Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL

Alexander Schimpf[1], Stephan Merz[2], and Jan-Georg Smaus[1]

[1] University of Freiburg, Germany, {schimpfa|smaus}@informatik.uni-freiburg.de
[2] INRIA Nancy, France, Stephan.Merz@loria.fr

Abstract. We present the implementation in Isabelle/HOL of a translation of LTL formulae into Büchi automata. In automaton-based model checking, systems are modelled as transition systems, and correctness properties stated as formulae of temporal logic are translated into corresponding automata. An LTL formula is represented by a (generalised) Büchi automaton that accepts precisely those behaviours allowed by the formula. The model checking problem is then reduced to checking language inclusion between the two automata. The automaton construction is thus an essential component of an LTL model checking algorithm. We implemented a standard translation algorithm due to Gerth *et al.* The correctness and termination of our implementation are proven in Isabelle/HOL, and executable code is generated using the Isabelle/HOL code generator.

1 Introduction

The term *model checking* [2] subsumes several algorithmic techniques for the verification of reactive and concurrent systems, in particular with respect to properties expressed as formulae of temporal logics. More specifically, the context of our work are LTL model checking algorithms based on Büchi automata [19]. In this approach, the system to be verified is modelled as a finite transition system and the property is expressed as a formula φ of *linear temporal logic* (LTL). The formula φ constrains executions, and the transition system is deemed correct (with respect to the property) if all its executions satisfy φ. After translating the formula into a Büchi automaton [1], the model checking problem can be rephrased in terms of language inclusion between the transition system (interpreted as a Büchi automaton) and the automaton representing φ or, technically more convenient, as an emptiness problem for the product of the transition system and the automaton representing $\neg\varphi$.

In this paper, we present a verified implementation in Isabelle/HOL of the classical translation algorithm due to Gerth *et al.* [7] of LTL formulae into Büchi automata.[3] The automaton translation is at the heart of automaton-based model

[3] The Isabelle sources on which our paper is based are available at http://www.informatik.uni-freiburg.de/~ki/papers/diplomarbeiten/LTL2LGBA.zip. Extensive documentation on Isabelle can be found at http://isabelle.in.tum.de. Throughout this paper *Isabelle* refers to *Isabelle/HOL*.

checking algorithms, and an error in the design or the implementation of the translation algorithm compromises the soundness of the verdict returned by the model checker. Indeed, the original implementation of the translation proposed by Gastin and Oddoux [6] contained a flaw that went unnoticed for several years, despite wide-spread use within the Spin model checker. The purpose of our work is to demonstrate that it is feasible to obtain an executable program implementing such a translation from a formalisation in a modern interactive proof assistant. Assuming that the kernel of the proof assistant and the code generator are correct, we thus obtain a highly trustworthy implementation.

We chose the algorithm of Gerth *et al.* because it is well-known and representative of the problems that such algorithms pose. More recent algorithms such as [6] are known to behave better for larger LTL formulae, but they require additional automata-theoretic concepts, and we leave their formalisation as a worthwhile and challenging topic for future work.

The algorithm of Gerth *et al.* is based on the construction of a graph of nodes labelled with subformulae of the original formula, similar to a tableau construction [4]. Acceptance conditions on infinite runs complement the tableau and enforce "eventuality" (liveness) properties. The main theorem states that the generated automaton should accept precisely those words (system executions) that are models of the temporal formula. The correctness of the translation is by no means obvious; in fact, we already found proving the termination of the method to be quite challenging. In our formalisation, we limit ourselves to data structures and operations that are supported by the Isabelle code generator. In this way, extraction of executable code becomes straightforward, but we are limited to relatively low-level constructions.

The paper is organised as follows: In the next section, we provide some preliminaries on LTL and Büchi automata. In Sect. 3, we recall the algorithm proposed by Gerth *et al.* [7]. Section 4 presents our implementation of the algorithm. In Sect. 5, we discuss the proof of termination and correctness of this implementation. Section 6 concludes. The results presented in this paper were obtained within the Diploma Thesis of the first author [14].

2 Preliminaries

2.1 Linear Temporal Logic

Linear-time temporal logic LTL [13] is a popular formalism for expressing correctness properties about (runs of) reactive systems. It extends propositional logic by modal operators that refer to future points of time.

Definition 1. Let Prop be a finite, non-empty set of propositions. The set Φ of *LTL formulae* is inductively defined as follows:

- $\mathsf{Prop} \subseteq \Phi$;
- if $\varphi \in \Phi$ and $\psi \in \Phi$, then $\neg\varphi \in \Phi$, $\varphi \vee \psi \in \Phi$, $\mathsf{X}\varphi \in \Phi$ ("next φ"), and $\varphi \,\mathsf{U}\, \psi \in \Phi$ ("φ until ψ").

Further logical connectives can be defined as abbreviations. In particular, we will use the propositional constants $\top$ (true) and $\bot$ (false), as well as the operators $\wedge$ and V ("release"), which are the duals of $\vee$ and U.

