state, we get a space bound of $O(|S| \cdot \log(|S|) \cdot |\phi|^2)$.

Verification of the winning strategies can be done in low polynomial time, when using the approach with SCC. By using a variant of *Dijkstra's algorithm* [7, Chapter 25] for finding SCC, which runs in linear time—precisely $O(|V| + |E|)$—, the algorithm proposed by Duret-Lutz, Poitrenaud, and Couvreur [9] runs linear in the number of transitions multiplied with the number of acceptance pairs.

In our application, the parity game graph for a formula ϕ and a set of states S has $O(|S| \cdot |\phi|)$ vertices (where $|\phi|$ is the length of the formula ϕ, and linear in the size of the Fischer-Ladner closure, which is a superset of all formulae that can appear as a state formula [32]), but since each node can have at most $|S|$ outbound edges at most $O(|S|^2 \cdot |\phi|)$ edges. The number of acceptance pairs is linear to the number of quantifiers used in ϕ, which is linear in $|\phi|$ as well. Thus, we get a total time complexity of $O(|S|^2 \cdot |\phi|^2)$ for verifying a winning strategy.

CHAPTER 4

Implementation and optimization

4.1. Implementation

Micromu is implemented as an OCaml 4.01 program totaling up to about 1200 lines of code, with no additional external dependencies. It only uses standard data structures like sets and maps. Details on how to use it can be found in Appendix A.

The implementation is mostly straight-forward from the exposition earlier. Systems of equations are used to avoid memory allocation during fixpoint iteration. *On the fly* checking avoids materializing the parity game in memory. Therefore, Micromu performance is CPU-bound. There is currently no support for parallelism.

The implementation tries to be modular and consists of six parts: μ-calculus formulae, LTS, winning strategies, the simple checker, the efficient checker, and the main driver.

μ-calculus formulae are represented using a simple algebraic data type that represents the syntax as given in definition 1.2. Using the on-board tools `ocamllex` and `ocamlyacc`, we have implemented a parser for the textual syntax (details about this in appendix A.3). The μ-calculus module also can convert between plain formulae and their representation as system of equations and vice versa (section 1.5), as well as compute the set semantics as shown in section 1.4.

The LTS module represents labelled transition systems. States are numbered, and transitions are stored as sets of integer pairs, indexed by their action. Again, there is a parser to read LTS from text files (see appendix A.2 for the format used).

Winning strategies are kept abstract, only exposing an `assoc` function to look up a possible move given a model and particular state. Computation of winning strategies happens by fixpoint iteration using a recursive function, just like presented in section 3.3.

Since there are two different checkers, we have introduced an interface to ensure they are exchangeable. We have implemented the simple strategy checker as well as an efficient, polynomial-time checker using Streett automata.

Finally, everything is connected using the main driver, which parses the command line arguments, reads data structures, and then computes the winning strategy and verifies it with one of the checkers.

4.2. Optimization of checking

The simple, recursive checker—albeit having exponential runtime in worst case—works pretty well. When verifying a property for many states, it often recomputes many steps needlessly, thus caching of these results (nodes known as “good” or “bad”) was implemented and the algorithm can terminate early upon finding such a node.

The SCC-based algorithm does not need this because it checks all states at once. It has, however, more overhead than the simple checker and is often slower.

4.3. Optimization of the implementation

No special considerations for performance have been taken during the initial implementation of Micromu. Once the program worked, the OCaml profiler [21, Chapter 17] was used for execution count profiling to determine hotspots in the code. Additionally, Linux `perf` [34] provided useful information about the OCaml runtime and was used for time profiling.

Micromu uses sets of integers heavily (for sets of states of a LTS), as well as sets of pairs of integers (for relations). By default, OCaml uses *polymorphic compare* for these (known as `compare_val` in the runtime) which is very expensive in relation to simply comparing a integer tuple. Specializing these data structures and providing a monomorphic comparison resulted in a major speedup.

During development, representing winning strategies as association lists proved to be useful for debugging, but the cost of linear lookup as well as accumulation of unreachable elements is prohibitive for actual use. Winning strategies are now implemented as simple maps, which guarantees efficient lookup and cheap merging.

4.4. Benchmarks

We provide three benchmarks that give insight into the algorithms at work.

Table 1. Runtimes on a AMD Phenom II X4 920 (2.80 GHz)

Problem	States	sem [s]	SEM [s]	Check [s]	Check SCC [s]
Flower 8	16	0.179	0.203	0.009	0.040
Flower 10	20	3.166	1.960	0.071	0.419
Flower 12	24	32.269	11.688	0.287	2.061
Flower 14	28	320.931	61.733	1.298	10.829
Flower 16	32	3196.043	326.666	6.131	58.871
Circle 100	100	0.003	0.001	0.001	0.001
Circle 1000	1000	0.109	0.018	0.005	0.006
Circle 10000	10000	15.763	3.398	0.054	0.057
Circle 100000	100000	2027.584	811.041	0.581	0.582
Braid 6	12	0.001	0.005	1.282	0.009
Braid 8	16	0.002	0.003	31.062	0.013
Braid 10	20	0.002	0.006	711.002	0.020
Braid 100	200	0.663	0.993	—	3.674
G_1	15	34.862	79.342	0.050	1.538

The "Flower" benchmark is a parity game (from Buhrke, Lescow, and Vöge [4]) translated into a μ-calculus formula, which shows the exponential complexity of deciding μ-calculus. However, the certificates can be checked in polynomial time.

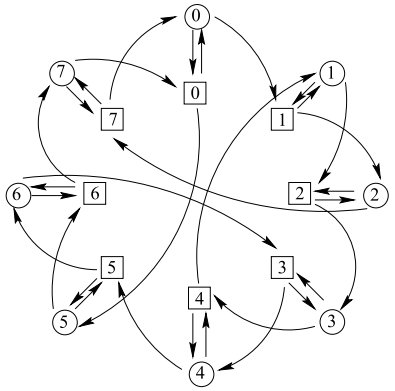


FIGURE 4.1. "Flower" with $n = 8$. (Illustration taken from [4].)

The "Circle" benchmark measures the overhead of the algorithms. It consists of a single cycle that needs to be traversed to check for a property. In this case, runtime is linear, and checking is very fast.

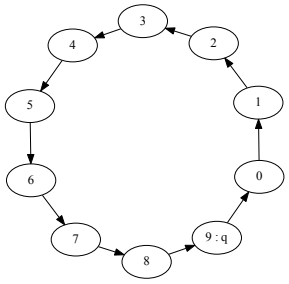


FIGURE 4.2. "Circle" with $n = 10$.

The "Braid" benchmark focuses on checking complexity. This family of graphs has exponentially many cycles (see page 24), thus the simple checker requires exponential time. The SCC algorithm is not affected and checks these graphs in linear time.

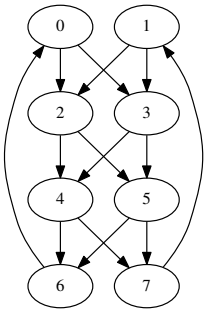


FIGURE 4.3. "Braid" with $n = 4$.

In his PhD thesis, Oliver Friedmann suggests a family G_n of parity games which have exponential behavior [17]. Only the first one is tractable with the current implementation. Calculation of G_2 has been aborted after several days of computation.

CHAPTER 5

Perspectives

5.1. Summary of the work

We have shown how to implement a model checker for μ-calculus that certifies its results using winning strategies, which are computed by fixpoint iteration in a straightforward manner.

In order to do this, we have shown the algorithms of Hofmann and Rueß [19] to be viable and improved upon them: the use of systems of equations not only makes our implementation run without consuming memory for formula representation at all, but also simplifies the specification of fixpoint iteration a lot.

Further, we have implemented two checkers for these winning strategies, one which is simple to understand but has worst-case exponential complexity, and one based on strongly connected components and emptiness of Streett automata (which has been proposed in [19] without further detail). Both have the benefit of being *on the fly* and are thus suitable for larger models as well as symbolic approaches.

