Implementing an LTL-to-Büchi Translator in Erlang

a ProTest Experience Report

Hans Svensson

Department of Applied IT, Chalmers University of Technology, Gothenburg, Sweden

hanssv@chalmers.se

Abstract

In order to provide a nice user experience in McErlang, a model checker for Erlang programs, we needed an LTL-to-Büchi translator. This paper reports on our experiences implementing a translator in Erlang using well known algorithms described in literature. We followed a *property driven development* schema, where QuickCheck properties were formulated before writing the implementation. We successfully implement an LTL-to-Büchi translator, where the end result performs on par (or better) than two well known reference implementations.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing tools

General Terms Algorithms, Verification

Keywords LTL-to-Büchi translator, QuickCheck, property driven development

1. Introduction

Correctness of concurrent or distributed software is a well known, and immensely complicated problem. It is also a fact that during the last couple of years, the problem has become more important due to the introduction of multi-core processors. This has led to an increased interest in the problem, and a variety of good work has made the problem a lot more tractable.

Model checking is one of the most successful and mostly used techniques to prove correctness of a (hardware or software) system. The standard model checking problem consists of a model system expressed as a *finite state machine (FSM)*, and a specification given by a *temporal logic formula*. Model checking for temporal logic formulas was pioneered by Clarke et al. (1986) and as well by Queille and Sifakis (1982). The main obstacle in model checking is the famous *state space explosion* problem, due to a combinatorial blow-up of the state space (the size of the FSM). This combinatorial blow up is very problematic when dealing with concurrent and fault tolerant systems (meaning that most Erlang software falls into this category). However, there are some different techniques that try to deal with the problem such as:

- Symbolic model checking, where the state space is instead represented symbolically in terms of logic formulas.
- Partial order reduction techniques, where equivalent paths through the FSM are grouped together and reduced to a single path.
- Abstraction techniques, where the system is simplified by an abstraction function, thus reducing the size of the FSM.

One tool that aims to simplify the verification of correctness for distributed Erlang programs is McErlang, a model checker for Erlang. McErlang was first presented in (Fredlund and Svensson 2007), and has been used in a couple of successful projects (Fredlund and Sánchez Penas 2007; Earle et al. 2008). Since 2008 McErlang is an important part of the ProTest project and the release of the first public open source version in March 2009 is a result of this.

McErlang (Fredlund and Svensson 2007) is a fairly standard explicit state model checker, although through plugins more advanced concepts such as state abstraction can easily be used. The interesting part of McErlang is that it is a model checker for Erlang written in Erlang. As a result of this, side-effect free parts of a system can be evaluated as is, without tedious translation. This means that one can focus on the complicated parts such as distribution and fault tolerance, and the result is that McErlang supports a large subset of Erlang. McErlang implements the full distributed Erlang semantics (Svensson and Fredlund 2007) and all the important OTP components (`gen_server, gen_fsm, ...`).

The first (non-public) versions of McErlang encoded correctness properties (specifications) as simple automata programmed directly in Erlang. Having very little support for higher level constructs, this meant that writing properties was both tedious and error prone. For more complex properties (such as *fairness* properties) McErlang also supported Büchi automata, but they still needed be hand written.

To make McErlang more accessible before its first public release it was decided to simplify the specifications by adding the possibility of using LTL properties. This is fairly straightforward, since LTL expressions can be automatically translated into Büchi automata (Wolper et al. 1983). However, for model checking to be efficient it is important to produce as small an automaton as possible, thus a good translator was needed. The obvious solution was to use an existing implementation. However, no Erlang implementation could be found and although it is a simple task to wrap an existing Java implementation (such as Giannakopoulou and Lerda (2002)) it did not appeal to us aesthetically having advocated the all-in-Erlang aspect of McErlang. From a distribution point of view it is also simpler to have a native implementation, we avoid the licensing aspect as well as the problem of missing external components, while being in control of new releases and bug fixes. In the end, this made us decide to implement an LTL formula to Büchi automata translator ourselves.

This paper describes the implementation of an LTL to Büchi automata translator in Erlang. The implementation includes some state-of-the-art technologies and it performs quite well as seen in Sect. 5. The paper focuses quite a lot on the verification of the correctness of the implementation, and could be seen as an experience report on *property driven development* and *property based testing*. We used Erlang QuickCheck (Hughes 2007) to formulate properties and run test cases randomly generated from the properties. Property driven development is the QuickCheck dual of *test driven development* (TDD) (Beck 2003), where the test cases drive the implementation forward. In practice it consists of three stages; (1) tests are written that do **not** pass for the implementation, (2) the implementation is extended so the tests pass, and (3) the code is refactored to produce the end result. The development process is iterative, and it worked well for our implementation. Using QuickCheck makes phase (2) even more productive. Whenever a test fail, QuickCheck provide a minimal counter example, and in many cases the counter example shorten the time to diagnose an error. In our work it was very useful to get small LTL formulas for which the implementation produces an incorrect automaton.

Paper organization – The paper is organized as follows; in Sect. 2 we introduce the theoretical background as well as the problem of testing an LTL to Büchi automata translation. Sect. 3 describes the properties that were defined during the implementation, and how various model checking concepts are mapped to QuickCheck. The implementation is described in Sect. 4, the implementation is evaluated and compared to other implementations in Sect. 5 and the paper is summarized in Sect. 6.