The semantics of an LTL formula is defined with respect to a *(temporal) interpretation* $\xi = a_0 a_1 \ldots$, which is an ω-word[4] over 2^{Prop}, consisting of propositional interpretations $a_i \in 2^{\mathsf{Prop}}$. The set a_0 contains exactly those propositions that are true in the initial state of the temporal interpretation ξ, a_1 gives the propositions true in the second state, and so on. When $\xi = a_0 a_1 \ldots$, we write ξ_i for a_i and $\xi|_i$ for the suffix $a_i a_{i+1} \ldots$, which is itself a temporal interpretation.

Definition 2. The relation $\xi \vDash \varphi$ ("ξ *is a model of* φ" or "φ *holds of* ξ") is inductively defined as follows:

$$
\begin{array}{lll}
\xi \vDash p & \text{iff} & p \in \xi_0 \quad (p \in \mathsf{Prop}) \\
\xi \vDash \neg\varphi & \text{iff} & \xi \nvDash \varphi \\
\xi \vDash \varphi \vee \psi & \text{iff} & \xi \vDash \varphi \text{ or } \xi \vDash \psi \\
\xi \vDash \mathsf{X}\varphi & \text{iff} & \xi|_1 \vDash \varphi \\
\xi \vDash \varphi \,\mathsf{U}\, \psi & \text{iff} & \text{there exists } i \in \mathbb{N} \text{ such that } \xi|_i \vDash \psi \text{ and } \xi|_j \vDash \varphi \text{ for all } 0 \leq j < i.
\end{array}
$$

2.2 Generalised Büchi Automata

The automata-theoretic approach to LTL model checking [19] relies on translating LTL formulae φ to Büchi automata $\mathcal{A}_\varphi$ such that a word ξ is accepted by $\mathcal{A}_\varphi$ if and only if $\xi \vDash \varphi$. The following variant of Büchi automata underlies the algorithm by Gerth et al. [7].

Definition 3. A *generalised Büchi automaton* (GBA) $\mathcal{A}$ is tuple (Q, I, δ, F) where:

- Q is a finite set of states;
- $I \subseteq Q$ is the set of initial states;
- $\delta \subseteq Q \times Q$ is the transition relation;
- $F \subseteq 2^Q$ is the set of acceptance sets (the *acceptance family*).

An ω-word σ over Q is called *path* of $\mathcal{A}$ if $\sigma_0 \in I$ and $(\sigma_i, \sigma_{i+1}) \in \delta$ for all $i \in \mathbb{N}$. The *limit* of ω-word σ is given as $\mathsf{limit}(\sigma) := \{q \mid \exists_\infty n.\ \sigma_n = q\}$[5]. The GBA $\mathcal{A}$ *accepts* a path σ of $\mathcal{A}$ if $\mathsf{limit}(\sigma) \cap M \neq \emptyset$ holds for all $M \in F$.

Observe that Def. 3 does not mention an alphabet. Instead, it is conventional to label automaton states by sets of propositional interpretations and use these labels to define the acceptance of a temporal interpretation by a GBA. Formally, this is achieved by the following definition, where $\mathcal{D}$ is chosen as 2^{Prop}.

Definition 4. A *labelled generalised Büchi automaton* (LGBA) is given by a triple $(\mathcal{A}, \mathcal{D}, \mathcal{L})$ where:

[4] An ω-word over alphabet Σ is a sequence $s_0 s_1 \ldots$ where $s_i \in \Sigma$ for all $i \in \mathbb{N}$.

[5] The symbol $\exists_\infty$ means "there are infinitely many".

– $\mathcal{A} = (Q, I, \delta, F)$ is a GBA;
– $\mathcal{D}$ is a finite set of labels;
– $\mathcal{L} : Q \to 2^{\mathcal{D}}$ is the label function.

A path σ of $\mathcal{A}$ is *consistent with* an ω-word ξ over $\mathcal{D}$ if $\xi_i \in \mathcal{L}(\sigma_i)$ for all $i \in \mathbb{N}$. An LGBA *accepts* an ω-word ξ over $\mathcal{D}$ iff it (more precisely, its underlying GBA) accepts some path of $\mathcal{A}$ that is consistent with ξ.

In model checking, systems are modelled as Kripke structures, that is, finite transition systems whose states are labelled with propositional interpretations. A Kripke structure $\mathcal{K}$ is an LGBA whose underlying GBA has a trivial (empty) acceptance family, and whose label function assigns a single propositional interpretation to every state. Assuming that the LGBA $\mathcal{A}$ represents the complement of the LTL formula φ ($\mathcal{A}$ accepts precisely those executions of which φ does not hold), $\mathcal{K}$ is a model of φ if no execution is accepted by both $\mathcal{K}$ and $\mathcal{A}$, i.e. if the intersection of the languages accepted by the two automata is empty.

3 Generating an LGBA for an LTL formula

We recall the algorithm proposed by Gerth *et al.* [7] for computing an LGBA $\mathcal{A}_\varphi$ (with set of labels 2^{Prop}) for an LTL formula φ such that $\mathcal{A}_\varphi$ accepts a temporal interpretation ξ iff $\xi \vDash \varphi$.

