The CAVA Automata Library

Peter Lammich

Technische Universität München, lammich@in.tum.de

Abstract. We report on the graph and automata library that is used in the fully verified LTL model checker CAVA. As most components of CAVA use some type of graphs or automata, a common automata library simplifies assembly of the components and reduces redundancy.
The CAVA automata library provides a hierarchy of graph and automata classes, together with some standard algorithms. Its object oriented design allows for sharing of algorithms, theorems, and implementations between its classes, and also simplifies extensions of the library. Moreover, it is integrated into the Automatic Refinement Framework, supporting automatic refinement of the abstract automata types to efficient data structures.
Note that the CAVA automata library is work in progress. Currently, it is very specifically tailored towards the requirements of the CAVA model checker. Nevertheless, the formalization techniques presented here allow an extension of the library to a wider scope. Moreover, they are not limited to graph libraries, but apply to class hierarchies in general.

1 Introduction

CAVA [3] is a fully verified and executable LTL model checker for finite state systems à la SPIN [5]. It is formalized and verified in Isabelle/HOL [10]. The design of CAVA follows a stepwise refinement approach [1]: An initial abstract formalization of the model checker is refined to an efficiently executable program, involving many intermediate refinement steps. The stepwise refinement allows a separation of concerns: The correctness proofs are done on the abstract level, which is not cluttered with implementation details, and thus allows for much simpler and clearer proofs. The subsequent refinement steps then concentrate on efficiently implementing some details of the abstract algorithm. In Isabelle/HOL, this approach is supported by the Isabelle Refinement Framework [8,7].

Figure 1 shows the overall architecture of CAVA. It follows a standard approach for LTL model checkers: The input is an LTL formula and a model, which is described either as a while program over Boolean variables or in Promela, the modeling language of SPIN. The model is converted to a Kripke structure, i. e. a directed graph with a set of initial nodes and sets of atomic propositions annotated at the nodes.

The LTL formula is converted by Gerth's Algorithm [4] to a generalized Büchi automaton (GBA), which accepts all infinite words that do *not* satisfy the formula. The GBA is then converted to an indexed representation, which is more suitable for the following steps.

Then, the synchronous product of the Kripke structure and the indexed GBA is computed, resulting in an indexed generalized Büchi graph (GBG). Finally, the indexed GBG is checked for emptiness by either using a strongly connected component algorithm, or by degeneralizing it to a Büchi Graph (BG) and using a nested depth first search. The result of the emptiness check either declares the automaton as empty, in which case the model satisfies the formula, or it returns a counterexample, which is a representation of an infinite run of the model that violates the formula.

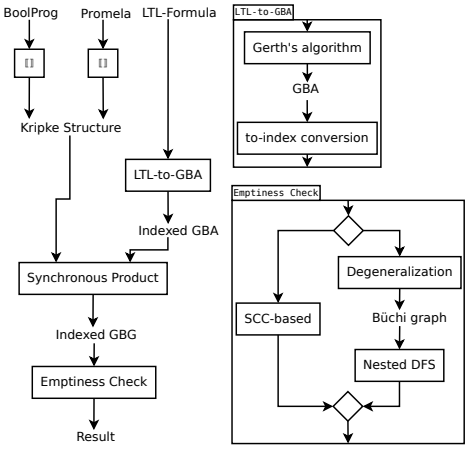

Fig. 1: Structure of the CAVA Model Checker

The different components of CAVA are implemented separately, and also maintained by different developers. As the interfacing between the components merely involves passing around different types of automata, there is a need for a common automata library that is used by all components.

Moreover, the different automata types share common properties, which can be exploited by a formalization: For example, both Büchi automata and Kripke structures are directed graphs with a set of root nodes, and all properties of directed graphs hold for both types. A main design goal of the CAVA Automata Library is to share common theorems, algorithms, and data structures between the different graph and automata classes, and thus reduce redundancy in the formalization.

The CAVA Automata Library has evolved over many years and is still evolving, as do the refinement techniques available for Isabelle. The initial automata library used by CAVA was written by Thomas Tuerk, and extended by Alexander Schimpf. It covered a wide spectrum of automata, however, not exactly what was needed by CAVA:

- First of all, CAVA is most conveniently formalized with state-labeled automata, while Tuerk's library only supported edge-labeled automata. This led to some unnecessary conversions that cluttered the proofs and even the implementation: For example, the Generalized Büchi automata were implemented by taking an implementation for edge-labeled automata from Tuerk's library, setting the type of edge labels to unit, and then adding another map for node labels, resulting in unnecessarily complicated definitions and proofs.
- Another source of unnecessary complication were the automata used to represent the Kripke structure and the synchronous product. A natural assumption is that the set of states reachable by the system is finite, whereas the type of the state space is typically infinite.