2. Theoretical background

In this section, we provide some theoretical background to the concepts discussed in later sections.

2.1 Linear temporal logic – LTL

To write specifications for reactive systems, we need to be able to precisely describe how the system behaves for (possibly) infinite executions. Temporal logic (Pnueli 1977) has become a de facto standard formalism for this kind of specifications, and there are quite a few flavors of temporal logic, here we focus on linear temporal logic. LTL is an extension of *propositional logic*. If AP is a (non-empty and finite) set of atomic propositions, then the LTL formulas are:

- All $p \in AP$ are LTL formulas.
- If φ and ψ are LTL formulas, then $\neg\varphi,\ (\varphi \wedge \psi)$, $\mathbf{X}\,\varphi,\ (\varphi\,\mathbf{U}\,\psi)$ are also LTL formulas.

The standard semantics for an LTL formula φ is defined in terms of an infinite sequence (ξ) over 2^{AP}. Let ξ^i be the infinite subsequence of ξ that begins at the ith position of ξ ($i \geq 0$). Now the relation ($\models$) is defined as:

- For $p \in AP$, $\xi \models p$ iff $p \in \xi_0$. (ξ_0 is the first element of ξ).
- $\xi \models \neg\varphi$ iff $\neg(\xi \models \varphi)$, usually written $\xi \not\models \varphi$.
- $\xi \models \mathbf{X}\,\varphi$ iff $\xi^1 \models \varphi$.
- $\xi \models (\varphi\,\mathbf{U}\,\psi)$ iff $\exists\, i \geq 0.\ \xi^i \models \psi$ and $\forall\, 0 \leq j < i.\ \xi^j \models \varphi$.

If $\xi \models \varphi$, then ξ is called a *model* of φ. The set of all models $\{\xi \in (2^{AP})^\omega \mid \xi \models \varphi\}$ is called the *language* of φ and is denoted by $\mathcal{L}_\varphi$.

2.2 Büchi automaton

Büchi automata were introduced by Büchi (1960). A Büchi automaton accepts infinite input sequences if and only if there exists a path that visits an accepting state infinitely often. A Büchi automaton is a tuple $BA = \langle \Sigma, Q, \Delta, Q^0, F\rangle$ where:

- Σ is the *alphabet*,
- Q is the finite set of *states*,
- $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*,
- $Q^0 \subseteq Q$ is the set of *initial states*, and
- $F \subseteq Q$ is the set of *accepting states*.

An *execution* of BA for an infinite sequence $\xi = \langle x_0, x_1, \ldots\rangle \in \Sigma^\omega$ is the infinite sequence of states $\sigma = \langle q_0, q_1, \ldots\rangle \in Q^\omega$ where $q_o \in Q^0$, and for all $i \geq 0,\ (q_i, p_i, q_{i+1}) \in \Delta$. Let $\mathit{inf}(\sigma) \subseteq Q$ denote the states that occur infinitely often in an execution σ, then σ is an *accepting* execution if and only if $(\mathit{inf}(\sigma) \cap F) \neq \emptyset$.

We say that the automaton *accepts* $\xi \in \Sigma^\omega$ exactly when there exists an accepting execution of BA on ξ. If there is no accepting execution, then BA is said to *reject* ξ. The *accepted* language of an automaton BA is the set of sequences ξ accepted by the automaton BA, it is denoted by $\mathcal{L}_{BA}$.

There is a close connection between languages induced by an LTL formula and the accepted language of a Büchi automaton, in fact any LTL formula can be represented as a Büchi automaton. This connection was explicitly shown by Wolper et al. (1983) and this property was used to develop the automata-theoretic approach to LTL model checking (Vardi and Wolper 1986).

2.3 LTL model checking

The LTL model checking problem answers the question of whether a specification (in the form of an LTL formula) is satisfied in a finite model of a system. The system is often (translated into) a state-transition graph where each state is augmented with exactly the atomic propositions that hold in the state. Such a transition system is known as a *Kripke structure*; more formally defined as a tuple $M = \langle S, \rho, \pi\rangle$, where:

- S the finite set of *states*,
- $\rho \subseteq S \times S$ is the *transition relation*,
- $\pi\ :\ S \to 2^{AP}$ is the *labeling function*, which means that $\pi(s)$ is the set of propositions that hold in a state $s \in S$.

The model checking problem can thus be formulated as: "For a Kripke structure $M = \langle S, \rho, \pi\rangle$, a state s and an LTL formula φ, is it true for all infinite paths $x = \langle s_0, s_1, \ldots\rangle \in S^\omega$ (with $s_0 = s$) that $\xi = \langle \pi(s_0), \pi(s_1), \ldots\rangle \models \varphi$". For practical reasons this is often reformulated as looking for *any* x such that $\xi = \langle \pi(s_0), \pi(s_1), \ldots\rangle \models \neg\varphi$, i.e. $M \models \varphi$ holds if no such x can be found.