The construction of $\mathcal{A}_\varphi$ proceeds in three stages. First, one builds the graph of the underlying GBA, using a procedure similar to a tableau construction [4]. Second, the function for labelling states of the LGBA is defined. Finally, the acceptance family is determined based on the set of "until" subformulae of φ. We now describe each stage in more detail.

The first step builds a graph of nodes (which will become the automaton states) that contain subformulae of φ. Intuitively, a node "promises" that the formulae it contains hold of any temporal interpretation that has an accepting run starting at that node. The construction is essentially based on "recursion laws" of LTL such as

$$\mu \,\mathsf{U}\, \psi \;\leftrightarrow\; \psi \vee (\mu \wedge \mathsf{X}(\mu \,\mathsf{U}\, \psi)) \tag{1}$$

that are used to split a promised formula into promises for the current state and for the successor state. The initial states of the automaton will be precisely those nodes that promise φ.

Without loss of generality, we assume that φ is given in *negation normal form* (NNF), i.e. the negation symbol is only applied to propositions. Transformation to NNF is straightforward once we include the dual operators $\wedge$ and V among the set of logical connectives, using laws such as $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$.

Gerth *et al.* [7] represent each node of the graph by a record with the following fields:

– *Name*: a unique identifier of the node.

– *Incoming*: the set of names of all nodes that have an edge pointing to the current node. Using this field, the entire graph is represented as the set of its nodes.
– *New*: A set of LTL formulae promised by this node but that have not yet been processed. This set is used during the construction and is empty for all nodes of the final graph.
– *Old*: A set of LTL formulae promised by this node and that have already been processed.
– *Next*: A set of LTL formulae that all successor nodes must promise.
– *Father*: During the construction, nodes will be split. This field contains the name of the node from which the current one has been split. It is used by Gerth *et al.* solely for reasoning about the algorithm, and we will not mention it any further.

The algorithm successively moves formulae from *New* to *Old*, decomposing them, and inserting subformulae into *New* and *Next* as appropriate. When the *New* field is empty, a successor node is generated whose *New* field equals the *Next* field of the current node. The algorithm maintains a list of all nodes generated so far to avoid generating duplicate nodes; this is essential for ensuring termination of the algorithm. More formally, the algorithm is realised by the function expand whose pseudo-code is reproduced in Fig. 1. For reasons of space and clarity, we omit some parts of the code in this presentation, in particular, some of the cases for the currently considered formula η, while preserving the original line numbering.

The automaton graph is constructed by the following function call:

$$\begin{aligned}&\mathsf{expand}([\,\mathit{Name} \Leftarrow \mathbf{new\_name}(), \mathit{Incoming} \Leftarrow \{\mathsf{init}\},\\ &\qquad \mathit{New} \Leftarrow \{\varphi\}, \mathit{Old} \Leftarrow \emptyset, \Leftarrow \mathit{Next} \Leftarrow \emptyset\,],\ \emptyset)\end{aligned} \tag{2}$$

where φ is the input LTL formula and init is a reserved identifier: all nodes whose *Incoming* field contains init will be initial states of the automaton.

In the second step of the construction, we define the function labelling the nodes with sets of propositional interpretations, each represented as the set of propositions that evaluate to true. The label of a node q is defined as the set of interpretations that are compatible with $\mathit{Old}(q)$. Formally, let

$$\mathit{Pos}(q) = \mathit{Old}(q) \cap \mathsf{Prop} \quad \text{and} \quad \mathit{Neg}(q) = \{\eta \in \mathsf{Prop} \mid \neg\eta \in \mathit{Old}(q)\}.$$

A propositional interpretation X is compatible with q iff it satisfies all atomic propositions in $\mathit{Pos}(q)$ but none in $\mathit{Neg}(q)$. This motivates the definition

$$\mathcal{L}(q) = \{X \subseteq \mathsf{Prop} \mid X \supseteq \mathit{Pos}(q),\ X \cap \mathit{Neg}(q) = \emptyset\}. \tag{3}$$

It remains to define the acceptance family of the LGBA. Reconsider the "recursion law" (1) for the U operator, which is implemented by lines 20–27 of the code of Fig. 1. Every node "promising" a formula $\mu\,\mathsf{U}\,\psi$ has one successor promising ψ and a second successor promising μ and $\mathsf{X}(\mu\,\mathsf{U}\,\psi)$. Thus, the graph

```
 3: function expand(Node, Nodes_Set)
 4:   if New(Node)=∅ then
 5:     if ∃ND∈Nodes_Set with Old(ND)=Old(Node) and Next(ND)=Next(Node) then
 6:     Incoming(ND):=Incoming(ND)∪Incoming(Node);
 7:       return(Nodes_Set)
 8:     else return(expand([Name⇐new_name(), Incoming⇐{Name(Node)},
10:         New⇐Next(Node), Old⇐ ∅, Next⇐ ∅], {Node}∪Nodes_Set))
11:   else
12:     let η ∈New(Node);
13:     New(Node):=New(Node)\{η};
14:     case η of
15:     ¬P ⇒
18:         Old(Node):=Old(Node)∪{η};
19:       return(expand(Node, Nodes_Set));
20:     η = μ U ψ ⇒
21:       Node1:=[Name⇐new_name(), Incoming⇐Incoming(Node),
22:         New⇐New(Node)∪({μ}\Old(Node))
23:         Old⇐Old(Node)∪{η}, Next⇐Next(Node)∪{η}];
24:       Node2:=[Name⇐new_name(), Incoming⇐Incoming(Node),
25:         New⇐New(Node)∪({ψ}\Old(Node))
26:         Old⇐Old(Node)∪{η}, Next⇐Next(Node)];
27:       return(expand(Node2, expand(Node1, Nodes_Set)));
32: end expand
```