5.2. Further possible optimizations

Micromu has been written as a prototype model checker for experimenting with winning strategies, not as an industrial-strength tool aimed at high performance. Only asymptotic algorithmic performance improvements have been considered.

For simplicity and clarity, many techniques well-known in the model checking world are not implemented, such as the use of binary decision diagrams, incremental construction of transitions, forward analysis, and frontier set simplification. These all are possible to use in μ-calculus model checkers and have been successfully implemented [3, 2].

An interesting way to make computation of winning strategies more efficient would be to use a *general-purpose fixpoint solver* that can use sophisticated algorithms and heuristics to optimize the μ- and ν-cases of the algorithm. There are fixpoint solvers which work on arbitrary lattices, however, the exact construction of a lattice representing partial winning strategies remains an open problem.

Instead of making Micromu a full-fledged model checker, it may be more reasonable to instrument a more advanced implementation of μ-calculus to compute winning strategies as well.

5.3. Formal verification

As mentioned in the preface, having a formal proof of the strategy checker will result in a verified model checker for μ-calculus.

Since strategy checking should run in polynomial time, an algorithm similar to the SCC-based one will have to be implemented and verified. Unfortunately, the SCC

algorithm as used is quite complex when expressed as a functional program. Perhaps the model checker could compute these SCC as part of strategy computation, reducing further verification effort.

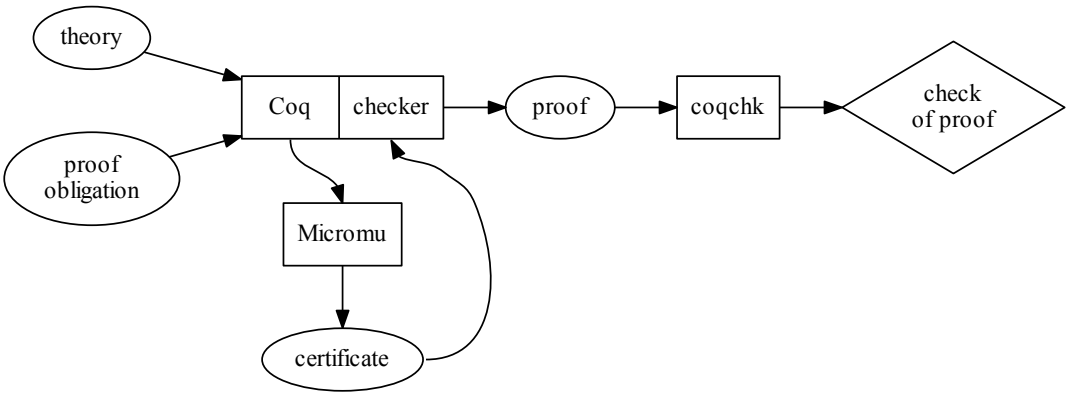


FIGURE 5.1. Possible future Micromu workflow.

Figure 5.1 shows how we imagine a future workflow using Micromu: again, we are in a setting like fig. 1.4. However, the model checker is still an external tool. Coq calls it, passing appropriate transcriptions of the model and μ-formula in question. Then, Micromu computes the certificate, which is verified by the strategy checker, that runs inside Coq. It checks the computed strategy and emits a proof term, which again can be checked separately using `coqchk` to guarantee everything is correct.

5.4. Comparison to verified model checkers

Esparza et al. [15] have implemented and formally verified a LTL checker named CAVA in Isabelle/HOL. It is meant to be used as a standalone model checker, and not as a proof tactic. They recognize implementing a full model checker is more work than just verifying a strategy checker, but claim not requiring potentially large certificates to their advantage. Indeed, certificate size can be problematic, especially with a large state space.

Compression of winning strategies could be an interesting area for further research. In particular, techniques from BDD-based symbolic model checking could be used to optimize storage and manipulation of winning strategies and their domains.

SMC, by Verma [35], is an implementation of a BDD library and a complete symbolic μ-calculus model checker for Coq. It computes proofs by reflection into proof terms and thus does not require a separate certification step. This can be seen as an advantage, but it also means the implementation is harder to optimize, while keeping the correctness proof.

APPENDIX A

Appendix: Using Micromu

A.1. Command line arguments

Micromu is used as a simple command line program. It requires two arguments, a file containing the LTS and a file containing the μ-calculus formula to be checked. An optional parameter `-c` can be passed as a first argument to compute a counterexample as explained in section 3.9.

When run, Micromu will output copious information. First, the formula is rewritten as a system of equations, which is printed. Then, set semantics of the formula are computed and the result outputted. Next, strategy semantics is computed and the winning strategy gets printed. Finally, the verifier checks the strategy and prints whether verification has passed or not.

To the casual user, only the last two lines of output matter: the result and whether it is correct.

With the command line flag `-s`, the SCC-based strategy checker from section 3.8 can be enabled. By default, the simple strategy checker (section 3.7) is used.

With the command line flag `-d`, the strategy run is output as a Graphviz *dot(1)* file [10] (this will look like fig. 3.3).

A.2. LTS file format

Micromu reads LTS in a format which is a variant of Aldebaran syntax [16, 5].

An LTS file starts with a header line of using the following syntax:

`des (`*initial-state*`,`*number-of-transitions*`,`*number-of-states*`)`

Next, there is one line per transition, in arbitrary order. Each line has the following syntax:

`(`*from-state*`,"`*label*`",`*to-state*`)`

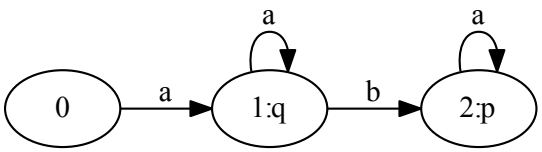
Finally (and as extension to Aldebaran format), you can specify the atomic propositions, one per line:

`"`*prop*`",`*state*

Lines starting with `#` are ignored.

For example:

```
des (0,4,3)
(0,"a",1)
(1,"a",1)
(1,"b",2)
(2,"a",2)
"q",1
"p",2
```



A.3. MU file format

Mu formula files contain an ASCII-fied representation of the syntax presented in definition 1.2. Lines starting with # are ignored. Variables start with uppercase letters, propositions with lowercase letters.

For example, the formula G_1 from the benchmark:

```
mu Z20. mu Z18. nu Z17. nu Z15. mu Z14. mu Z12. mu Z10.
nu Z9. mu Z8. nu Z7. mu Z6. mu Z4. nu Z3. mu Z2. nu Z1.
(y0 /\ x20 /\ <a>Z20) \/ (y0 /\ x18 /\ <a>Z18) \/ (y0 /\ x17 /\ <a>Z17) \/
(y0 /\ x15 /\ <a>Z15) \/ (y0 /\ x14 /\ <a>Z14) \/ (y0 /\ x12 /\ <a>Z12) \/
(y0 /\ x10 /\ <a>Z10) \/ (y0 /\ x9 /\ <a>Z9) \/ (y0 /\ x8 /\ <a>Z8) \/
(y0 /\ x7 /\ <a>Z7) \/ (y0 /\ x6 /\ <a>Z6) \/ (y0 /\ x4 /\ <a>Z4) \/
(y0 /\ x3 /\ <a>Z3) \/ (y0 /\ x2 /\ <a>Z2) \/ (y0 /\ x1 /\ <a>Z1) \/
(y0 /\ x20 /\ [a]Z20) \/ (y0 /\ x18 /\ [a]Z18) \/ (y0 /\ x17 /\ [a]Z17) \/
(y0 /\ x15 /\ [a]Z15) \/ (y0 /\ x14 /\ [a]Z14) \/ (y0 /\ x12 /\ [a]Z12) \/
(y0 /\ x10 /\ [a]Z10) \/ (y0 /\ x9 /\ [a]Z9) \/ (y0 /\ x8 /\ [a]Z8) \/
(y0 /\ x7 /\ [a]Z7) \/ (y0 /\ x6 /\ [a]Z6) \/ (y0 /\ x4 /\ [a]Z4) \/
(y0 /\ x3 /\ [a]Z3) \/ (y0 /\ x2 /\ [a]Z2) \/ (y0 /\ x1 /\ [a]Z1)
```

TABLE 1. Transcription rules for Micromu formulae.