By treating the set of paths starting in a specific state s as a language $\mathcal{L}_{M,s}$ the problem can compactly be expressed as checking whether $\mathcal{L}_{M,s} \cap \mathcal{L}_{\neg\varphi} = \emptyset$. This means that the LTL model checking problem can be solved by constructing $BA_{\neg\varphi}$ and $BA_{M,s}$ from the LTL formula φ and the Kripke structure M respectively, and then checking the automaton $BA_{\neg\varphi} \otimes BA_{M,s}$ for *emptiness* ($\otimes$ is the synchronous composition of two automata, and the resulting automata recognizes exactly the infinite sequences recognized by both automata). This check can be done by an algorithm that checks whether the constructed automata can reach an accepting cycle from any of its initial states (Clarke et al. 2000). The time complexity is linear in the size of the new automata.

2.4 Testing an LTL to Büchi automata translation

The theoretical aspect of how to test an LTL formula translation into Büchi automata is thoroughly covered by Tauriainen and Heljanko (2002). Here we briefly describe the most important aspects from Sect. 3 of (Tauriainen and Heljanko 2002). The methods described do not aim to *prove* the correctness of the implementation, but instead perform tests to detect inconsistencies in the implementations. Most of the ideas test the results given by an LTL-to-Büchi translator with the results from using other implementations. One particular situation could be to test different optimizations against

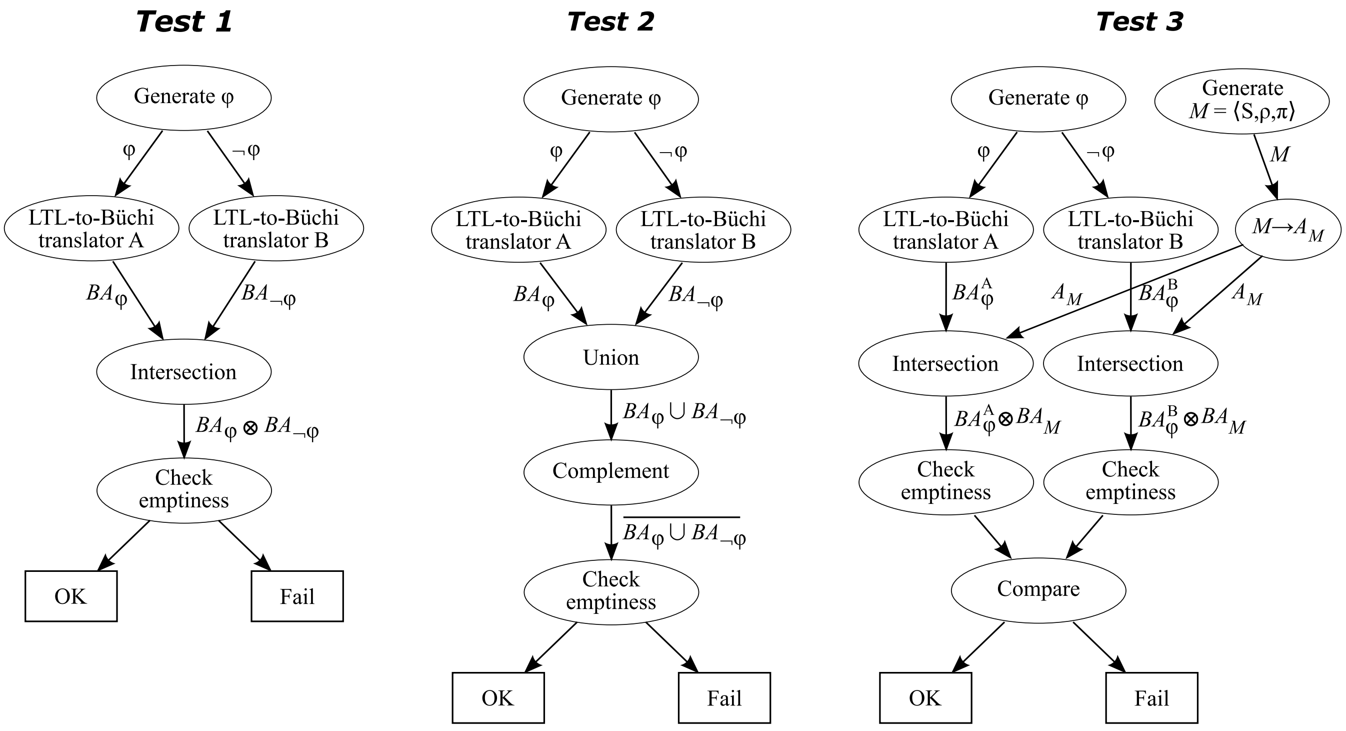

Figure 1. Testing an LTL-to-Büchi translation

(a stable) reference implementation. This is something that we have used frequently. We also used two reference implementations WRING by Somenzi and Bloem (2000) and LTL2BUCHI by Giannakopoulou and Lerda (2002).

Interestingly, we found that WRING fails for some particular input. Therefore, in order to be able to use WRING as a reference implementation, we implemented a pre-condition check that guarantee that the generated formula is one that does not crash WRING[1].