However, the automata library only supported automata with an explicit set of states that had to be finite, and the transition relation had to respect this set of states. Thus, the transition relation had to be artificially reduced to the set of reachable states. On the abstract level, this restriction is quite harmless. However, on the concrete level, this restriction is infeasible to implement, as it would require exploration of the whole state space first, which is contrary to the idea of an on-the-fly model checker. Luckily, the search algorithms only query transitions from reachable states, such that the implementation could circumvent this problem, however, at the cost of a quite complicated proof that broke the stepwise refinement approach.

When adding more features to the initial CAVA model checker, the problems described above led to a complete reimplementation of the automata library, tailored to the needs of CAVA. This paper reports on the resulting automata library, which is already used by CAVA, but is still work in progress, as it undergoes continuous extensions and simplifications. We also report on the formalization techniques that we used to achieve our design goal, namely to develop a verified, efficient automata library, which supports various types of automata while minimizing redundancy between the formalizations for the different automata types, and being easily extensible to new automata types. These techniques can be used to formalize class hierarchies in general.

Preliminaries As this paper is submitted to the Isabelle Workshop, concepts from Isabelle are used without further explanation. We also assume some familiarity of the reader with automata theory and LTL model checking. However, concepts that are essential for understanding the paper are quickly introduced where required.

Structure of the Paper The rest of this paper is structured as follows: In Section 2 we present the CAVA Automata Library, where Section 2.1 describes the provided classes, Section 2.2 describes the concepts defined on these classes, Section 2.3 describes the provided algorithms, and Section 2.4 describes the implementations. Finally, Section 3 gives the conclusions and a description of related work.

2 The CAVA Automata Library

In this section we describe the structure of the CAVA Automata Library and its formalization in Isabelle/HOL. We use an object oriented design that provides a hierarchy of classes on which certain concepts and algorithms are defined.

2.1 Classes

Figure 2 shows the classes of the CAVA Automata Library. Each class inherits the fields and invariants from its superclasses, and may add new fields and invariants.

The class *fr_graph* models a *finitely reachable graph* (FRG, for short). It consists of a set of nodes *V*, a set of root nodes *V0*, and a set of edges *E*, which

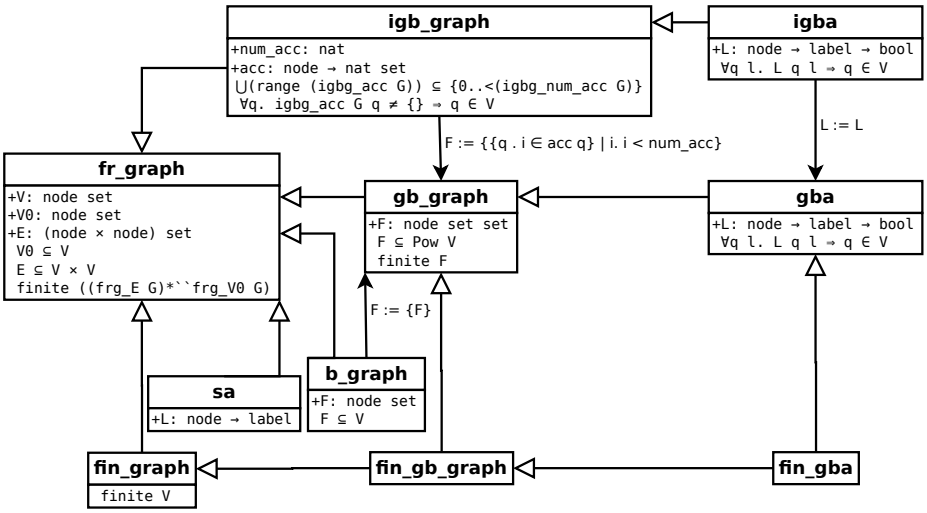

Fig. 2: Structure of the CAVA Automata Library

are modeled as pairs of nodes. The invariant ensures that the root nodes are actually nodes, that edges are only between nodes, and that there are only finitely many reachable nodes.

In order to formalize the class hierarchy, we use records for the fields, and locales to express the invariants:

type_synonym *'v digraph = ('v×'v) set*

record *'v fr_graph_rec =*
frg_V :: 'v set
frg_E :: 'v digraph
frg_V0 :: 'v set

locale *fr_graph* = **fixes** *G :: ('v,'more) fr_graph_rec_scheme*
assumes *finite_reachableE_V0[simp, intro!]: finite ((frg_E G)*``frg_V0 G)*
assumes *V0_ss: frg_V0 G ⊆ (frg_V G)*
assumes *E_ss: frg_E G ⊆ (frg_V G)×(frg_V G)*
begin
abbreviation *V ≡ frg_V G*
abbreviation *E ≡ frg_E G*
abbreviation *V0 ≡ frg_V0 G*

Inside the locale, we define suitable abbreviations for the fields. This makes definitions and lemmas inside the locale more readable.