Formula	ASCII representation		
$\langle a \rangle$	`<a>`		
$[a]$	`[a]`		
$a \wedge b$	`a /\ b`	or	`a && b`
$a \vee b$	`a \/ b`	or	`a || b`
$\neg a$	`~a`		
$\mu X.a$	`mu X.a`		
$\nu X.a$	`nu X.a`		

APPENDIX B

Appendix: Micromu Source Code

B.1. Labelled transition systems

LISTING B.1. lts.ml

```
module States = Set.Make(struct
  type t = int
  let compare a1 a2 = a1 - a2
end)

module Actions = Map.Make(String)

type action = Actions.key

module Props = Map.Make(String)

type prop = Props.key

type state_set = States.t
type state = States.elt

module StateState = Set.Make(struct
  type t = int * int
  let compare (a1,b1) (a2,b2) = if a1 = a2 then b1 - b2 else a1 - a2
end)

type t = {
  states: state_set;
  by_action: StateState.t Actions.t;
  by_prop: States.t Props.t
}

let empty _ = States.empty
let all_states t = t.states

let rec gather_states = function
  | [] -> States.empty
  | (_,s,t)::xs -> States.add t @@ States.add s @@ gather_states xs

let rec group_states = function
  | [] -> Actions.empty
```

```
  | (a,s,t)::xs ->
    Actions.merge (fun _ left right ->
      match left, right with
      | Some l, Some r -> Some (StateState.union l r)
      | Some l, None -> Some l
      | None, Some r -> Some r
      | None, None -> None
    ) (group_states xs) (Actions.singleton a (StateState.singleton (s,t)))

let rec group_props = function
  | [] -> Props.empty
  | (a,s)::xs ->
    Actions.merge (fun _ left right ->
      match left, right with
      | Some l, Some r -> Some (States.union l r)
      | Some l, None -> Some l
      | None, Some r -> Some r
      | None, None -> None
    ) (group_props xs) (Props.singleton a (States.singleton s))

let add_default_props lts =
  { lts with
    by_prop =
      Props.add "false" (empty lts) @@
        Props.add "true" (all_states lts) lts.by_prop }

let from_lists l p : t =
  add_default_props
    { states = gather_states l;
      by_action = group_states l;
      by_prop = group_props p }

let prop_true lts prop =
  Props.find prop lts.by_prop

let prop_false lts prop =
  States.diff (all_states lts) (prop_true lts prop)

let pre lts a set =
  let actions = Actions.find a lts.by_action in
  States.filter (fun s ->
    States.exists (fun t -> StateState.mem (s,t) actions) set) (all_states lts)

let pre' lts a set =
  let actions = Actions.find a lts.by_action in
  States.filter (fun s ->
    States.for_all (fun t ->
      (not (StateState.mem (s,t) actions)) || States.mem t set)
      (all_states lts)) (all_states lts)
```

```
let succ lts a s =
  let actions = Actions.find a lts.by_action in
  States.filter (fun t -> StateState.mem (s,t) actions) (all_states lts)
```

B.2. μ-calculus formulae

Listing B.2. mu.mli

```
open Lts

type var = string
type props = string

type mu_form =
  | X of var
  | P of props
  | NotP of props
  | Box of action * mu_form
  | Diamond of action * mu_form
  | Conj of mu_form * mu_form
  | Disj of mu_form * mu_form
  | Mu of var * mu_form
  | Nu of var * mu_form

val negate : mu_form -> mu_form
val pretty : mu_form -> string

type env
val empty_env : env
val subst : env -> var -> state_set -> env
val lookup : env -> var -> state_set option

type mu_map_form
type munu = MU | NU
val study : mu_form -> mu_form * mu_map_form
val unmmf : mu_map_form -> mu_form -> mu_form
val lookup_mmf : mu_map_form -> var -> (munu * int * mu_form) option
val pretty_mmf : mu_map_form -> string

val sem : Lts.t -> mu_map_form -> mu_form -> env -> state_set
```

Listing B.3. mu.ml

```
type var = string
type props = string

type mu_form =
| X of var
| P of props
```

```
| NotP of props
| Box of Lts.action * mu_form
| Diamond of Lts.action * mu_form
| Conj of mu_form * mu_form
| Disj of mu_form * mu_form
| Mu of var * mu_form
| Nu of var * mu_form

type truth = Lts.state -> props -> bool

module VarMap = Map.Make(String)

type munu = MU | NU

type mu_map_form = (munu * int * mu_form) VarMap.t

type env = Lts.state_set VarMap.t

let empty_env = VarMap.empty

let rec pretty = function
  | X v -> v
  | P p -> p
  | NotP p -> "~" ^ p
  | Box (a, f) -> Printf.sprintf "([%s]%s)" a (pretty f)
  | Diamond (a, f) -> Printf.sprintf "(<%s>%s)" a (pretty f)
  | Conj (f1, f2) -> Printf.sprintf "%s /\\ %s" (pretty f1) (pretty f2)
  | Disj (f1, f2) -> Printf.sprintf "%s \\/ %s" (pretty f1) (pretty f2)
  | Mu (v, f1) -> Printf.sprintf "mu %s. %s" v (pretty f1)
  | Nu (v, f1) -> Printf.sprintf "nu %s. %s" v (pretty f1)

let subst eta x u =
  VarMap.add x u eta

let lookup eta x =
  try Some (VarMap.find x eta)
  with Not_found -> None

let lookup_mmf mmf x =
  try Some (VarMap.find x mmf)
  with Not_found -> None

let study (f : mu_form) : (mu_form * mu_map_form) =
  let rec add f depth mmf = match f with
    | X _ | P _ | NotP _ -> (f, mmf)
    | Box (a, f) ->
      let f', mmf' = add f (depth+1) mmf in
      (Box (a, f'), mmf')
    | Diamond (a, f) ->
```

```
      let f', mmf' = add f (depth+1) mmf in
      (Diamond (a, f'), mmf')
    | Conj (f1, f2) ->
      let f', mmf' = add f1 (depth+1) mmf in
      let f'', mmf'' = add f2 (depth+1) mmf' in
      (Conj (f', f''), mmf'')
    | Disj (f1, f2) ->
      let f', mmf' = add f1 (depth+1) mmf in
      let f'', mmf'' = add f2 (depth+1) mmf' in
      (Disj (f', f''), mmf'')

    | Mu (v, f1) ->
      let f', mmf' = add f1 (depth+1) mmf in
      let mmf'' = VarMap.add v (MU, depth, f') mmf' in
      (X v, mmf'')

    | Nu (v, f1) ->
      let f', mmf' = add f1 (depth+1) mmf in
      let mmf'' = VarMap.add v (NU, depth, f') mmf' in
      (X v, mmf'')
  in
  add f 1 VarMap.empty

let unmmf mmf =
  let rec unmmf' d = function
    | X v ->
      (match lookup mmf v with
      | Some (MU, d', f) when d < d' -> Mu (v, unmmf' d' f)
      | Some (NU, d', f) when d < d' -> Nu (v, unmmf' d' f)
      | _ -> X v)
    | Box (a, f1) -> Box (a, unmmf' d f1)
    | Diamond (a, f1) -> Diamond (a, unmmf' d f1)
    | Conj (f1, f2) -> Conj (unmmf' d f1, unmmf' d f2)
    | Disj (f1, f2) -> Disj (unmmf' d f1, unmmf' d f2)
    | x -> x
  in
  unmmf' 0

let pretty_mmf mmf =
  String.concat "" @@
  VarMap.fold (fun k (q,d,f) l ->
    l @ [ Printf.sprintf "%s =%s/%d= %s\n"
            k
            (match q with | MU -> "mu" | NU -> "nu")
            d
            (pretty f) ]) mmf []
```

```
let rec negate = function
  | X v -> X v
  | P p -> NotP p
  | NotP p -> P p
  | Box (a, f) -> Diamond (a, negate f)
  | Diamond (a, f) -> Box (a, negate f)
  | Conj (f1, f2) -> Disj (negate f1, negate f2)
  | Disj (f1, f2) -> Conj (negate f1, negate f2)
  | Mu (v, f1) -> Nu (v, negate f1)
  | Nu (v, f1) -> Mu (v, negate f1)

let rec sem lts mmf (f : mu_form) (eta : env) : Lts.state_set =
  match f with
  | X v ->
    (match lookup eta v with
    | Some x -> x
    | None ->
      match lookup mmf v with
      | Some (MU, _, f) -> sem_iter lts mmf v f eta (Lts.empty lts)
      | Some (NU, _, f) -> sem_iter lts mmf v f eta (Lts.all_states lts)
      | None -> failwith ("unbound variable "^v))
  | Conj (f1, f2) -> Lts.States.inter (sem lts mmf f1 eta) (sem lts mmf f2 eta)
  | Disj (f1, f2) -> Lts.States.union (sem lts mmf f1 eta) (sem lts mmf f2 eta)
  | Box (a, f) -> Lts.pre' lts a (sem lts mmf f eta)
  | Diamond (a, f) -> Lts.pre lts a (sem lts mmf f eta)
  | P p -> Lts.prop_true lts p
  | NotP p -> Lts.prop_false lts p

and sem_iter lts mmf x f eta u =
  let u_p = sem lts mmf f (subst eta x u) in
  if Lts.States.equal u u_p
  then u
  else
    (
      Printf.eprintf "%d -> %d\n%!" (Lts.States.cardinal u)
        (Lts.States.cardinal u_p);
      sem_iter lts mmf x f eta u_p
    )
```