2.4.1 Language property tests

The first two tests use properties of the languages $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$, in particular we use the fact that for each $\xi \in (2^{AP})^\omega$, $\xi \in \mathcal{L}_\varphi$ if and only if $\xi \notin \mathcal{L}_{\neg\varphi}$. Thus we can conclude that:

- The languages $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$ are disjoint, which means that $\mathcal{L}_\varphi \cap \mathcal{L}_{\neg\varphi} = \emptyset$
- Each $\xi \in (2^{AP})^\omega$ belongs to either $\mathcal{L}_\varphi$ or $\mathcal{L}_{\neg\varphi}$, and therefore $\mathcal{L}_\varphi \cup \mathcal{L}_{\neg\varphi} = (2^{AP})^\omega$

The first test is easy to realize in practice, we just need a way to generate random LTL formulas. For each formula φ we produce BA_φ and $BA_{\neg\varphi}$ using different translators and check that the intersection ($BA_\varphi \otimes BA_{\neg\varphi}$) is empty. If the intersection is non-empty we know that one (or both) translator did not produce the correct behavior.[2]

The second test is harder to utilize directly. The sequences involved are infinite (and infinitely many), thus we can not generate them all, and to algorithmically check for universality is PSPACE-complete (as shown by Vardi (1996)). The test can (as shown in Fig. 1) be reduced to emptiness checking, by *complementing* $BA_\varphi \cup BA_{\neg\varphi}$. However, the complementation automata of a (non-deterministic) Büchi automata may have an exponential blow-up in size. (Currently the upper bound is $(0.97n)^n$ states and the lower bound is $(0.76n)^n$ states, see Vardi (2007).) Therefore, instead of using complementation, we simplify this test into something which can easily be used in QuickCheck, as shown in Sect. 3. Since we are only testing for defects, a simplified test is of interest.

2.4.2 Model checking result tests

For a given model (Kripke structure) M and a given LTL formula φ the value of $M \models \varphi$ is well defined by the semantics of LTL. This means that we can use the result of the model checking problem to test the LTL-to-Büchi translations against each other. Using two different LTL-to-Büchi translators we can produce BA^1_φ and BA^2_φ. Then we translate the model M into another automaton BA_M. The test then compare the results of the emptiness checks for $BA^1_\varphi \otimes BA_M$ and $BA^2_\varphi \otimes BA_M$. The three tests are graphically described in Fig. 1.

3. Properties

In this section the QuickCheck specification for the testing of the LTL-to-Büchi translation is introduced. Most of the section is devoted to explaining how the tests from Sect. 2.4 are implemented in terms of QuickCheck.

3.1 LTL formula generator

Random LTL formulas are generated by the QuickCheck *generator* presented in Fig. 2. The generator is recursively defined. In the base case we pick an element from the alphabet, and in the recursive case we pick one of the seven different ways to produce an LTL formula from sub-formulas. Note, here we use the more common operators *always* ($\Box\varphi$) and *eventually* ($\Diamond\varphi$). These operators are defined in terms of the until-operator ($\Box\varphi \equiv$ true $\mathbf{U}$ φ and $\Diamond\varphi \equiv \neg$(true$\mathbf{U}\neg\varphi$)). In the generator, the `LETSHRINK`-macro is used to be

[1] In the documentation, it is claimed that the reason for the crashes is an incompatibility between WRING and more recent versions of Perl

[2] We should also note that this test itself is not enough since a translator that produces the empty automaton for all LTL formulas would pass the test.

able to *shrink* generated formulas when a counter example is found. (Refer to Hughes (2007) for a thorough introduction to QuickCheck and its shrinking facility, and Arts et al. (2008) for an explanation of LETSHRINK.) The divisors (2 and 4) are rather arbitrarily chosen since they result in reasonably complex LTL formulas. An alphabet is a sorted non-empty list of propositional variables (chosen from a-f). The SIZED-macro is used to grab the implicit *size*-parameter in QuickCheck and pass it to the formula generator.

```
lprop() ->
    elements([{lprop,X}
              || X <- [a,b,c,d,e,f]]).

alpha() ->
  ?LET(Lst, [lprop() | list(lprop())],
            lists:usort(Lst)).

ltl_formula() ->
  ?SIZED(Size,
        ?LET(Alpha, alpha(),
             ltl_formula(Size, Alpha))).

ltl_formula(0, Alpha) ->
  elements(Alpha);
ltl_formula(Size,Alpha) ->
  Small = ltl_formula(Size div 2,Alpha),
  Smaller = ltl_formula(Size div 4,Alpha),
  oneof(
   [ltl_formula(0,Alpha),
    ?LETSHRINK([Phi,Psi],[Smaller,Smaller],
               {land,Phi,Psi}),
    ?LETSHRINK([Phi,Psi],[Smaller,Smaller],
               {lor,Phi,Psi}),
    ?LETSHRINK([Phi],[Small],{next,Phi}),
    ?LETSHRINK([Phi],[Small],{eventually,Phi}),
    ?LETSHRINK([Phi],[Small],{always,Phi}),
    ?LETSHRINK([Phi],[Small],{lnot,Phi})
   ]).
```

Figure 2. LTL formulas generator

3.2 Test 1

The first test is implemented in the property prop_test1, listed in Fig. 3. The property is parametrized by the LTL-to-Büchi translation functions to use B1 and B2. The rest of the property is straightforward, we create the two Büchi automata, calculate the intersection and check for emptiness.