Note that there is currently no record or locale for *'v digraph*. This is mainly due to historical reasons, as formalizing a single value as a record with a single field seems overkill without having later extension to a whole class hierarchy

in mind. Moreover, there is no class for a directed graph with an explicit set of nodes. This, as well as other obvious omissions and shortcuts in the class hierarchy, is because those classes have not yet been needed for CAVA. However, thanks to the extensible design of the CAVA Automata Library, the effort for adding those additional classes is expected to be rather low.

The class *gb_graph* extends FRGs to *generalized Büchi graphs* (GBGs), by adding a set F of sets of accepting nodes. We again define a record that inherits its fields from the base class record. Also the locale includes the locale for the base class:

record *'Q gb_graph_rec = 'Q fr_graph_rec +*
  gbg_F :: 'Q set set

locale *gb_graph = fr_graph G*
  for *G :: ('Q,'more) gb_graph_rec_scheme +*
  assumes *finite_F[simp, intro!]: finite (gbg_F G)*
  assumes *F_ss: gbg_F G ⊆ Pow V*
begin
  abbreviation *F ≡ gbg_F G*

Similarly, we formalize the class *gba*, which derives state labeled *generalized Büchi automata* (GBAs) from GBGs, by adding a labeling predicate that assigns labels to states.

Each of the three classes described above also comes with a finite version, which restricts the set of nodes to be finite. Although most properties of a finitely reachable graph are preserved when restricting it to the set of its reachable nodes, this makes a subtle difference from an implementation point of view, as discussed in Section 1. We use multiple inheritance for the finite graph classes. As only one of the superclasses contributes new fields, this can be elegantly formalized by locale expressions, without the need to define new records:

locale *fin_fr_graph = fr_graph G*
  for *G :: ('v,'more) fr_graph_rec_scheme*
+ **assumes** *finite_V[simp, intro!]: finite V*

locale *fin_gb_graph = gb_graph G + fin_fr_graph G*
  for *G :: ('Q,'more) gb_graph_rec_scheme*

locale *fin_gba = gba G + fin_gb_graph G*
  for *G :: ('Q,'L,'more) gba_rec_scheme*

Another method to represent the set of sets of accepting nodes of a GBG is to assign each state to a set of *acceptance classes*, which are represented by natural numbers. This representation is particularly useful for implementing the degeneralization and the SCC-based emptiness check algorithms. The corresponding classes are *indexed generalized Büchi graphs* (*igb_graph*) and *indexed generalized Büchi automata* (*igba*). From a refinement point of view, they can be seen as implementations of the non-indexed versions. In the diagram, this is

denoted by an arrow annotated with the abstraction function, which transforms an indexed graph to the corresponding non-indexed one. In the formalization, this correspondence is expressed by a sublocale relationship:

context *igb_graph* **begin**
  definition *accn i* ≡ {*q* . *i*∈*acc q*}
  definition *F* ≡ { *accn i* | *i*. *i*<*num_acc* }

  definition *to_gbg_ext m*
    ≡ (| *frg_V* = *V*, *frg_E* = *E*, *frg_V0* = *V0*, *gbg_F* = *F*, . . . =*m* |)

  sublocale *gbg!: gb_graph* (*to_gbg_ext m*)

The class *sa* models *system automata*, which are node-labeled finitely reachable graphs. In CAVA, they are used to represent the Kripke structure generated from the model to be checked.

Finally, the class *b_graph* models *Büchi graphs*, which are FRGs with a set of accepting nodes. They can be seen as generalized Büchi graphs with a single acceptance set, which is, again, expressed by a sublocale relationship.

2.2 Concepts

On the above classes, the CAVA Automata Library defines some basic concepts, and provides useful theorems about them.