B.3. Partial winning strategies

Listing B.4. strat.mli

```
open Mu
open Lts

type ext_state =
| Step of (mu_form * state)
| One
| Two
| Star

type partial_strat

val pretty : ?sep:string -> partial_strat -> string
val assoc : mu_form * state -> partial_strat -> ext_state option

val sem : Lts.t -> mu_map_form -> mu_form -> env -> partial_strat
```

Listing B.5. strat.ml

```
open Mu

type ext_state =
| Step of (mu_form * Lts.state)
| One
| Two
| Star

module PartialStrat = Map.Make(struct
  type t = mu_form * Lts.state
  let compare = compare
end)

type partial_strat = ext_state PartialStrat.t

let (+++) st1 st2 =
  PartialStrat.merge (fun _ left right ->
    match left, right with
    | Some l, Some _ -> Some l
    | Some l, None -> Some l
    | None, Some r -> Some r
    | None, None -> None) st1 st2

let (++) st (k, v) =
  if PartialStrat.mem k st
  then st
  else PartialStrat.add k v st
```

```
let empty_strat =
  PartialStrat.empty

let dom s = List.map fst (PartialStrat.bindings s)

let in_dom k s = PartialStrat.mem k s

let states_by_form f s = List.map snd @@ List.filter (fun (f', _) -> f = f') s

let rec states_from_list = function
  | x :: xs -> Lts.States.add x (states_from_list xs)
  | [] -> Lts.States.empty

let pretty_ext_state = function
  | Step (f, s) -> Printf.sprintf "%s %d" (Mu.pretty f) s
  | One -> "#1"
  | Two -> "#2"
  | Star -> "*"

let pretty ?(sep=" | ") st = String.concat sep @@
  List.map (fun ((f, s), t) ->
    Printf.sprintf "%s %d -> %s" (Mu.pretty f) s (pretty_ext_state t))
  (PartialStrat.bindings st)

let assoc (f,s) st =
  try Some (PartialStrat.find (f,s) st)
  with Not_found -> None

(* extensional equality of two partial strategies.  *)
let strat_ext_eq st1 st2 =
  PartialStrat.equal (=) st1 st2

(* merge st1 and st2, keeping the domain of st2 but the values of st1.  *)
let strat_merge st1 st2 =
  PartialStrat.merge (fun _ left right ->
    match left, right with
    | Some l, Some _ -> Some l
    | Some _, None -> None
    | None, Some r -> Some r
    | None, None -> None) st1 st2

let string_of_set s =
  Lts.States.fold (fun e a -> a ^ " " ^ string_of_int e) s ""

let rec sem lts mmf (f : mu_form) (eta : env) : partial_strat = match f with
  | P p -> Lts.States.fold (fun s strat -> strat ++ ((P p, s), Star))
                           (Lts.prop_true lts p) empty_strat
  | NotP p -> Lts.States.fold (fun s strat -> strat ++ ((NotP p, s), Star))
                              (Lts.prop_false lts p) empty_strat
```

```
| Conj (f1, f2) ->
  let s1 = sem lts mmf f1 eta in
  let s2 = sem lts mmf f2 eta in
  s1 +++ s2 +++
    Lts.States.fold (fun s strat ->
      if in_dom (f1,s) s1 && in_dom (f2,s) s2
      then strat ++ ((Conj (f1, f2), s), Star)
      else strat) (Lts.all_states lts) empty_strat
| Disj (f1, f2) ->
  let s1 = sem lts mmf f1 eta in
  let s2 = sem lts mmf f2 eta in
  s1 +++ s2 +++
    List.fold_left (fun strat s ->
      strat ++ ((Disj (f1, f2), s), One)
    ) empty_strat (states_by_form f1 (dom s1))
    +++
    List.fold_left (fun strat s ->
      strat ++ ((Disj (f1, f2), s), Two)
    ) empty_strat (states_by_form f2 (dom s2))
| Box (a, f1) ->
  let s1 = sem lts mmf f1 eta in
  let actions = Lts.Actions.find a lts.Lts.by_action in
  s1 +++
    Lts.States.fold (fun s strat ->
      if Lts.StateState.for_all (fun (ss,s') ->
        s <> ss || in_dom (f1,s') s1) actions
      then strat ++ ((Box (a, f1), s), Star)
      else strat
    ) (Lts.all_states lts) empty_strat
| Diamond (a, f1) ->
  let s1 = sem lts mmf f1 eta in
  Printf.printf "Diamond says: %s\n" (pretty s1);
  let actions = Lts.Actions.find a lts.Lts.by_action in
  let domain = states_from_list (states_by_form f1 (dom s1)) in
  let dst = Lts.StateState.filter (fun (_,s') ->
    Lts.States.mem s' domain) actions in
  s1 +++
    Lts.StateState.fold (fun (s,s') strat ->
      strat ++ ((Diamond (a, f1), s), Step (f1, s'))) dst empty_strat
| X v ->
  match lookup eta v with
  | Some states ->
    Lts.States.fold (fun s strat -> strat ++ ((X v, s), Star))
                    states empty_strat
  | None ->
    match lookup_mmf mmf v with
    | Some (MU, _, f) -> mu_step lts mmf f eta v
    | Some (NU, _, f) -> nu_step lts mmf f eta v
    | None -> failwith ("unbound variable "^v)
```

```
and mu_step lts mmf f eta v =
  let rec step mmf eta prev =
    Printf.printf "mu_step env %s:\n" v;
    let eta' = subst eta v (states_from_list (states_by_form f (dom prev))) in
    (sem lts mmf f eta'), eta'
  and iter mmf eta prev =
    let next, eta' = step mmf eta prev in
    (* make them agree on the image *)
    let next = strat_merge prev next in
    if strat_ext_eq prev next
    then next
    else
      (Printf.printf "strat %d -> %d\n%!"
                     (PartialStrat.cardinal prev) (PartialStrat.cardinal next);
       iter mmf eta' next)
  in
  iter mmf eta empty_strat

and nu_step lts mmf f eta v =
  Printf.eprintf "f %s\n%!" (Mu.pretty f);
  let u = cache_sem lts mmf (X v) eta in
  Printf.printf "result: %s\n%!"  (string_of_set u);
  sem lts mmf f (subst eta v u)

and cache = Hashtbl.create 127

and cache_sem lts mmf f eta =
  try
    let r = Hashtbl.find cache (f) in
    Printf.eprintf "CACHE HIT %s\n%!" (Mu.pretty f);
    r
  with Not_found ->
    set_sem lts mmf f eta

and set_sem lts mmf (f : mu_form) (eta : env) : Lts.state_set =
  let rec iter lts mmf x f eta u =
    let u_p = set_sem lts mmf f (subst eta x u) in
    if Lts.States.equal u u_p
    then (Hashtbl.add cache (f) u; u)
    else iter lts mmf x f eta u_p
  in
  match f with
  | X v ->
    (match lookup eta v with
    | Some x -> x
    | None ->
      match lookup_mmf mmf v with
      | Some (MU, _, f) -> iter lts mmf v f eta (Lts.empty lts)
```

```
      | Some (NU, _, f) -> iter lts mmf v f eta (Lts.all_states lts)
      | None -> failwith ("unbound variable "^v))
  | Conj (f1, f2) -> Lts.States.inter (cache_sem lts mmf f1 eta)
                                      (cache_sem lts mmf f2 eta)
  | Disj (f1, f2) -> Lts.States.union (cache_sem lts mmf f1 eta)
                                      (cache_sem lts mmf f2 eta)
  | Box (a, f) -> Lts.pre' lts a (cache_sem lts mmf f eta)
  | Diamond (a, f) -> Lts.pre lts a (cache_sem lts mmf f eta)
  | P p -> Lts.prop_true lts p
  | NotP p -> Lts.prop_false lts p
```