3.3 Test 2

The second test is implemented in the property prop_test2, listed in Fig. 4. This property is also parametrized by the LTL-to-Büchi translation functions to use B1 and B2. Since it is not computationally feasible to check exactly what is presented we have simplified the test a bit. We have introduced a simple deterministic Kripke structure, which we denote a *witness*. A witness is in effect an infinite sequence of sets of labels produced by a *prefix* and a (non-empty) *loop*. The generator for witness is also presented in Fig. 4. The test checks that exactly one of the two automata BA_φ and $BA_{\neg\varphi}$ accepts the witness.

3.4 Test 3

The implemented property for the third test is very similar to the second test. Instead of generating BA_φ and $BA_{\neg\varphi}$, we generate BA^1_φ and BA^2_φ and check that their model checking results agree for different witnesses W. The third test property is listed in Fig. 5.

```
prop_test1(B1, B2) ->
    ?FORALL(Phi, (ltl_formula()),
      begin
        Bu1 = B1(Phi),
        Bu2 = B2(negate(Phi)),
        Bu1Bu2 = buchi:intersection(Bu1, Bu2),
        buchi:is_empty(Bu1Bu2)
      end).
```

Figure 3. Test 1 – QuickCheck property

```
witness(Alpha) ->
    #witness{alpha  = Alpha,
             prefix = list(lbl_set(Alpha)),
             loop   = [lbl_set(Alpha)
                       | list(lbl_set(Alpha))]}.

prop_test2(B1, B2) ->
  ?FORALL(Alpha, alpha(),
    ?FORALL(
      {Phi, W},
      {ltl_formula(Alpha), witness(Alpha)},
    begin
      Bu1 = B1(Phi),
      Bu2 = B2(negate(Phi)),
          is_witness(W, Bu1) =/=
            is_witness(W, Bu2)
    end)).
```

Figure 4. Test 2 – QuickCheck property

```
prop_test3(B1, B2) ->
  ?FORALL(Alpha, alpha(),
    ?FORALL(
       Phi, ltl_formula(Alpha),
       begin
         Bu1 = B1(Phi),
         Bu2 = B2(Phi),
         ?FORALL(W, witness(Alpha),
           is_witness(W, Bu1) ==
             is_witness(W, Bu2))
       end)).
```

Figure 5. Test 3 – QuickCheck property

3.5 Additional tests

The properties listed above cover all of the high level properties for the LTL-to-Büchi translation. In order to help during development we used many more, more fine grained, properties that only covered small parts of the translation. We also wrote some tests to check the intersection function used in the tests listed above. One example of a property for the intersection function is listed in Fig. 6. This test checks that for all Büchi automata B1 and B2, if the intersection B1B2 accepts a witness, then also B1 as well as B2 should accept the same witness.

When using this test, we ran into one of the standard problems with QuickCheck; namely that randomly generated data is not good enough. For a random (non-empty) Büchi automaton B, only about 2 out of 100 random witnesses are accepted by B. For the more complex B1B2 in the intersection property, the numbers are even worse: only in about 1 out of 250 generated combinations of B1B2 and W, was W accepted by B1B2.

The general solution to this problem is to use a better (more specific) data generator. In our case the problem is two-fold; (1) the intersection of two random non-empty Büchi automata is often

```
prop_intersection() ->
  ?FORALL(A, alpha(),
    ?FORALL({B1, B2}, {buchi(A), buchi(A)},
      begin
        B1B2 = buchi:intersection(B1, B2),
        ?FORALL(W, witness(A),
          (not is_witness(W, B1B2)) orelse
           (is_witness(W, B1) andalso
             is_witness(W, B2)))
      end)).
```

Figure 6. Intersection – QuickCheck property

```
prop_intersection() ->
  ?FORALL(A, alpha(),
    ?FORALL({B1, B2}, {buchi(A), buchi(A)},
      begin
        B1B2 = buchi:intersection(B1, B2),
        ?IMPLIES(
          not buchi:is_empty(B1B2),
          ?FORALL(W, witness_for_buchi(B1B2),
             (is_witness(W, B1) andalso
               is_witness(W, B2))))
      end)).
```

Figure 7. Intersection – optimized QuickCheck property

an automaton that do not accept any witnesses and (2) a random witness is not very likely to be accepted by a non-empty Büchi automaton. Therefore, we added a filter (using the `IMPLIES`-macro in QuickCheck) for empty automata and implemented a new generator `witness_for_buchi(B)`, that given a non-empty Büchi automaton `B` produces a witness that is accepted by `B`.

The trick is to search for an accepting cycle (picking one randomly if there are several) in the automaton, and use this together with a suitable prefix leading up to the cycle as a witness. For a non-empty automaton this generator, by construction, *always* gives a witness accepted by the automaton. This might not always be desired, but for the intersection property we are interested in the case when `W` is witness for `B1B2` and `witness_for_buchi` is the perfect generator to use. In Fig. 7 the optimized property is presented.