Edge Relations On edge relations, it defines *finite and infinite paths* and *strongly connected components*:

inductive *path :: 'v digraph* ⇒ *'v* ⇒ *'v list* ⇒ *'v* ⇒ *bool* **for** *E* **where**
  path0: path E u [] *u*
| *path_prepend:* ⟦ (*u*,*v*)∈*E*; *path E v l w* ⟧ ⟹ *path E u* (*u*#*l*) *w*

definition *ipath :: 'q digraph* ⇒ *'q word* ⇒ *bool*
  where *ipath E r* ≡ ∀*i*. (*r i*, *r* (*Suc i*))∈*E*

definition *is_scc :: 'q digraph* ⇒ *'q set* ⇒ *bool*
  where *is_scc E U* ⟷ *U* ∈ *UNIV* // (*E** ∩ (*E*$^{-1}$)*)

A finite path is modeled by its start and end node, and the list of intermediate nodes, including the start node but excluding the end node. This formalization allows convenient theorems for concatenation and splitting of paths, for example:

lemma *path_conc_conv:*
  path E u (*la*@*lb*) *w* ⟷ (∃*v*. *path E u la v* ∧ *path E v lb w*)

An infinite path is modeled as an infinite sequence of nodes, represented by a function from natural numbers to nodes[1] (*'a word* is a synonym for *nat* ⇒ *'a*). The start node of an infinite path *r* is *r 0*.

[1] The formalization of ω-words is by Stephan Merz.

A strongly connected component (SCC) is a maximal set of mutually connected nodes. The SCCs can elegantly be defined as the equivalence classes of the ,,mutually connected" relation. However, we provide a bunch of equivalent characterizations, which are useful for special purposes.

Finitely Reachable Graphs On finitely reachable graphs, an *infinite run* is defined as an infinite path that starts at a root node:

context *fr_graph* **begin**
  definition *is_run* $r \equiv r\ 0 \in V0 \wedge$ *ipath* $E\ r$

Functions that are clearly related to a single class are defined inside the locale of that class. They roughly correspond to methods of the class. Note that we use the abbreviations declared earlier to conveniently refer to the fields of the class. Moreover, all lemmas proved inside the locale implicitly contain the locale's assumptions. For example, we state that a run only contains finitely many nodes as follows, implicitly using the finitely reachable assumption from the locale:

lemma *is_run_finite:* *is_run* $r \Longrightarrow$ *finite* (*range* r)

Generalized Büchi Graphs On generalized Büchi graphs, we define an *accepting run* as a run that visits states from each acceptance set infinitely often:

context *gb_graph* **begin**
  definition *is_acc_run* $r \equiv$ *is_run* $r \wedge (\forall A{\in}F.\ \exists_\infty i.\ r\ i \in A)$

Note how this definition exploits the subtyping provided by the record mechanism. The *is_run* predicate comes from the *fr_graph* locale, where it has been defined for FRGs. However, here it can also be used with GBGs. The ability to easily reuse definitions and lemmas from a superclass in the subclass is an important mechanism to keep redundancy of the formalization low.

We also prove an alternative characterization of accepting runs: A run is accepting iff the set of states that it visits infinitely often contains states from each acceptance set:

lemma *is_acc_run_limit_alt:*
  is_acc_run $r \longleftrightarrow$ *is_run* $r \wedge (\forall A{\in}F.$ *limit* $r \cap A \neq \{\})$

Note that this characterization depends on the finitely reachable assumption, which is imported from the superclass.

Generalized Büchi Automata On generalized Büchi automata, we define *accepted words* as words that are related to an accepting run via the labeling predicate. Moreover, the *language* is defined as the set of all accepted words. The definitions are done in a similar way, reusing the definitions from the superclass.

2.3 Algorithms

Beyond defining concepts like run or language on a single class, the CAVA Automata Library also provides algorithms that typically incorporate multiple objects.

Renaming A basic concept on graphs is that of renaming the nodes: Given an injection f on the nodes, compute a new graph that is isomorphic to the original graph w. r. t. f. Renaming is first defined on edge relations and finitely reachable graphs:

abbreviation *rename_E f E* ≡ (λ(*u*,*v*). (*f u*, *f v*))‘*E*

definition *fr_rename_ext ecnv f G* ≡ (|
  frg_V = *f*‘(*frg_V G*),
  frg_E = *rename_E f* (*frg_E G*),
  frg_V0 = (*f*‘*frg_V0 G*),
  ... = *ecnv G*
|)

locale *fr_rename_precond*
  = *fr_graph G* **for** *G* :: (*'u*,*'more*) *fr_graph_rec_scheme* +
  fixes *f* :: *'u* ⇒ *'v*
  fixes *ecnv* :: (*'u*, *'more*) *fr_graph_rec_scheme* ⇒ *'more'*
  assumes *INJ: inj_on f V*
begin
  abbreviation *G'* ≡ *fr_rename_ext ecnv f G*
  sublocale *G'!: fr_graph G'*