B.4. Strategy checker interface

LISTING B.6. checker_intf.ml

```
module type Checker = sig
open Mu
open Strat
open Lts

val check : Lts.t -> mu_map_form -> partial_strat ->
            state -> mu_form -> state_set * bool
val gendot : Lts.t -> mu_map_form -> partial_strat ->
             state_set -> mu_form -> unit
end
```

B.5. Simple strategy checker

LISTING B.7. checker.ml

```
open Mu
open Strat

let rec free v = function
  | X v' when v = v' -> true
  | Conj (f1, f2)
  | Disj (f1, f2) -> free v f1 || free v f2
  | Box (_, f1)
  | Diamond (_, f1) -> free v f1
  | Mu (v', f1)
  | Nu (v', f1) when v != v' -> free v f1
  | _ -> false

let rec nd (f : mu_form) =
  let rec nd' v f =
    if free v f then
      match f with
      | Conj (f1, f2)
      | Disj (f1, f2) -> max (nd' v f1) (nd' v f2)
```

```
      | Box (_, f1)
      | Diamond (_, f1) -> nd' v f1
      | Mu (_, f1)
      | Nu (_, f1) -> max (nd f) (nd' v f1)
      | _ -> 0
    else
      0
  in
  match f with
  | Mu (v, f1)
  | Nu (v, f1) -> (nd' v f1) + 1
  | _ -> 0

let prio' lts eta s = function
  | Mu (v, f1) -> 2 * nd (Mu (v, f1)) + 1
  | Nu (v, f1) -> 2 * nd (Nu (v, f1))
  | P p -> if Lts.States.mem s (Lts.prop_true lts p) then 0 else 1
  | NotP p -> if Lts.States.mem s (Lts.prop_false lts p) then 0 else 1
  | X v -> (match lookup eta v with
    | Some props -> if Lts.States.mem s props then 0 else 1
    | None -> failwith ("prio: unbound variable " ^ v))
  | _ -> 0

let prio lts mmf eta s f =
  prio' lts eta s (unmmf mmf f)

module MuStateSet = Set.Make(struct
  type t = mu_form * Lts.state
  let compare = compare
end)

let seen_states mu seen =
  MuStateSet.fold (fun (mu', s) states ->
    if mu = mu'
    then Lts.States.add s states
    else states) seen Lts.States.empty

let check lts mmf st s f =
  let rec trav seen maxprio s f =
    Printf.printf "TRAV %s , %d , %s\n" (Mu.pretty f) s (match maxprio with
    | Some (_,_,x) -> "maxprio " ^ string_of_int x ^
                      " here " ^ string_of_int (prio lts mmf empty_env s f)
    | None -> "loop-search");
    if MuStateSet.mem (f,s) seen
    then
      (match maxprio with
      | None -> (* loop again, counting the maxprio this time. *)
        trav MuStateSet.empty (Some (s, f, 0)) s f
      | Some (s', f', prio) when s = s' && f = f' ->
```

```
      if prio mod 2 = 0 then
        seen_states f seen, true  (* good case *)
      else
        seen_states f seen, false
    | Some (_, _, _) ->
      (* not the loop we are looking for *)
      Lts.States.empty, true)
  else
    let maxprio' = match maxprio with
      | Some (s',f',maxprio) -> Some (s',f',max maxprio (prio lts mmf empty_env s f))
      | None -> None
    in
    let seen' = MuStateSet.add (f,s) seen in
    match f, Strat.assoc (f,s) st with
    | X v, Some Star ->
      (match lookup_mmf mmf v with
      | Some (MU, _, f) -> trav seen' maxprio' s f
      | Some (NU, _, f) -> trav seen' maxprio' s f
      | None -> failwith ("check.trav: unbound variable " ^ v))
    | P _, Some Star -> trav seen' maxprio' s f
    | NotP _, Some Star -> trav seen' maxprio' s f
    | Disj (f1, _), Some One -> trav seen' maxprio' s f1
    | Disj (_, f2), Some Two -> trav seen' maxprio' s f2
    | Conj (f1, f2), Some Star ->
      let s1, r1 = trav seen' maxprio' s f1 in
      if not r1
      then s1, false
      else
        let s2, r2 = trav seen' maxprio' s f2 in
        Lts.States.union s1 s2, r2
    | Box (a, f1), Some Star ->
      Lts.States.fold (fun s' (se',r1) ->
        let s'', r'' = trav seen' maxprio' s' f1 in
        if r1 == false
        then (se', false)
        else (Lts.States.union se' s'', r''))
        (Lts.succ lts a s)
        (Lts.States.empty, true)
    | Diamond (a, f1), Some Step (_, s') ->
      if Lts.StateState.mem (s, s') (Lts.Actions.find a lts.by_action)
      then trav seen' maxprio' s' f1
      else (Lts.States.empty, false)
    | Diamond _, _ ->
      Lts.States.empty, false
    | _, None ->
      seen_states f seen', false
in
trav MuStateSet.empty None s f
```

```
let gendot lts mmf st s f =
  let edge ?(take=true) f1 s1 f2 s2 =
    let p1 = prio lts mmf empty_env s1 f1 in
    let p2 = prio lts mmf empty_env s2 f2 in
    Printf.printf
      "\"%s, %d = %d\"[shape=%s]; \"%s, %d = %d\" -> \"%s, %d = %d\" [style=%s];\n"
      (String.escaped (Mu.pretty f1)) s1 p1
      (if p1 mod 2 == 0
       then "box"
       else "oval")
      (String.escaped (Mu.pretty f1)) s1 p1
      (String.escaped (Mu.pretty f2)) s2 p2
      (if take
       then "bold"
       else "dotted")
  in
  let rec iter seen s f =
    if not (MuStateSet.mem (f,s) seen) then
    let seen' = MuStateSet.add (f,s) seen in
    match f, Strat.assoc (f,s) st with
    | X v, Some Star ->
      (match lookup_mmf mmf v with
      | Some (MU, _, f1) -> edge f s f1 s; iter seen' s f1
      | Some (NU, _, f1) -> edge f s f1 s; iter seen' s f1
      | None -> failwith ("check.trav: unbound variable " ^ v))
    | P _, Some Star -> edge f s f s; iter seen' s f
    | NotP _, Some Star -> edge f s f s; iter seen' s f
    | Disj (f1, f2), Some One -> edge f s f1 s;
                                 edge ~take:false f s f2 s; iter seen' s f1
    | Disj (f1, f2), Some Two -> edge f s f2 s;
                                 edge ~take:false f s f1 s; iter seen' s f2
    | Conj (f1, f2), Some Star ->
      edge f s f1 s;
      edge f s f2 s;
      iter seen' s f1;
      iter seen' s f2
    | Box (a, f1), Some Star ->
      Lts.States.iter
        (fun s' -> edge f s f1 s'; iter seen' s' f1)
        (Lts.succ lts a s)
    | Diamond (_, f1), Some Step (_, s') -> edge f s f1 s'; iter seen' s' f1
    | _, None -> ()
  in
  print_string "strict digraph {\n";    (* strict = don't duplicate edges *)
  Lts.States.iter (fun state ->
    iter MuStateSet.empty state f) s;
  print_string "}\n"
```