4. Implementation

After having all the QuickCheck properties available, as well as some already tested Büchi automaton manipulation functions (`intersection, is_empty`, etc), we were ready to implement the LTL-to-Büchi translator. Most of what is implemented has already been described elsewhere, we have looked for inspiration in several different places and combined many bits and pieces. Most well performing LTL-to-Büchi translator consist of the following three parts:

1. A rewrite engine, which aims to simplify the LTL formula. It normally uses a fixed set of (heuristically chosen) rewrite rules. One example is well documented by Somenzi and Bloem (2000).
2. Core translation algorithm – Construction of the Büchi automaton from the re-written LTL formula. There are two main algorithms for this phase: the tableau-based algorithms (for example described by (Gerth et al. 1996)), and algorithms based on *alternating automata* as introduced by Gastin and Oddoux (2001).
3. If needed, a translation of the result in phase 2, into a standard Büchi automaton. (Many translations works with intermediate automata formats, such as *generalized Büchi automata*, *alternating automata*, *transition-based Büchi automata*, etc.) Thereafter, reductions and optimizations, such as *simulation reductions* (see for example Etessami and Holzmann (2000)) and removal of *non-reachable* and *non-accepting* states, are applied to the Büchi automaton.

4.1 Rewrite LTL formula

Implementation of a rewrite functionality is fairly straightforward. You have to choose which rewrite rules to use and implement the application of a rewrite rule. We chose to use the rewrite rules described by Somenzi and Bloem (2000), these rules are the result of some thorough experiments and have also been used in (Giannakopoulou and Lerda 2002). The rules aim to simplify the LTL formula in a way that is (according to the heuristics) favorable in terms of the size of the resulting Büchi automaton. For example the LTL formula $(\mathbf{X}\,\varphi)\;\mathbf{U}\;(\mathbf{X}\,\psi)$ is re-written into $\mathbf{X}\;(\varphi\;\mathbf{U}\;\psi)$. Testing the rewrite facility using the properties described in Sect. 3 (by using one translator function with rewriting and one function without) quickly removed some (rather silly) implementation errors.

4.2 Core translation algorithm

As indicated above, there are two main alternatives for this phase: the tableau-based algorithms and algorithms based on alternating automata. We have chosen to use a tableau-based algorithm. We implemented an algorithm in the style described by Giannakopoulou and Lerda (2002). The main reason for choosing a tableau-based algorithm was non-technical, we simply were more familiar with this style of algorithm. We believe that a core translation based on alternating automata would have performed at about the same level.

The translation algorithm generates *transition-based generalized Büchi automata*, which carry labels on transitions instead of the normal state-based automata. The benefit of using a transition-based automata for the core translation is that more states can possibly be merged during translation. The result is a potentially smaller resulting automaton. The algorithm is tableau-based and works by expanding $\varphi\;\mathbf{U}\;\psi$ into $\psi \vee (\varphi \wedge \mathbf{X}\;(\varphi\;\mathbf{U}\;\psi))$. Step by step the automaton is build up, while keeping track of equivalent states as well as acceptance conditions.

By using the QuickCheck properties and the (small) counter examples it produced it was a rather painless process to get the core translation algorithm correctly (at least to the level of passing a very large number of tests) implemented.

4.3 Degeneralization

Since the core translation algorithm produces a (transition-based) *generalized* Büchi automaton, while McErlang only supports non-generalized automata, we needed to implement degeneralization. Again we follow what is described by Giannakopoulou and Lerda (2002), with some additions. A generalized automaton has (possibly) more than one set of accepting states, and an infinite sequence is accepting only if it passes through *all* accepting sets infinitely often. To convert a generalized automaton into a non-generalized automaton we need to translate the concept of visiting all states into a single accepting state set. The approach taken is to use a second specialized automaton called a *degeneralizer*. The degeneralizer has a size (and shape) that corresponds directly to the number of accepting sets in the generalized automaton. The degeneralizer 'counts' the visits of accepting sets, and all accepting cycles in the degeneralizer visit all accepting sets. To produce the final degeneralized automaton the *synchronous product* between the generalized automaton and the degeneralizer is computed. (For a thorough explanation of degeneralization, refer to (Gastin and Oddoux 2001).)

Although the size and shape are fixed, the order in which the accepting sets are counted can be varied. If the number of accepting sets is n, there are $n!$ variations. Normally a heuristic, for example based on the sizes of the accepting sets, is used to select an order. Using some additional QuickCheck properties that compare the size of the result, we performed some experiments with different ordering heuristics. We tried to avoid having to calculate (and use) all possible degeneralizers, but unfortunately we could not find a heuristics that was consistently better than the random choice. However, since we are not worried about the performance of the translator, we decided to settle for a computationally more expensive solution. That is, we generate all possible automata and pick the best result after computing the synchronous product. This ensures that we get the smallest final automaton. We should note, that in many cases the reductions described in the next section actually produces the same final automaton regardless of the degeneralizer used. Finally, the justification for this more expensive solution is simply that more is hopefully gained in the model checking phase by having a smaller automaton, than what is spent in translation.

4.4 Reductions and optimizations

In automata theory there is a multitude of different reduction techniques and optimizations proposed. Some perform well on some structures, while others work best in completely different cases. We have chosen to implement some reduction algorithms that have proved useful for others, see for example (Etessami and Holzmann 2000; Giannakopoulou and Lerda 2002; Somenzi and Bloem 2000).