We first define the renaming function. Apart from the injection *f* and the graph *G*, it gets an additional parameter *ecnv*, which describes how the extension field (...) of the record is transformed. The extension field contains the extensions made by subclasses. This additional parameter will help us to extend renaming to subclasses, reusing the original definition and lemmas. Next, we define a locale that summarizes the preconditions of the renaming, i. e. that f is actually an injection. Inside the precondition locale, we use a sublocale relationship to express the fact that the renamed graph is also a finitely reachable graph. This makes all fields and concepts of the new graph readily accessible, with the prefix *G'*. For example, we state that runs of the original and the renamed graph simulate each other as follows, where *fi* is the inverse of *f*:

  lemma *run_sim1: is_run r* ⟹ *G'.is_run* (*f o r*)
  lemma *run_sim2: G'.is_run r* ⟹ *is_run* (*fi o r*)

Next, we extend renaming to generalized Büchi graphs:

definition *gb_rename_ecnv ecnv f G* ≡ (|
  gbg_F = { *f*‘*A* | *A*. *A*∈*gbg_F G* }, ... = *ecnv G*
|)
abbreviation *gb_rename_ext ecnv f* ≡ *fr_rename_ext* (*gb_rename_ecnv ecnv f*) *f*

locale *gb_rename_precond* = *gb_graph G* +
  fr_rename_precond G f (*gb_rename_ecnv ecnv f*)
  for *G* :: (*'u*,*'more*) *gb_graph_rec_scheme*

and *f :: 'u ⇒ 'v* **and** *ecnv*
begin
sublocale *G'!: gb_graph G'* **proof** *unfold_locales*
(∗ *// Goals for assumptions of fr_graph already discharged* ∗)

We first define the effect of renaming on the record extension for GBGs, again adding an extra parameter for further subclass extensions. Then, we define a precondition locale that includes both the locale for GBGs and the original precondition locale for renaming FRGs. Inside this locale, we extend the sublocale relationship for the renamed graph to GBGs. In the proof, we only have to discharge the goals for the assumptions added by *gb_graph*, the goals from *fr_graph* are already discharged by Isabelle/HOL's locale mechanism, because we have included the *fr_rename_precond* locale. Inside the new locale, we prove simulation of accepting runs:

lemma *acc_run_sim1: is_acc_run r ⟹ G'.is_acc_run (f o r)*
using *run_sim1 ...*
lemma *acc_run_sim2: G'.is_acc_run r ⟹ is_acc_run (fi o r)*

For the proofs, we reuse the simulation between runs that we have proved for the renaming of FRGs. This reduces the proof to showing that the acceptance condition is preserved under renaming.

In a similar way, we extend renaming to generalized Büchi automata, and show that it preserves the language.

Synchronous Product An important operation of an LTL model checker is to compute the synchronous product of the Kripke structure generated from the model and the indexed GBA generated from the LTL formula. The result is an indexed generalized Büchi graph, whose states are the product of the GBA's and SA's states. We prove that the product is a valid GBG, and that its accepting runs correspond to words in the language of both the SA and the GBA.

For CAVA, we currently define the product construction directly between indexed GBAs and system automata, without defining products between the superclasses, or providing an extension possibility to subclasses. The rest of the approach is, however, similar to renaming: We define a precondition locale, include the indexed GBG locale for the product via a sublocale relationship, and prove the properties inside the precondition locale.

Degeneralization Another algorithm is degeneralization, i. e. to convert a generalized Büchi graph to a standard Büchi graph. The standard construction, which we have formalized, is to create a copy of the GBG for each acceptance class, and link the copies such that the automaton switches to the copy for the next acceptance class when visiting a node that is in the current acceptance class. We have formalized this algorithm on indexed GBGs, and proved that accepting runs of the degeneralized graph correspond to accepting runs of the original graph.

From a formalization point of view, this algorithm falls into a special class: It has no preconditions apart from its input being a valid GBG. Thus, its

precondition locale would collapse to be a copy of the GBG locale. Instead of defining a precondition locale, we formalize degeneralization directly inside the GBG locale:

context *igb_graph* **begin**
  definition *degeneralize_ext* :: _ ⇒ ('Q × nat, _) *b_graph_rec_scheme*
  sublocale *degen!: b_graph degeneralize_ext ecnv*
  lemma *degen_acc_run_iff:*
    is_acc_run r ⟷ (∃*r'. fst o r' = r* ∧ *degen.is_acc_run T ecnv r'*)

Note that the sublocale relationship has a free parameter (*ecnv*). For that reason, all constants imported via this sublocale relationship also get an extra free parameter. In the current version of Isabelle (Isabelle-2013-2) they also get a phantom type parameter, which is, however, unnecessary and probably due to some issue in the locale package. By universally quantifying the lemmas over these additional parameters, they also apply to extensions of degeneralization to subclasses.