Fig. 1. The algorithm by Gerth *et al.* [7] (incomplete).

of the LGBA may contain paths such that all nodes along the path promise μ but no node promises ψ. Such paths are not models of $\mu \,\mathsf{U}\, \psi$, which requires ψ to be true eventually, and the acceptance family is defined in order to exclude them. Formally, we define for each formula $\mu \,\mathsf{U}\, \psi$ the set of nodes

$$F_{\mu \mathsf{U} \psi} \;=\; \{q \in Q \mid \mu \,\mathsf{U}\, \psi \notin \mathit{Old}(q) \text{ or } \psi \in \mathit{Old}(q)\}, \tag{4}$$

and define the acceptance family F as

$$F \;=\; \{F_{\mu \mathsf{U} \psi} \mid \mu \,\mathsf{U}\, \psi \text{ is a subformula of } \varphi\}. \tag{5}$$

4 Implementation in Isabelle

4.1 LTL Formulae

We represent LTL formulae in Isabelle as an inductive data type. For the purposes of this presentation, we restrict to NNF formulae, although our full development also includes unrestricted LTL formulae and NNF transformation. For simplicity, we represent atomic propositions as strings; alternatively, the type of propositions could be made a parameter of the data type definition.

```
datatype
  frml = LTLTrue            ("true")
       | LTLFalse           ("false")
       | LTLProp string     ("prop'(_')")
       | LTLNProp string    ("nprop'(_')")
       | LTLAnd frml frml   ("_ and _")
       | LTLOr frml frml    ("_ or _")
       | LTLNext frml       ("X _")
       | LTLUntil frml frml ("_ U _")
       | LTLUDual frml frml ("_ V _")
```

The above definition includes the concrete syntax for each clause of the data type. For example, `(X prop(''p'')) and prop(''q'')` would be the Isabelle representation of $(\mathsf{X}p) \wedge q$.

We next introduce types for representing ω-words and temporal interpretations, and define the semantics of LTL formulae by a straightforward primitive recursive function definition.

```
types
  'a word = nat ⇒ 'a
  interprt = "(string set) word"

fun semantics :: "[interprt, frml] ⇒ bool" ("_ ⊨ _" [80,80] 80)
where
  "ξ ⊨ true = True"
| "ξ ⊨ false = False"
| "ξ ⊨ prop(q) = (q∈ξ(0))"
| "ξ ⊨ nprop(q) = (q∉ξ(0))"
| "ξ ⊨ φ and ψ = (ξ ⊨ φ ∧ ξ ⊨ ψ)"
| "ξ ⊨ φ or ψ = (ξ ⊨ φ ∨ ξ ⊨ ψ)"
| "ξ ⊨ X φ = (suffix 1 ξ ⊨ φ)"
| "ξ ⊨ φ U ψ = (∃i. suffix i ξ ⊨ ψ ∧ (∀j<i. suffix j ξ ⊨ φ))"
| "ξ ⊨ φ V ψ = (∀i. suffix i ξ ⊨ ψ ∨ (∃j<i. suffix j ξ ⊨ φ))"
```

4.2 Büchi Automata

We now encode GBAs and LGBAs in Isabelle, following Defs. 3 and 4. In this encoding we approximate the set Q of states by a type parameter `'q`, which will later be instantiated by the type representing the nodes of the graph. Although not enforced by the definition, finiteness of the actual set of nodes will be ensured by the termination of the algorithm, which produces states one by one.

GBAs and LGBAs are naturally modelled as *records* in Isabelle. Since we aim at producing executable code, all sets that appear in the original definition are represented as lists.

```
record 'q gba =
  initial :: "'q list"
```

```
  trans   :: "('q × 'q) list"
  accept  :: "'q list list"

record 'q lgba =
  gbauto :: "'q gba"
  label  :: "'q ⇀ string list list"
```

The node labelling function is represented by a *partial* function (denoted by the ⇀ symbol) because it needs only be defined over actual states of the LGBA, whereas the type `'q` may contain extra elements.