B.6. Streett-automaton strategy checker

Listing B.8. streett.ml

```
open Mu
open Strat

let rec free v = function
  | X v' when v = v' -> true
  | Conj (f1, f2)
  | Disj (f1, f2) -> free v f1 || free v f2
  | Box (_, f1)
  | Diamond (_, f1) -> free v f1
  | Mu (v', f1)
  | Nu (v', f1) when v != v' -> free v f1
  | _ -> false

let rec nd (f : mu_form) =
  let rec nd' v f =
    if free v f then
      match f with
      | Conj (f1, f2)
      | Disj (f1, f2) -> max (nd' v f1) (nd' v f2)
      | Box (_, f1)
      | Diamond (_, f1) -> nd' v f1
      | Mu (_, f1)
      | Nu (_, f1) -> max (nd f) (nd' v f1)
      | _ -> 0
    else
      0
  in
  match f with
  | Mu (v, f1)
  | Nu (v, f1) -> (nd' v f1) + 1
  | _ -> 0

let prio' lts eta s = function
  | Mu (v, f1) -> 2 * nd (Mu (v, f1)) + 1
  | Nu (v, f1) -> 2 * nd (Nu (v, f1))
  | P p -> if Lts.States.mem s (Lts.prop_true lts p) then 0 else 1
  | NotP p -> if Lts.States.mem s (Lts.prop_false lts p) then 0 else 1
  | X v -> (match lookup eta v with
    | Some props -> if Lts.States.mem s props then 0 else 1
    | None -> failwith ("prio: unbound variable " ^ v))
  | _ -> 0

let prio lts mmf eta s f =
  prio' lts eta s (unmmf mmf f)
```

```
module MuStateSet = Set.Make(struct
  type t = mu_form * Lts.state
  let compare = compare
end)

let seen_states mu seen =
  MuStateSet.fold (fun (mu', s) states ->
    if mu = mu'
    then Lts.States.add s states
    else states) seen Lts.States.empty

let gendot lts mmf st s f =
  let edge ?(take=true) f1 s1 f2 s2 =
    let p1 = prio lts mmf empty_env s1 f1 in
    let p2 = prio lts mmf empty_env s2 f2 in
    Printf.printf
      "\"%s, %d = %d\"[shape=%s]; \"%s, %d = %d\" -> \"%s, %d = %d\" [style=%s];\n"
      (String.escaped (Mu.pretty f1)) s1 p1
      (if p1 mod 2 == 0
       then "box"
       else "oval")
      (String.escaped (Mu.pretty f1)) s1 p1
      (String.escaped (Mu.pretty f2)) s2 p2
      (if take
       then "bold"
       else "dotted")
  in
  let rec iter seen s f =
    if not (MuStateSet.mem (f,s) seen) then
    let seen' = MuStateSet.add (f,s) seen in
    match f, Strat.assoc (f,s) st with
    | X v, Some Star ->
      (match lookup_mmf mmf v with
      | Some (MU, _, f1) -> edge f s f1 s; iter seen' s f1
      | Some (NU, _, f1) -> edge f s f1 s; iter seen' s f1
      | None -> failwith ("check.trav: unbound variable " ^ v))
    | P _, Some Star -> edge f s f s; iter seen' s f
    | NotP _, Some Star -> edge f s f s; iter seen' s f
    | Disj (f1, f2), Some One -> edge f s f1 s;
                                 edge ~take:false f s f2 s; iter seen' s f1
    | Disj (f1, f2), Some Two -> edge f s f2 s;
                                 edge ~take:false f s f1 s; iter seen' s f2
    | Conj (f1, f2), Some Star ->
      edge f s f1 s;
      edge f s f2 s;
      iter seen' s f1;
      iter seen' s f2
    | Box (a, f1), Some Star ->
      Lts.States.iter
```

```
        (fun s' -> edge f s f1 s'; iter seen' s' f1)
        (Lts.succ lts a s)
    | Diamond (_, f1), Some Step (_, s') -> edge f s f1 s'; iter seen' s' f1
    | _, None -> ()
  in
  print_string "strict digraph {\n";    (* strict = don't duplicate edges *)
  Lts.States.iter (fun state ->
    iter MuStateSet.empty state f) s;
  print_string "}\n"


(* Emptyness of street automaton.  *)

(* from Alexandre DURET-LUTZ: Contributions a l'approche automate pour
la verification de proprietes de systemes concurrents, fig 7.6.
https://www.lrde.epita.fr/~adl/dl/adl/duret.07.phd.pdf *)

type state = Mu.mu_form * Lts.state
type label = int

type trans = state * label * state

module StateSet = Set.Make(struct type t = state let compare = compare end)
module LabelSet = Set.Make(struct type t = label let compare = compare end)

let string_of_set s =
  "{" ^ (LabelSet.fold (fun e a -> a ^ " " ^ (string_of_int e)) s "") ^ "}"

type scc = {
  state: state;
  root: int;
  la: LabelSet.t;
  acc: LabelSet.t;
  rem: StateSet.t;
  succ: trans list;
  fsucc: trans list;
}

type avoid = {
  root: int;
  acc: LabelSet.t;
}

module Hmap = Map.Make(struct type t = state let compare = compare end)

type info = {
  scc: scc list;
  h: int Hmap.t;
  max: int;
```

```
  min: int list;
  avoid: avoid list;
  delta: Lts.t * Mu.mu_map_form * Strat.partial_strat
}

let succ_states (lts,mmf,st) (f,s) =
  match Strat.assoc (f,s) st with
  | None -> [(f,s),1,(f,s)]            (* \bot loop on no strategy reply *)
  | Some move ->
    (function [] -> [(f,s),1,(f,s)]    (* \bot loop on no more move *)
            | a -> a) @@
    List.map (fun (f',s') -> ((f,s), prio lts mmf empty_env s f, (f',s'))) @@
    match f, move with
    | X v, Star ->
      (match lookup_mmf mmf v with
       | Some (MU, _, f) -> [(f,s)]
       | Some (NU, _, f) -> [(f,s)]
       | None -> failwith ("streett.succ_states: unbound variable " ^ v))
    | P _, Star -> [(f,s)]
    | NotP _, Star -> [(f,s)]
    | Disj (f1, _), One -> [(f1,s)]
    | Disj (_, f2), Two -> [(f2,s)]
    | Conj (f1, f2), Star -> [(f1,s); (f2,s)]
    | Box (a, f1), Star ->
      Lts.States.fold (fun s' l ->
                       (f1,s') :: l)
                      (Lts.succ lts a s) []
    | Diamond (a, f1), Step (_, s') ->
      if Lts.StateState.mem (s, s') (Lts.Actions.find a lts.by_action)
      then [(f1,s')]
      else []
    | _ -> []                            (* invalid move *)

let dfs_push ({scc; h; max; avoid=avoid::_; delta} as info) a q =
  let max' = max + 1 in
  let h' = Hmap.add q max' h in
  let succ, fsucc =
    succ_states delta q
    |> (fun l ->
        List.iter (fun ((a,x),y,(b,z)) ->
                   Printf.printf "%s %d ===p%d===> %s %d\n"
                                 (Mu.pretty a) x y (Mu.pretty b) z) l;
        l)
    |> List.partition (fun (_,a,_) -> LabelSet.mem a avoid.acc) in
  let scc' = { state = q;
               root = max';
               la = a;
               acc = LabelSet.empty;
               rem = StateSet.empty;
```

```
              succ;
              fsucc } :: scc
  in
  { info with scc=scc'; h=h'; max=max' }

let dfs_pop ({scc=scc1::sccs; h; min=min1::mins as min;
             avoid=avoid1::avoids as avoid} as info) =
  let n = scc1.root in
  let q = scc1.state in
  let max' = n - 1 in
  let min' = if n <= min1 then mins else min in
  let old_avoid = avoid1.acc in
  let avoid' = if n == avoid1.root then avoids else avoid in
  let even_wo_bigger_odd =
    LabelSet.fold (fun i a ->
                   if i mod 2 == 0 &&
                      not (LabelSet.exists (fun j -> j mod 2 == 1 && j > i)
                                           scc1.acc)
                   then LabelSet.add i a
                   else a) scc1.acc LabelSet.empty in
  let new_avoid = LabelSet.union old_avoid even_wo_bigger_odd
  in
  if LabelSet.equal old_avoid new_avoid then
    let h' = StateSet.fold (fun s h' -> Hmap.add s 0 h') scc1.rem h in
    { info with scc=sccs; h=h'; max=max'; min=min'; avoid=avoid' }
  else
    let h' = StateSet.fold (fun s h' -> Hmap.remove s h') scc1.rem h in
    let avoid'' = { root = n; acc = new_avoid } :: avoid' in
    dfs_push { info with scc=sccs; h=h'; max=max'; min=min'; avoid=avoid'' } scc1.la q

let merge {scc} a t =
  let rec iter (scc::sccs : scc list) a r s f =
    if t < scc.root then
      iter sccs
           (LabelSet.union a @@ LabelSet.union scc.acc scc.la)
           (StateSet.union r @@ StateSet.add scc.state scc.rem)
           (s @ scc.succ)
           (f @ scc.fsucc)
    else
      { scc with acc = LabelSet.union scc.acc a;
                 rem = StateSet.union scc.rem r;
                 succ = scc.succ @ s;
                 fsucc = scc.fsucc @ f }
      :: sccs
  in
  iter scc (LabelSet.singleton a) StateSet.empty [] []


let rec iter ({scc; h; min=min::_} as info) =
```

```
  match scc with
  | [] -> true
  | scc1::sccs ->
    match scc1.succ with
    | [] ->
      (match scc1.fsucc with
       | [] -> iter (dfs_pop info)
       | fsucc -> iter { info with
                         scc = { scc1 with succ = fsucc; fsucc = [] } :: sccs })
    | (_,a,d)::succ' ->
      let info = { info with scc = { scc1 with succ = succ' } :: sccs } in
      if not @@ Hmap.mem d h then
        iter (dfs_push info (LabelSet.singleton a) d)
      else if Hmap.find d h > min then
        (let sccs'' = merge info a (Hmap.find d h) in
         let acc = (List.hd sccs'').acc in
         if (LabelSet.max_elt acc) mod 2 == 1
         then false
         else iter { info with scc=sccs'' }
        )
      else
        iter info

let check_streett delta q0 =
  let avoid = [{root = 1; acc = LabelSet.empty}] in
  let info = {scc = []; h = Hmap.empty; max = 0; min = [0]; avoid; delta} in
  iter (dfs_push info (LabelSet.empty) q0)

let check lts mmf st s f' =
  Lts.States.empty, check_streett (lts,mmf,st) (f',s);
```