To-Index Conversion Any finite GBG can be converted to an indexed version, by enumerating its acceptance sets, and then converting them to acceptance classes. This algorithm is implemented for GBGs and then extended to GBAs. It is used by the conversion algorithm from LTL formulas to GBAs, which initially produces a non-indexed GBA that is then converted to an indexed GBA. We show that the sets of accepted runs and the languages of the GBA and its indexed version are the same.

To-index conversion adds another technical challenge, as it is inherently nondeterministic: There are many enumerations of the acceptance sets, and without knowing the precise implementation, one cannot decide for a specific one. Thus, it has to be formalized inside the nondeterminism monad of the Isabelle Refinement Framework [8], i. e. its return type is a set of possible results[2]:

definition *gbg_to_idx_ext* :: _ ⇒ _ ⇒ ('a, _) *igb_graph_rec_scheme nres*

This makes it impossible to define a sublocale for the result. Instead, we prove a lemma of the form

lemma (**in** *fin_gb_graph*) *gbg_to_idx_ext_correct:*
  gbg_to_idx_ext ecnv G
  ≤ (**spec** *G'. igb_graph G'* ∧ *igb_graph.is_acc_run G' = is_acc_run*)

When proving correctness of an algorithm that uses to-index conversion, the verification condition generator will generate a proof obligation of the form:

⟦*igb_graph G'; igb_graph.is_acc_run G' = is_acc_run*⟧ ⟹ ...

When discharging such a proof obligation, we can interpret the *igb_graph* locale locally in the proof, making available all concepts from *igb_graph* for *G'*.

[2] Actually, the inner type of the nondeterminism monad has the form *'a nres = RES 'a set* | *FAIL*, where the value *FAIL* represents failed assertions or nontermination.

2.4 Implementations

In CAVA, we use the Automatic Refinement Framework (Autoref) [7] to refine abstract algorithms to efficient implementations, using the efficient data structures for standard types provided by the Isabelle Collection Framework (ICF) [8]. Autoref is based on the idea of parametricity [13,14] to express refinement: Each concrete type comes with a relator, which transforms a relation on the arguments to a relation between the concrete and abstract type. For example, the relator

list_set_rel :: (*'a* × *'b*) *set* ⇒ (*'a list* × *'b set*) *set*

transforms a relation on elements to a relation between (distinct) lists and sets.

For each operation, a refinement theorem is provided, which relates the concrete with the abstract operation. For example, for sets implemented by distinct lists, we have:

([],{}) ∈ ⟨*R*⟩*list_set_rel*
⟦ (*eq*,*op* =) ∈ *R* → *R* → *bool_rel* ⟧
  ⟹ (*glist_insert eq*, *insert*) ∈ *R* → ⟨*R*⟩*list_set_rel* → ⟨*R*⟩*list_set_rel*

Note that we write the application of relator arguments in prefix form with ⟨⟩, and use → for the natural relator on functions. This has technical reasons, and, additionally, makes relators more look like type constructors, which matches their intuition.

For the CAVA Automata Library, we first define data structures for the classes together with suitable relators, and show refinement theorems for the basic operations, namely field access and constructors. Then, we use Autoref to generate implementations of the algorithms.

On-the-fly Model Checking An important requirement that influences the choice of data structures is the fact that CAVA is an on-the-fly model checker: The state space of the model is constructed lazily while searching for a counterexample. When a counterexample is found, the model checker terminates without having constructed the complete state space.

Edge Relations The edges in CAVA are implemented by a successor function, which maps a node to a distinct list of its successors. This implementation is suited for a lazy exploration of the state space, as the successors of a state can be computed on request. Using a list to represent the set of successors is also adequate, as the only operation on successors is to iterate over them, which can be implemented efficiently by a fold operation. Thus, we make the following definitions, and set up the Autoref framework accordingly:

type_synonym *'a slg = 'a ⇒ 'a list*
definition *slg_rel* :: (*'a*×*'b*) *set* ⇒ (*'a slg* × *'b digraph*) *set* **where** ⟨*R*⟩*slg_rel*
  = (*R* → ⟨*R*⟩*list_set_rel*) *O br* (λ*succs*. {(*u*,*v*). *v*∈*succs u*}) (λ_. *True*)
lemma (λ*succs v*. *succs v*, λ*E v*. *E*‘‘{*v*}) ∈ ⟨*R*⟩*slg_rel* → *R* → ⟨*R*⟩*list_set_rel*

Finitely Reachable Graphs For finitely reachable graphs, we represent the set of root nodes by a distinct list. This choice is appropriate, as the only required operation is, again, iteration. However, the adequate representation of the set of nodes depends on where in CAVA the graph is used: For the system automata, the set of nodes is the universal set of the state type. We can implement this by a degenerate set implementation, that is only able to represent the universal set[3]. On the other hand, when implementing renaming, we need to iterate over the set of nodes, which requires it to be finite. For this purpose, we use distinct lists again.