It remains to define the runs and the acceptance family of (L)GBAs. The following definitions are a straightforward transcription of Def. 3: the utility function `set` from the Isabelle library computes the set of list elements, the `limit` function is defined as indicated in Def. 3.

```
definition gba_path :: "['q gba, 'q word] ⇒ bool" where
  "gba_path A σ
   ≡ σ 0 ∈ set (initial A) ∧
     (∀n. ((σ n), σ (Suc n)) ∈ set (trans A))"
definition gba_accept :: "['q gba, 'q word] ⇒ bool" where
  "gba_accept A σ
   ≡ gba_path A σ ∧
     (∀i<length (accept A). limit σ ∩ set (accept A!i) ≠ {})"
```

The acceptance condition for an LGBA is defined in a similar fashion. Finally, the predicate `lgba_accept` characterises the language of an LGBA: a temporal interpretation ξ is accepted by the LGBA A if there exists some path σ that is accepted by the GBA underlying A and that is consistent with ξ (cf. Def. 4).

```
definition lgba_accept :: "['q lgba, interprt] ⇒ bool"
where
  "lgba_accept A ξ
   ≡ ∃σ. (∀i. ξ i ∈ set (map set (the (label A (σ i)))))
         ∧ gba_accept (gbauto A) σ"
```

The use of the function "`the`" in the above code is a technicality related to the fact that the node labelling function is, in principle, partial. What matters is that "`the (label A (σ i))`" is of type `string list list`.

4.3 Translation from LTL to LGBA

We now formalise in Isabelle the algorithm due to Gerth *et al.* that we have presented informally in Sect. 3. As discussed there, three elementary steps have to be addressed:

- construct the graph of the underlying GBA using the expand function;
- define the acceptance family of the GBA;

```
function (sequential) expand :: "[cnode, node list] ⇒ node list"
where
  "expand ([], n) ns
   = (if (∃nd∈set ns. set (old nd) = set (old n) ∧
                      set (next nd) = set (next n))
      then upd_nds (λn nd. set (old nd) = set (old n) ∧
                           set (next nd) = set (next n)) ns n
      else expand (next n,
                   (|name = Suc(name n),
                    incoming = [name n],
                    old = [],
                    next = [] |)) (n#ns))"

| "expand ((nprop(q))#fs, n) ns
   = expand (fs, n(| old := (nprop(q))#(old n) |)) ns"

| "expand ((μ U ψ) #fs, n) ns
   = (let nds = expand (μ#fs,
                        n(| old := (μ U ψ)#(old n),
                            next := (μ U ψ)#(next n) |)) ns
      in expand (ψ#fs,
                 n(| name := ...,
                     old := (μ U ψ)#(old n) |)) nds)"
```

Fig. 2. The Isabelle implementation of expand, simplified.

– compute the labelling of the states of the LGBA with sets of propositional interpretations.

The algorithm expand constructs a graph, represented as a set of nodes. In Isabelle, we again use lists instead of finite sets in order to simplify code generation. We represent node names as integers, and model a node as a record containing the fields introduced in Sect. 3. We omit the *Father* field, which is unnecessary for the construction of the graph. We also replace the field *New*, which is used only during the construction, by an extra argument to the expand function. More precisely, the first argument of the function is of type `cnode`, defined as a pair of a formula list and a node.

```
record node =
  name :: nat
  incoming :: "nat list"
  old :: "frml list"
  next :: "frml list"
types cnode = "frml list * node"
```

Figure 2 contains the fragment of the definition of function expand in Isabelle that corresponds to the pseudo-code shown in Fig. 1. The function `upd_nds` merges

the `incoming` fields of the current node with those of the already constructed nodes whose `old` and `next` fields agree with those of the current node.

For the sake of presentation, the code shown in Fig. 2 is somewhat simplified with respect to our Isabelle theories: the actual definition produces a pair consisting of a list of nodes and the highest used node name, which is used in the (omitted) definition of the name of the node created in the second call to `expand` in the clause for "until" formulae. Moreover, the actual definition checks for duplicates whenever a formula is added to the `old` or `next` components of a node.

The graph for an LTL formula is computed by the function `create_graph`, which in analogy to (2) is defined as

```
definition create_graph :: "frml ⇒ node list"
where
  "create_graph φ
   ≡ expand ([φ], ⦇ name = 1, incoming = [0],
                    old = [], next = [] ⦈) []"
```

We now address the second problem, i.e. the computation of the acceptance family for an LTL formula and a graph represented as a list of nodes. The following function `accept_family` is a quite direct transcription of the definition of the acceptance family in (5):

```
definition accept_family :: "[frml, node list] ⇒ node list list"
where
  "accept_cond φ ns
   ≡ map (λη. case η of
                _ U ψ ⇒ [q←ns. η∈set(old q) ⟶ ψ∈set(old q)])
         (all_until_frmls φ)"
```

where `all_until_frmls` computes the list of "until" subformulae of the argument formula, without duplicates. It is now straightforward to define a function `create_gba` that constructs a GBA (of type `node gba`) from a node list representing the graph.