B.7. Main driver

Listing B.9. micromu.mli

```
val checker : (module Checker_intf.Checker) ref
val counterexample : bool ref
val dot : bool ref
val force : bool ref
val main : unit -> unit
```

Listing B.10. micromu.ml

```
let load_file f =
  let ic = open_in f in
  let n = in_channel_length ic in
  let s = String.create n in
  really_input ic s 0 n; close_in ic;
  s
```

```
let lts_from_string s =
  Aldebaranparse.aldebaran Aldebaranlex.main (Lexing.from_string s)

let mu_from_string s =
  Muparse.mu Mulex.main (Lexing.from_string s)

let string_of_set s =
  Lts.States.fold (fun e a -> a ^ " " ^ string_of_int e) s ""

let time s f =
  let t = Sys.time() in
  let r = f () in
  Printf.printf "### Execution time %s: %fs\n" s (Sys.time() -. t);
  r

let checker = ref (module Checker : Checker_intf.Checker)

let counterexample = ref false
let dot = ref false
let force = ref false
let argv = ref []

let args = [("-d", Arg.Set dot, "Output strategy run as .dot");
            ("-f", Arg.Set force, "Force checking of all positions");
            ("-c", Arg.Set counterexample, "Negate formula");
            ("-s", Arg.Unit (fun () ->
                     checker := (module Streett : Checker_intf.Checker)),
             "Use Streett automaton for checking")]
let usage = "Usage: micromu [-cdfs] ltsfile mufile"

let main () =
  Arg.parse args (fun arg -> argv := !argv @ [arg]) usage;

  let (module Checker : Checker_intf.Checker) = !checker in

  match !argv with
  | [ltsfile; mufile] ->
    let lts = lts_from_string (load_file ltsfile)
    and mu = mu_from_string (load_file mufile) in

    let mu =
      if !counterexample
      then Mu.negate mu
      else mu in

    let mu', mmf = Mu.study mu in

    Printf.printf "%s\n\n%!" (Mu.pretty_mmf mmf);
```

```
let c = time "Mu.sem" (fun () -> Mu.sem lts mmf mu' Mu.empty_env) in

Printf.printf "result: %s\n\n%!" (string_of_set c);

let st = time "Strat.sem" (fun () -> Strat.sem lts mmf mu' Mu.empty_env) in
print_string "\n------\n";
print_string (Strat.pretty ~sep:" |\n" st);
print_string "\n------\n";

if !dot then
  time "gendot" (fun () -> Checker.gendot lts mmf st c mu');

print_string "\n------\n";

let verified = ref true in

let seen_good = ref Lts.States.empty in
let seen_bad = ref Lts.States.empty in

time "verify" (fun () ->
Lts.States.iter (fun s ->
  Printf.printf "verifying for good state %d:\n%!" s;

  if Lts.States.mem s !seen_good
  then
    Printf.printf "state %d: known good\n%!" s
  else
  let seen,t = Checker.check lts mmf st s mu' in
  if t
  then
    (Printf.printf "state %d: true\n%!" s;
     if not !force then
     seen_good := Lts.States.union !seen_good seen)
  else
    (Printf.printf "state %d: FALSE\n%!" s;
     verified := false)
) c;

Lts.States.iter (fun s ->
  Printf.printf "verifying for bad state %d:\n%!" s;

  if Lts.States.mem s !seen_bad
  then
    Printf.printf "state %d: known bad\n%!" s
  else
  let seen,t = Checker.check lts mmf st s mu' in
  if not t
  then
    (Printf.printf "state %d: false\n%!" s;
```

```
         if not !force then
         seen_bad := Lts.States.union !seen_bad seen)
       else
         (Printf.printf "state %d: TRUE\n%!" s;
         verified := false)
    ) (Lts.States.diff (Lts.all_states lts) c));

    Printf.printf "result: %s\n%!" (string_of_set c);
    Printf.printf "verification %s\n%!" (if !verified then "passed" else "FAILED")
  | _ ->
    Arg.usage args usage

let _ = main ()
```

Bibliography

[1] Henk Barendregt and Freek Wiedijk. "The challenge of computer mathematics." In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 363.1835 (Oct. 15, 2005), pp. 2351–2375. ISSN: 1471-2962.

[2] Armin Biere. "μcke – Efficient μ-Calculus Model Checking." In: *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*. Ed. by Orna Grumberg. Vol. 1254. Lecture Notes in Computer Science. Springer, 1997, pp. 468–471. ISBN: 3-540-63166-6. DOI: 10.1007/3-540-63166-6_50. URL: http://dx.doi.org/10.1007/3-540-63166-6_50.

[3] Armin Biere. "Efficient Model Checking of the Mu-Calculus with Binary Decision Diagrams." German. PhD Thesis. Uni Karlsruhe, Jan. 1997.