Technically, we face two problems when formalizing implementations of finitely reachable graphs: First, we want to reduce redundancy between the different implementations of the node set. Second, we want to allow for an easy extension of the implementation to subclasses.

The first problem is solved by defining a generic implementation, which is parameterized with implementations for the fields. The second problem is solved by also including a parameter for the implementation of the record extension field:

record (*'vi,'ei,'v0i*) *gen_frg_impl* =
  frgi_V :: 'vi
  frgi_E :: 'ei
  frgi_V0 :: 'v0i

definition $\langle Rm,Rv,Re,Rv0\rangle$ *gen_frg_impl_rel_ext* $\equiv \ldots$

lemma *gen_frg_refine:*
  (*frgi_V,frg_V*) $\in \langle Rm,Rv,Re,Rv0\rangle$ *gen_frg_impl_rel_ext* $\rightarrow$ *Rv*
  (*frgi_E,frg_E*) $\in \langle Rm,Rv,Re,Rv0\rangle$ *gen_frg_impl_rel_ext* $\rightarrow$ *Re*
  (*frgi_V0,frg_V0*) $\in \langle Rm,Rv,Re,Rv0\rangle$ *gen_frg_impl_rel_ext* $\rightarrow$ *Rv0*
  (*gen_frg_impl_ext, fr_graph_rec_ext*)
    $\in$ *Rv* $\rightarrow$ *Re* $\rightarrow$ *Rv0* $\rightarrow$ *Rm* $\rightarrow \langle Rm,Rv,Re,Rv0\rangle$ *gen_frg_impl_rel_ext*

Here, the record *gen_frg_impl* is parameterized with the types of the implementations, and the relator *gen_frg_impl_rel_ext* is parameterized with relations for the actual implementations, including a relation *Rm* for the implementation of the record extension field (internally called *more*). The refinement lemma states refinement of the fields and the record constructors. From this generic implementation, the concrete implementations are derived by instantiation.

Generalized Büchi Graphs For GBGs, we implement the set of acceptance sets by a list of lists. This is appropriate, as the non-indexed GBGs occurring in CAVA are generally small, and converted to indexed GBGs before the time critical emptiness check phase.

Technically, we only have to formalize a relation for the record extension, and can reuse the formalization for the base class implementation. Again, we

[3] Currently, we use an implementation by characteristic functions, which is also suitable and, unlike the degenerate implementation, already provided by the ICF.

provide an extra parameter for the record extension field, allowing reuse for further subclasses:

record $('vi,'ei,'v0i,'fi)\ gen\_gbg\_impl = ('vi,'ei,'v0i)\ gen\_frg\_impl +$
$gbgi\_F :: 'fi$

definition $\langle Rm,Rf\rangle gen\_gbg\_impl\_rel\_eext \equiv \ldots$

abbreviation $\langle Rm,Rf\rangle gen\_gbg\_impl\_rel\_ext$
$\equiv \langle\langle Rm,Rf\rangle gen\_gbg\_impl\_rel\_eext\rangle gen\_frg\_impl\_rel\_ext$

lemma *gen_gbg_refine:*
$(gbgi\_F,gbg\_F) \in \langle Rm,Rf,Rv,Re,Rv0\rangle gen\_gbg\_impl\_rel\_ext \to Rf$
$(gen\_gbg\_impl\_ext,\ gb\_graph\_rec\_ext)$
$\in Rf \to Rm \to \langle Rm,Rf\rangle gen\_gbg\_impl\_rel\_eext$

Further Implementation Choices The implementations of the other classes are derived analogously. Here, we briefly report on the choice of data structures:

The labeling functions for Generalized Büchi Automata are implemented as functions. This choice is appropriate for model checking: The labels are sets of atomic propositions, and the labeling function of the GBA effectively checks whether the atomic propositions that hold at a state of the system satisfy some constraints from the LTL formula. Note that an explicit tabulation of the labeling function would blow up the representation size exponentially, as it would have to consider all combinations of atomic propositions.

For indexed GBGs, we represent the *acc* function, which maps a state to the set of its acceptance classes, by a function from states to bitvectors. Bitvectors provide efficient union and compare operations, as required by the SCC-based emptiness check algorithm, which incrementally builds up subsets of SCCs, and immediately returns a counterexample when all acceptance classes are covered by such a subset.

Finally, the set of accepting nodes of a Büchi graph is implemented by its characteristic function. This is suitable, as the only operation required by CAVA is membership query.