It remains to compute the function labelling the nodes with sets of propositional interpretations, in order to obtain an LGBA. The following definitions implement the labelling defined by (3) in a straightforward way.

```
definition
  gen_label :: "[string list list, node] ⇒ string list list"
where
  "gen_label lbls n
   ≡ [xs←lbls. set (pos_props (old n)) ⊆ set xs
               ∧ list_inter xs (neg_props (old n)) = []]"
definition
  create_lgba :: "frml ⇒ node lgba"
where
```

```
"create_lgba φ
 ≡ (let ns = create_graph φ in
     (‖ gbauto = create_gba φ ns,
       label = [ns[↦]map (gen_label (list_Pow (get_props φ)))
                         ns] ‖))"
```

The auxiliary functions `pos_props` and `neg_props` compute the lists of positive and negative literals contained in a list of formulae; `get_props` computes the list of atomic propositions contained in a temporal formula.

4.4 Code generation

We have set up our theories in such a way that they use only data types and operations supported by the code generator, except for certain tests that convert lists to sets. In order to make these tests executable, we derive some auxiliary lemmas such as

```
lemma [code inline]:
  "set xs ⊆ set ys ⟷ list_all (λx. x mem ys) xs"
lemma [code inline]:
  "set xs = set ys ⟷ set xs ⊆ set ys ∧ set ys ⊆ set xs"
```

After these preliminaries, executable code can be extracted by simply issuing the command

```
export_code create_lgba in OCaml file "ltl2lgba.ml"
```

from the Isabelle theory file. This command produces an OCaml module containing the function `create_lgba` and all definitions and functions on which that function depends.

In order to use this code we have manually written a parser and driver program that parses an LTL formula, calls the function `create_lgba`, and outputs the result. We have used this program to generate automata corresponding to formulae φ_n that are representative of the verification of liveness properties under fairness constraints[6]

$$\varphi_n \;\equiv\; \neg\,((\mathsf{GF}p_1 \wedge \ldots \wedge \mathsf{GF}p_n) \Longrightarrow \mathsf{G}(q \Longrightarrow \mathsf{F}r))$$

for atomic propositions p_i, q, and r.

We have compared our code with implementations of the algorithm of Gerth *et al.* that are available in the tools Spin (`http://spinroot.com`) and Wring (`http://vlsi.colorado.edu`). The running times (in seconds, on a dual-core notebook computer with a 2.4GHz CPU and 2GB of RAM) for translating φ_n are shown in the table on the right. However, this comparison is

	Our code	Spin	Wring
$n = 5$	30	> 1200	90
$n = 6$	540	> 1200	900

Table 1: Runtimes.

[6] $\mathsf{F}\psi$ ("finally ψ") is an abbreviation for $\top\,\mathsf{U}\,\psi$; $\mathsf{G}\psi$ ("globally ψ") denotes $\neg\mathsf{F}\neg\psi$.

not quite fair, because the other tools go on to translate the LGBA to ordinary Büchi automata. We plan to formalise this additional (polynomial) translation in the future, but take the present results as an indication that the execution times of the implementation generated from Isabelle are not prohibitive.

We have used the LTL-to-Büchi translator testbench [17] for gaining additional confidence in our program, including the hand-written driver. As expected, our code passes all the tests.

5 Verifying the Automaton Construction

Our main motivation for implementing the algorithm in Isabelle is of course the possibility to verify the correctness of our definitions. Assuming we trust Isabelle's proof kernel and its code generator, we obtain a verified program for translating LTL formulae into LGBA. We outline the correctness proof in this section. In fact, we must address two subproblems: we prove that the function `expand` terminates on all arguments, and we show that a temporal interpretation is accepted by the resulting LGBA iff it is a model of the input formula.

5.1 Termination

HOL is a logic of total functions, and it is essential for consistency to prove that every function that we define terminates. Indeed, Isabelle inserts a termination predicate in all theorems that involve a function whose termination has not been proven. Termination of the `expand` function (cf. Fig. 2) is not obvious on first sight but, remarkably, is not discussed at all in the original paper [7].

Consider Fig. 2. A call to `expand` is of the form `expand (fs,n) ns`. Now in all cases of the definition, except the first one, some formula is removed from `fs`, suggesting a well-founded ordering based on the size of the list `fs`. (This observation is also true of the cases of the definition omitted in Fig. 2.)

However, that simple definition breaks down for the first case where argument `fs` equals `[]`. Indeed, the recursive call constructs a new node based on the contents of the `next` field of the node `n`. In this case, the termination argument must be based on the argument `ns` of the function call. The apparent difficulty here is that this list does not become shorter on recursive calls, but (potentially) longer, so it is not completely obvious how to define a well-founded order. The solution here is to find a suitable upper bound for the argument `ns`. This can be done using the fact that all the nodes that are ever constructed contain subformulae of the input formula φ in their fields `old` and `next`, the same holds for the argument `fs` of formulae to process, and no two different nodes containing the same formulae in their `old` and `next` fields are ever constructed. It follows that there are only finitely many possible nodes since there exist only finitely many distinct sets of subformulae of φ. Very roughly speaking, the well-founded order by which argument `ns` decreases is given by `(LIM` φ `- ns)` where `LIM` is a function that calculates the appropriate upper bound given an LTL formula φ. The actual definition of the upper bound, which appears in the definition

of the ordering below, depends on the arguments of function `expand`, not the formula φ.