[4] Nils Buhrke, Helmut Lescow, and Jens Vöge. "Strategy construction in infinite games with Streett and Rabin chain winning conditions." In: *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 1055. Lecture Notes in Computer Science. Springer, 1999, pp. 207–225. ISBN: 3-540-61042-1.

[5] CADP, ed. *aut, AUT – simple file format for labelled transition systems*. 2014. URL: http://www.inrialpes.fr/vasy/cadp/man/aut.html.

[6] P. Cousot and R. Cousot. "Constructive Versions of Tarski's Fixed Point Theorems." In: *Pacific Journal of Mathematics* 81.1 (1979), pp. 43–57.

[7] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] Alexandre Duret-Lutz. "Contributions à l'approche automate pour la vérification de propriétés de systèmes concurrents." PhD Thesis. Université Pierre et Marie Curie (Paris 6), July 2007. URL: https://www.lrde.epita.fr/~adl/th.html.

[9] Alexandre Duret-Lutz, Denis Poitrenaud, and Jean-Michel Couvreur. "On-the-fly Emptiness Check of Transition-based Streett Automata." In: *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*. Ed. by Zhiming Liu and Anders P. Ravn. Vol. 5799. Lecture Notes in Computer Science. Springer, 2009, pp. 213–227.

[10] John Ellson et al. "Graphviz – Open Source Graph Drawing Tools." In: *Graph Drawing* (2001), pp. 483–484.

[11] E. Allen Emerson. "Model Checking and the Mu-calculus." In: *Descriptive Complexity and Finite Models, Proceedings of a DIMACS Workshop, January 14-17, 1996, Princeton University*. Ed. by Neil Immerman and Phokion G. Kolaitis. Vol. 31. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996, pp. 185–214. ISBN: 0-8218-0517-7.

[12] E. Allen Emerson and Charanjit S. Jutla. "The Complexity of Tree Automata and Logics of Programs." In: *SIAM J. Comput.* 29.1 (Sept. 1999), pp. 132–158. ISSN: 0097-5397. DOI: 10.1137/S0097539793304741. URL: http://dx.doi.org/10.1137/S0097539793304741.

[13] E. Allen Emerson and Charanjit S. Jutla. "Tree Automata, Mu-Calculus and Determinacy (Extended Abstract)." In: *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. IEEE Computer Society, 1991, pp. 368–377. DOI: 10.1109/SFCS.1991.185392. URL: http://doi.ieeecomputersociety.org/10.1109/SFCS.1991.185392.

[14] E. Allen Emerson and Chin-Laung Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus (Extended Abstract)." In: *LICS'86*. 1986, pp. 267–278.

[15] Javier Esparza et al. "A Fully Verified Executable LTL Model Checker." In: *Computer Aided Verification (CAV 2013)*. Ed. by N. Sharygina and H. Veith. Vol. 8044. 2013, pp. 463–478.

[16] Jean-Claude Fernandez. "ALDEBARAN : un système de vérification par réduction de processus communicants." PhD Thesis. Université Joseph-Fourier – Grenoble I, 1988. eprint: http://tel.archives-ouvertes.fr/tel-00326157.

[17] Oliver Friedmann. "An Exponential Lower Bound for the Parity Game Strategy Improvement Algorithm as We Know it." In: *LICS*. IEEE Computer Society, 2009, pp. 145–156. ISBN: 978-0-7695-3746-7.

[18] Martin Hofmann and Martin Lange. *Automatentheorie und Logik*. 1st ed. Springer-Verlag, 2011. ISBN: 978-3-642-18089-7.

[19] Martin Hofmann and Harald Rueß. "Certification for mu-calculus with winning strategies." In: *ArXiv e-prints* (Jan. 2014). arXiv: 1401.1693 [cs.LO].

[20] Giacomo Lenzi. "A hierarchy theorem for the μ-calculus." In: *Automata, Languages and Programming*. Ed. by Friedhelm Meyer and Burkhard Monien. Vol. 1099. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 87–97. ISBN: 978-3-540-61440-1. DOI: 10.1007/3-540-61440-0_119. URL: http://dx.doi.org/10.1007/3-540-61440-0_119.

[21] Xavier Leroy et al. *The OCaml system release 4.01: Documentation and user's manual*. Anglais. Sept. 2013. URL: http://hal.inria.fr/hal-00930213.

[22] Tiziana Margaria and Bernhard Steffen, eds. *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*. Vol. 1055. Lecture Notes in Computer Science. Springer, 1996. ISBN: 3-540-61042-1.

[23] Donald A. Martin. "Borel determinacy." In: *Annals of Mathematics* 102 (1975), pp. 363–371.

[24] Kenneth L. McMillan. "Symbolic Model Checking." PhD Thesis. Carnegie Mellon University, 1993. URL: http://www.kenmcmil.com/pubs/thesis.pdf.

[25] Kedar S. Namjoshi. "Certifying Model Checkers." In: *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*. Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Vol. 2102. Lecture Notes in Computer Science. Springer, 2001, pp. 2–13. ISBN: 3-540-42345-1. DOI:

10.1007/3-540-44585-4_2. URL: http://dx.doi.org/10.1007/3-540-44585-4_2.

[26] Tobias Nipkow and Gerwin Klein. *Concrete Semantics*. 1st ed. Mar. 21, 2014. URL: http://www21.in.tum.de/~nipkow/Concrete-Semantics/ (visited on Aug. 22, 2014).

[27] Etienne Renault et al. "Three SCC-based Emptiness Checks for Generalized Büchi Automata." In: *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'13)*. Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. Lecture Notes in Computer Science. Springer, Dec. 2013, pp. 668–682.

[28] Hassen Saïdi and Natarajan Shankar. "Abstract and Model Check while You Prove." In: *Computer Aided Verification*. Ed. by Nicolas Halbwachs and Doron Peled. Vol. 1633. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 443–454. ISBN: 978-3-540-66202-0. DOI: 10.1007/3-540-48683-6_38. URL: http://dx.doi.org/10.1007/3-540-48683-6_38.

[29] N. Shankar and M. Sorea. *Counterexample-Driven Model Checking*. Revisited version. Technical Report. SRI International, 2003. URL: http://www.csl.sri.com/users/sorea/reports/wmc.ps.gz (visited on June 18, 2014).

[30] Colin Stirling. "Games and Modal Mu-Calculus." In: *TACAS*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 1055. Lecture Notes in Computer Science. Springer, 1996, pp. 298–312. ISBN: 3-540-61042-1.

[31] Colin Stirling and David Walker. "Local Model Checking in the Modal Mu-Calculus." In: *TAPSOFT, Vol.1*. Ed. by Josep Díaz and Fernando Orejas. Vol. 351. Lecture Notes in Computer Science. Springer, 1989, pp. 369–383. ISBN: 3-540-50939-9.

[32] Robert S. Streett and E. Allen Emerson. "An Automata Theoretic Decision Procedure for the Propositional Mu-Calculus." In: *Information and Computation* 81.3 (1989), pp. 249–264.

[33] Robert Streett and E. Emerson. "The propositional mu-calculus is elementary." In: *Automata, Languages and Programming*. Ed. by Jan Paredaens. Vol. 172. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1984, pp. 465–472. ISBN: 978-3-540-13345-2.

[34] perf users, ed. *perf: Linux profiling with performance counters*. 2014. URL: https://perf.wiki.kernel.org/ (visited on Sept. 4, 2014).

[35] Kumar Neeraj Verma. *Reflecting Symbolic Model Checking in Coq*. Report on DEA internship. GIA Dyade and INRIA, 2000.

[36] Pierre Wolper. "Temporal Logic Can Be More Expressive." In: *Information and Control* 56.1/2 (1983), pp. 72–99.

Hiermit versichere ich, dass die zum heutigen Tag an der Fakultät für Mathematik, Informatik und Statistik eingereichte Masterarbeit zum Thema "Computation of winning strategies for μ-calculus by fixpoint iteration" selbstständig verfasst wurde und ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Datum, Unterschrift: _____