Implementation of Algorithms Once we have provided data structures for the graphs, and implemented the basic operations, i. e. field access and constructors, we can use stepwise refinement to derive implementations for the algorithms. The last refinement step — from the abstract automata classes to their implementations — is performed automatically by the Autoref tool. Like on the abstract level, for subclasses, we only have to implement the functions that generate the record extension.

3 Conclusions

In this paper we have presented the CAVA Automata Library, which is used by the fully verified LTL model checker CAVA. It uses an object oriented design, modeling the different types of graphs and automata as a class hierarchy. This design allows to eliminate redundancy by reusing theorems and definitions from superclasses inside subclasses. Moreover, it is integrated into the Automatic Refinement Framework, featuring automatic refinement of graph and automata algorithms to use efficient data structures.

The formalization techniques presented here are not limited to graphs and automata, but apply to class hierarchies in general.

3.1 Future Work

Current and future work includes stratifying and extending the class hierarchy. For example, Büchi graphs could be modeled as a subclass of generalized Büchi graphs, which would eliminate some more redundancy. Moreover, Büchi automata and various alternative implementation data structures are still missing, but will soon be required by extensions planned for CAVA. Also the current state of indexed GBGs seems a bit odd. Conceptually, they are only implementations of GBGs, e. g. refinements that should come with a relator and a set of refinement theorems.

Moreover, one could implement an Isabelle package for class hierarchies, which would significantly reduce the boilerplate code required for our current formalization.

Finally, there are alternative approaches for embedding class hierarchies into Isabelle/HOL, which need to be explored. One possibility might be the usage of *typedef* to represent the class invariants, and coercions to realize subtyping.

3.2 Related Work

There are several automata and graph libraries for Isabelle/HOL. We already reported on Tuerk's automata library that was initially used by CAVA. Moreover, there is a tree automata library [6]. While it already uses refinement techniques and an early version of the Isabelle Collection Framework[4], it has no object oriented design.

The work closest to ours is the graph library by Noschinski [12,11], which defines a hierarchy of directed graph classes, also using records and locales[5]. This library is focused on directed graphs, where it provides very general concepts like labeled and parallel arcs. This generality, which is not needed by CAVA, comes at the price of a considerable formalization overhead. Thus, we decided to

[4] In fact, the ICF was initially developed specifically for this library.

[5] Actually, many ideas have flown in both directions between our and Noschinski's library, which is pandered to by the fact that the respective authors' offices are vis-à-vis on the same floor.

reimplement the special type of directed graphs required by CAVA, which was not a big deal, but made the subsequent extensions to automata much simpler.

Records in Isabelle/HOL have been implemented for the purpose of shallowly embedding object oriented designs. Naraschewski and Wenzel [9] report on techniques to shallowly embed class hierarchies in Isabelle/HOL. Our technique is similar in that it uses extensible records. However, we additionally use locales to represent class invariants.

Brucker and Wolff [2] have developed a translation from object oriented data models to a shallow embedding in HOL. However, they mainly focus on modeling the object store and references, which is not required for our formalization.

References

1. Back, R.J.: On the correctness of refinement steps in program development. Ph.D. thesis, Department of Computer Science, University of Helsinki (1978)
2. Brucker, A.D., Wolff, B.: A package for extensible object-oriented data models with an application to IMP. In: SVV 2006, Computing Research Repository (2006)
3. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: CAV, LNCS, vol. 8044, pp. 463–478. Springer (2013)
4. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, P., Sredniawa, M. (eds.) Proc. Int. Symp. Protocol Specification, Testing, and Verification. IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman & Hall (1996)
5. Holzmann, G.J.: The Spin Model Checker — Primer and Reference Manual. Addison-Wesley (2003)
6. Lammich, P.: Tree automata. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Tree-Automata.shtml` (Dec 2009), formal proof development
7. Lammich, P.: Automatic data refinement. In: Interactive Theorem Proving, LNCS, vol. 7998, pp. 84–99. Springer Berlin Heidelberg (2013)
8. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Proc. of ITP. LNCS, vol. 7406, pp. 166–182. Springer (2012)
9. Naraschewski, W., Wenzel, M.: Object-oriented verification based on record subtyping in higher-order logic. In: Proc. of TPHOLs. vol. LNCS 1479, pp. 349–366. Springer (1998)
10. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
11. Noschinski, L.: Graph theory. Archive of Formal Proofs (Apr 2013), `http://afp.sf.net/entries/Graph_Theory.shtml`, Formal proof development
12. Noschinski, L.: A graph library for isabelle. Mathematics in Computer Science (2014), to appear
13. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523 (1983)
14. Wadler, P.: Theorems for free! In: Proc. of FPCA. pp. 347–359. ACM (1989)