The two orderings are combined lexicographically, that is to say, either the argument `ns` decreases w.r.t. the ordering discussed above, or the `ns` argument stays the same and there is a decrease on the `fs` argument.

The termination proof is complicated further by the fact that we have a nested recursive call in the last case. This is obvious in line 27 in Fig. 1, but the `let` expression in Fig. 2 amounts to the same. We therefore start off by showing a partial termination property, which states that if `expand` terminates, then `nds` $\supseteq$ `ns`, where `nds` is the result computed by the inner call (see Fig. 2). This partial result is then used to show that the arguments of the outer recursive call are smaller according to the well-founded ordering explained above.

The termination order is formally defined in Isabelle as follows:

```
abbreviation
  "expand_term_ord ≡
     inv_image (finite_psubset <*lex*> less_than)
               (λ(n, ns). (nds_limit n ns - (old_next_pair ‘ set ns),
                           size_frml_list (fst n)))"
```

We explain this definition. The termination order compares pairs of the form (n, ns) where n is a `cnode` and ns is a `node list`. This corresponds exactly to the argument types of `expand`. The function $\lambda(\mathtt{n}, \mathtt{ns}).\ldots$ in the above definition turns (n, ns) into another pair, say (st, sz), where st is given by the `old` and `next` fields of all nodes in ns and *subtracting* those from the set of all *possible* `old` and `next` fields—i.e., st states "how far ns is from the limit". The second argument sz is simply the length of the list appearing as the first component of the pair n. To compare two pairs (n, ns) and (n', ns'), the function is used to compute the corresponding (st, sz) and (st', sz'), and those pairs are compared using a lexicographical combination of $\subseteq$ and $\leq$.

The formal termination proof takes about 500 lines of Isar proof script.

5.2 Correctness

We now address the proper correctness proof of the algorithm, whose idea is presented in the original paper [7]. We have to prove that the LGBA computed by function `create_lgba` φ accepts precisely those temporal structures that are a model of φ. Formally, this is expressed as the Isabelle theorem

```
theorem lgba_correct:
  assumes "∀i. ξ i ∈ Pow (set (get_props φ))"
    shows "lgba_accept (create_lgba φ) ξ ⟷ ξ ⊨ φ".
```

The hypothesis of the theorem states that ξ is a temporal interpretation over 2^{Prop} where Prop is the set of atomic propositions that occur in φ (cf. Sect. 2.2).

As explained in Sect. 3, the idea of the construction is to construct nodes that "promise" certain formulae and to make sure that these promises are enforced

along any path starting at that node. However, the graph construction by itself can ensure this only partly. For example, we can prove the following lemma about "until" formulae promised by a node:

```
lemma L4_2a:
  assumes "gba_path (gbauto (create_lgba φ)) σ"
      and "f U g ∈ set (old (σ 0))"
    shows "(∀i. {f, f U g} ⊆ set (old (σ i))
                ∧ g ∉ set (old (σ i)))
           ∨ (∃j. (∀i<j. {f, f U g} ⊆ set (old (σ i)))
                  ∧ g ∈ set (old (σ j)))".
```

In other words, we know for any path that starts at a node promising formula $f \mathsf{U} g$ that f and $f \mathsf{U} g$ are promised as long as g is not promised. However, we cannot be sure that g will indeed be promised by some node along the path. We defined the acceptance family precisely in a way to make sure that such paths are non-accepting, and indeed we can prove the following stronger lemma about the *accepting* paths starting at a node promising some formula $f \mathsf{U} g$:

```
lemma L4_2b:
  assumes "gba_path (gbauto (create_lgba φ)) σ"
      and "f U g ∈ set (old (σ 0))"
      and "gba_accept (gbauto (create_lgba φ)) σ"
    shows "∃j. (∀i<j. {f, f U g} ⊆ set (old (σ i)))
               ∧ g ∈ set (old (σ j))"
```

The proof of theorem `lgba_correct` above relies on similar lemmas for each temporal operator, and then proves by induction on the structure of LTL formulae that all formulae promised along an accepting path indeed hold of the corresponding suffix of the temporal interpretation. For the proof of the "if" direction of theorem `lgba_correct` we inductively construct an accepting path for any temporal interpretation satisfying a formula. The length of the overall correctness proof is about 4500 lines of Isar proof script. The effort of working out the Isabelle proofs was around four person months.

6 Conclusion

In this paper we have presented a formally verified definition of labelled generalised Büchi automata in the interactive proof assistant Isabelle. Our formalisation is based on the classical algorithm by Gerth *et al.* [7], and Isabelle can generate executable code from our definitions. In this way, we obtain a highly trustworthy program for a critical component of a model checking engine for LTL.