We have particularly opted for algorithms that reduce the size of the automaton, there are other algorithms that for example tries to make the automaton more deterministic (but also larger), see (Sebastiani and Tonetta 2003). Although a more deterministic automaton is sometimes preferable, we have chosen not to consider it in our implementation.

4.4.1 Simple reductions

We have implemented a couple of simple reductions:

- **Remove unnecessary transitions –** Unnecessary transitions are removed. For example two transitions from state X to state Y with the labels $a \wedge b$ and $a \wedge \neg b$ can be merged to one transition with the label a.
- **Remove non-reachable states –** States that cannot be reached from an initial state can be removed (together with their outgoing transitions).
- **Remove never accepting states –** States, from which it is impossible to reach an accepting state (or rather an accepting cycle) can be removed.
- **Reducing the number of accepting states –** Not technically a reduction, but it is favorable for the particular simulation reductions we have implemented to have few accepting states. Thus, accepting states that are not part of a cycle are removed from the set of accepting states.

It is worth pointing out that these reductions are usually not necessary for the initial result of the translation algorithm. However, after performing other reductions, also these simple reductions can be useful.

4.4.2 Bi-simulation reduction

Bi-simulation reduction is a standard reduction algorithm (see Kanellakis and Smolka (1983)). The reduction algorithm is based on a color-refinement partitioning of the states. The algorithm is adapted to the fact that transitions are labeled by conjunctions of propositional variables rather than just variables, we followed the algorithm presented by Etessami and Holzmann (2000).

Most problems with the implementation occurred due to the fact that most algorithm descriptions are rather imperative and thus took some time to convert into something not-so-ugly looking in Erlang. Again having the possibility to quickly find small counter examples when things went wrong helped a lot.

4.4.3 Strong fair simulation reduction

The most elaborate reduction algorithm we implemented was a *strong fair simulation* reduction. There are many different versions of fair simulation (see, e.g. Henzinger et al. (1997)). The version we implemented is described by Etessami and Holzmann (2000), it is rather similar to the bi-simulation reduction algorithm. However, more details are considered and a more fine grained ordering makes it possible to perform some more complicated reductions.

We had some difficulties implementing the strong fair simulation algorithm, mostly due to a misinterpretation of the algorithm description. (The *i-dominates* relation should be seen as a total order, if transition A i-dominates transition B then B **cannot also** i-dominate A. This is not clear from the definition.) While looking for this bug we were actually not helped by the QuickCheck tests, rather the opposite. Since errors occurred quite infrequently (and for rather complex automata), we were for a long time looking for a *less fundamental*(!!) error. Eventually we found the error, thanks to more basic debugging techniques, and could quickly verify that the algorithm implementation was working by running a large number of tests.

5. Results

Translator	Max size	Avg. size	Total size
erl_ltl2buchi	416	17.202	17202
erl_ltl2buchi+red	32	6.15	6150
erl_ltl2buchi+red+rew	31	6.005	6005
WRING	74	10.997	10997
LTL2BUCHI	31	6.066	6066

Table 1. Test results - 1000 random LTL formulas

To measure the performance in terms of size of the resulting automata we used a simple QuickCheck property, utilizing the built-in measure functions. An alternative would have been to use something readily available, like LBTT (Tauriainen 2001), but still, quite a bit of work would have been needed to write wrappers for the compared implementations. Therefore, since we had already used WRING and LTL2BUCHI as reference implementations during testing and thus had all the plumbing in place, we settled for the more lightweight QuickCheck measuring approach. The property listed in Fig. 8, is all that is needed to measure the performance for randomly generated LTL formulas. We test three different versions of our implementation: only the basic translation, translation + reductions and rewriting + translation + reductions.

Running the property for 1000 tests (1000 randomly generated LTL formulas) gives a result as presented in Table 1. The results vary slightly due to the random nature of the tests, but the numbers presented are representative. The table presents the maximal size of an individual translated automaton, as well as the average size of the automata.

We see that for random LTL formulas, we perform a lot (about 45%) better than WRING and also slightly (about 1%) better than *LTL2Buchi*. It is a result that we are satisfied with, since the alternative to writing a new implementation was to wrap *LTL2Buchi* and use it in McErlang. Nevertheless, we believe that there is still some room for improvement, and with the properties in place it should be easy to test new ideas for reduction algorithms in the future.

```
%% List of five different translations ,
%% used in prop_compare_size .
translations () ->
    [{"erl_ltl2buchi",
        fun ltl2buchi:translate_basic/1},
     {"erl_ltl2buchi+red",
        fun ltl2buchi:translate_norew/1},
     {"erl_ltl2buchi+red+rew",
        fun ltl2buchi:translate/1}
     {"wring",
        fun wring_wrap:run/1},
     {"ltl2buchi",
        fun ltl2buchi_wrap:run/1}
    ].

prop_compare_size() ->
    ?FORALL(Phi, (ltl_formula()),
      %% Filter formulas crashing Wring
      ?IMPLIES(wring_ok(Phi),
        begin
          Trs = [{N, F(Phi)} || {N, F} <-
               translations()],
          nested_measure(Trs, true)
        end)).
```

Figure 8. Performance measurement property

For two reasons we have not measured the speed of the translators; (1) it is impossible to fairly compare a native implementation with wrapped implementations called externally, and (2) the model checking connection for the translator makes size much more important than speed.