Few formalisations of similar translations have been studied in the literature. Schneider [15] presents a HOL conversion for LTL that produces a symbolic encoding of an LGBA, which can be used in connection with a symbolic (in particular BDD-based) model checker. In contrast, our implementation produces

a full LGBA that can be used with explicit-state LTL model checkers. Moreover, it generates a stand-alone program that can be used independently of any particular proof assistant. The second author [11] previously presented a formalisation of weak alternating automata (WAA [12]), including a translation of LTL formulae into WAA. Due to their much richer combinatorial structure, WAA afford a rather straightforward LTL translation of linear complexity, whereas the translation into (generalised) Büchi automata is exponential. Indeed, the main contribution of [11] was the formalisation of a game-theoretic argument due to [10, 18] that underlies a complementation procedure for WAA.

Since the translation of LTL formulae to Büchi automata is of exponential complexity, one cannot expect to translate large formulae. Fortunately, the formulae that express typical correctness properties of concurrent systems are quite small. Although efficiency was not of much concern to us during the development of our theories, our experiments so far indicate that the extracted program does not behave significantly worse than existing implementations of the algorithm of Gerth *et al.* Of course, several improvements to the code are possible. For example, we could represent the sets of propositional interpretations labelling the automaton states symbolically instead of through an explicit enumeration, for example using a Boolean function that checks whether an interpretation is consistent with the label. Optimisations at a lower level could be obtained by replacing the list representation of finite sets with a more efficient data structure.

More significant optimisations could be achieved by basing the construction on a different algorithm altogether. Although the construction of Gerth *et al.* is well known and widely implemented, several alternative constructions have been studied in the literature [3, 16, 6, 5, 8], and the algorithm presented in [6] is widely considered to behave best in practice. This algorithm makes use of more advanced automata-theoretic notions, including WAA and various simulation relations on WAA and Büchi automata. These concepts have wider applications than just the automata constructions used in model checkers, including the complementation of ω-automata [9] and the synthesis of concurrent systems.

Encouraged by the success we have had so far, we would indeed like to formalise the construction of [6] in future work. Our current formalisation will continue to serve as an important building block that contains essential, fundamental concepts.

References

1. R. Büchi. On a decision method in restricted second-order arithmetic. In *Intl. Cong. Logic, Methodology, and Philosophy of Science 1960*, pages 1–12. Stanford University Press, 1962.
2. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2002.
3. M. Daniele, F. Giunchiglia, and M. Vardi. Improved automata generation for linear temporal logic. In *11th Intl. Conf. Computer Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 249–260, Trento, Italy, 1999. Springer-Verlag.
4. M. C. Fitting. *Proof Methods for Modal and Intuitionistic Logic*. Synthese Library: Studies in Epistemology, Logic, Methodology and Philosophy of Science. D. Reidel, Dordrecht, The Netherlands, 1983.

5. C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In O. H. Ibarra and Z. Dang, editors, *8th Intl. Conf. Implementation and Application of Automata (CIAA 2003)*, volume 2759 of *LNCS*, pages 35–48, Santa Barbara, CA, USA, 2003.
6. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *13th Intl. Conf. Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 53–65. Springer-Verlag, 2001.
7. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *15th Intl. Symp. Protocol Specification, Testing, and Verification (PSTV 1996)*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1996.
8. S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi. On complementing nondeterministic Büchi automata. In D. Geist and E. Tronci, editors, *12th IFIP Work. Conf. Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *LNCS*, pages 96–110. Springer-Verlag, 2003.
9. O. Kupferman and M. Vardi. Complementation constructions for nondeterministic automata on infinite words. In N. Halbwachs and L. Zuck, editors, *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *LNCS*, pages 206–221, Edinburgh, Scotland, 2005. Springer-Verlag.
10. O. Kupferman and M. Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. Comput. Log.*, 2(3):408–429, 2001.
11. S. Merz. Weak alternating automata in Isabelle/HOL. In M. Aagaard and J. Harrison, editors, *13th Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs 2000)*, volume 1869 of *LNCS*, pages 424–441. Springer-Verlag, 2000.
12. D. Muller, A. Saoudi, and P. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *3rd IEEE Symp. Logic in Computer Science (LICS 1988)*, pages 422–427, Edinburgh, Scotland, 1988. IEEE Press.
13. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
14. A. Schimpf. Implementierung eines Verfahrens zur Erzeugung von Büchi-Automaten aus LTL-Formeln in Isabelle. Diplomarbeit, Albert-Ludwigs-Universität Freiburg, 2008. `http://www.informatik.uni-freiburg.de/~ki/papers/diplomarbeiten/schimpf-diplomarbeit-08.pdf`.
15. K. Schneider and D. W. Hoffmann. A HOL conversion for translating linear time temporal logic to ω-automata. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *12th Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs 1999)*, volume 1690 of *LNCS*, pages 255–272, Nice, France, 1999. Springer-Verlag.
16. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. Emerson and A. Sistla, editors, *12th Intl. Conf. Computer Aided Verification (CAV 2000)*, volume 1633 of *LNCS*, pages 257–263, Chicago, IL, 2000. Springer-Verlag.
17. H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer*, 4(1):57–70, 2002. See also `http://www.tcs.hut.fi/Software/lbtt/`.
18. W. Thomas. Complementation of Büchi automata revisited. In G. Rozenberg and J. Karhumäki, editors, *Jewels are forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 109–122. Springer-Verlag, 2000.
19. M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.