6. Summary

Using a property driven development for the implementation of an LTL-to-Büchi translator in Erlang turned out to be a nice experience. We had a fun time, and the implementation quickly stabilized into a mature translator. We believe that by first formulating the properties, we saved time both in testing and debugging and also in having a clearer picture of what to implement. Also, the shrinking facilities of QuickCheck meant that we usually got a fairly simple LTL formula for which the translator produced an incorrect automaton, which in the end probably reduced the time spent debugging the translator.

However, we should also note that we were helped (quite a lot) by the implementation task being very well defined. This also meant that the properties were not too hard to formulate. So, for situations where a clearly defined and well described algorithm should be implemented, this style of development is especially well suited. Having the paper about testing LTL-to-Büchi translations by Tauriainen and Heljanko (2002) was also very helpful.

In the end we managed to implement an LTL-to-Büchi translator that performs at least as well as our reference implementations WRING and LTL2BUCHI.

To further improve the QuickCheck experience it would be good to directly generate and *effectively* shrink Büchi automata. Especially when working with reduction algorithms it would be nice to not get the smallest LTL formula that translates into an automaton where the reduction fails, but rather the smallest automaton. We did implement a generator for Büchi automata, but shrinking them in an intelligent way was deemed a too complicated task and we did not have time to investigate this further.

Finally we should also comment on our focus to trade a smaller resulting automaton for a longer translation time. The run time of the LTL-to-Büchi translator is usually some couple of hundred *milliseconds*, while a model checking run could easily spend some hundred *seconds*. Thus spending a factor ten longer time in generation for gaining 10% of the model checking time is still a good trade-off.

Acknowledgments

This research was sponsored by EU FP7 Collaborative project *ProTest*, grant number 215868. Many thanks to the McErlang team (Lars-Åke Fredlund and Clara Benac Earle) for their efforts in including the translator in the McErlang release. Also thanks to Koen Claessen for helpful comments on testing and optimization of Büchi automata.

References

Thomas Arts, Laura M. Castro, and John Hughes. Testing Erlang data types with Quviq QuickCheck. In *ERLANG '08: Proc. of the 7th ACM SIGPLAN workshop on ERLANG*, pages 1–8, New York, NY, USA, 2008. ACM.

Kent Beck. *Test-driven development : by example*. Addison-Wesley, Boston, MA, 2003.

J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci.*, 1960.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

C. Benac Earle, L-Å. Fredlund, J. A. Iglesias, and A. Ledezma. Verifying Robocup teams. In *Proc. 5th International Workshop, Mochart 2008*, pages 34–48. Springer, 2008.

K. Etessami and G. Holzmann. Optimizing Büchi automata. In *CONCUR '00: Proc. of the 11th International Conference on Concurrency Theory*, pages 153–167, London, UK, 2000. Springer-Verlag.

L-Å. Fredlund and J.J. Sánchez Penas. Model checking a VoD server using McErlang. In *In proceedings of the 2007 Eurocast conference*, Feb 2007.

L-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2007.

P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV '01: Proc. of the 13th International Conference on Computer Aided Verification*, pages 53–65, London, UK, 2001. Springer-Verlag.

R. Gerth, D. A. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.

D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *FORTE '02: Proc. of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, pages 308–326, London, UK, 2002. Springer-Verlag.

T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. In *Information and Computation*, pages 273–287. Springer-Verlag, 1997.

J. Hughes. QuickCheck testing for fun and profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, pages 1–32. Springer-Verlag, Berlin Heidelberg, 2007.

P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *PODC '83: Proc. of the second annual ACM symposium on Principles of distributed computing*, pages 228–240, New York, NY, USA, 1983. ACM.

McErlang – https://babel.ls.fi.upm.es/trac/McErlang/. (Web page, 2009).

A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Nov 1977.

ProTest Project – http://www.protest-project.eu. (Web page, 2009).

J-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

R. Sebastiani and S. Tonetta. "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *12th IFIP WG 10.5 Advanced Research Working Conference, CHARME*. Springer-Verlag, 2003.

F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV '00: Proc. of the 12th International Conference on Computer Aided Verification*, pages 248–263, London, UK, 2000. Springer-Verlag.

H. Svensson and L-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Erlang '07: Proc. of the 2007 SIGPLAN workshop on Erlang Workshop*, pages 43–54, New York, NY, USA, 2007. ACM.

H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *STTT*, 4(1):57–70, 2002.

Heikki Tauriainen. lbtt 1.0.0 – an LTL-to-Büchi translator testbench. Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2001. Software.

M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 238–266, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

Moshe Y. Vardi. The Büchi complementation saga. In *STACS 2007*, volume 4393 of *Lecture Notes in Computer Science*, pages 12–22. Springer Berlin / Heidelberg, 2007.

M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 332–344, New York, 1986. IEEE Computer.

Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 185–194, Nov. 1983.