The Isabelle Refinement Framework
For Verification of Large Software Systems

Peter Lammich

TU Munich lammich@in.tum.de

Abstract. This paper overviews the techniques that we use to develop verified large software systems inside the Isabelle/HOL theorem prover. It is based on our development of the fully verified efficiently executable CAVA LTL model checker.
The verification follows a stepwise refinement approach, to which we adapted standard engineering techniques such as object orientation and modularization. It is entirely conducted in the Isabelle/HOL theorem prover, which results in a high confidence correctness theorem that only depends on the small inference kernel of Isabelle/HOL.
The techniques presented in this paper cover the Isabelle Refinement Framework, which provides a formalization of refinement calculus and a tool chain which makes it conveniently usable. Moreover, we describe the Isabelle Collection Framework, which provides an extensible library of efficient verified collection data structures. We also describe the object oriented techniques used to develop the automata library below our model checker, and the modularization techniques used to separate the various components of the model checker.

1 Introduction

The objective of this paper is to give an overview of the development techniques that we use to verify large-scale software systems in the Isabelle/HOL interactive theorem prover. We present some of the engineering techniques that we used to develop the verified CAVA model checker [7], a fully-fledged efficient LTL model checker.

Our development process is based on stepwise refinement, to which we adapt standard engineering techniques for structuring large software systems, like object orientation and modularization.

Stepwise refinement is a well-known technique to verify programs. The idea is to refine an abstract specification to an efficient implementation via a series of correctness preserving refinement steps. Usually, the first refinement steps introduce the algorithmic ideas of the program, and further refinement steps then replace the abstract data types used to describe the algorithm by efficient implementations.

Stepwise refinement reduces the proof complexity by separation of concerns: Instead of one big proof that deals with both, the high-level algorithmic ideas and the implementation details, it allows for several small proofs, each focusing

on a single aspect. Our experience shows that direct correctness proofs of efficient implementations tend to get unmanageable already for medium-complex algorithms like Dijkstra's Shortest Paths.

Refinement calculus [2] formalizes stepwise refinement in a Hoare-logic like framework, based on rigorous mathematical foundations. Thus, it is well suited for a theorem prover based development.

Our main tool is the Isabelle Refinement Framework[20] (cf. Section 2), which implements a refinement calculus for shallowly embedded monadic programs. It comes with tool support, which makes it practically usable. Besides a verification condition generator, it also contains the Autoref tool [16], which can automatically refine abstract data types to efficient implementations. Suitable implementations are selected via user-adjustable heuristics.

When developing efficient algorithms, it is important to have a library of reusable general purpose data structures. The Isabelle Collection Framework [14] (Section 3) provides such a library. It is based on the concepts of interfaces, generic algorithms, and implementations. Its integration into Autoref ensures easy usability.

The CAVA model checker operates on different types of graphs and automata. To avoid redundancies, these are presented as a class diagram with inheritance. In Section 4, we give a brief overview of the CAVA Automata Library [17] and how it uses object oriented techniques inside Isabelle/HOL.

The CAVA model checker itself consists of several components, which are separately maintained and developed. In Section 5, we review the modularization techniques we use to ensure isolation between these components, and their interplay with verification.

Finally, we conclude the paper in Section 6.

Note that this paper is an overview paper, presenting results that have been detailed in [14, 20, 16, 17, 7], with some small parts of newer developments. We have tried to indicate new developments in the paper at the points where they are described. The main focus of this paper is on the refinement calculus and the associated tool chains. Further engineering techniques that helped us in developing the CAVA model checker are only briefly discussed, with references to more detailed descriptions.

1.1 Related Work

Refinement Based on Back et al.'s initial formalization in HOL [1], there are several formalizations of refinement calculus in different theorem provers (e. g. [4, 23, 3]). However, they usually focus on the meta-theory of refinement calculus, and only come with relatively small example programs that are actually verified.

For the Coq theorem prover, there is a refinement tool [5], which has been used to refine some algebraic algorithms to use efficient data structures. Also the Fiat-System [6] automatically synthesizes efficient implementations of database queries phrased in an abstract query language. Both tools resemble our Autoref-tool [16], which we use to automate canonical refinement steps.

Another implementation of data refinement is supported by the Isabelle Code Generator [9]. However, it relies on an extension of the code generator outside the logic. Moreover, it can only be used for deterministic algorithms, while many abstract algorithms of a model-checker are inherently nondeterministic.

Verification of Big Software Systems There are several verifications of big software systems, even bigger ones than the CAVA model checker. One example is the verified C compiler CompCert [21]. Here, modularization is achieved by splitting the compiler into several phases, which translate between different intermediate languages. For each translation step, bisimulation between the input and output is proved. Other important techniques include tailoring of algorithms to be verification friendly, and to use a-posteriori verification of results computed by external unverified algorithm.

Another big system that has been verified is the seL4 microkernel [12]. It uses a refinement-centric development process: First, a prototype of the kernel was implemented in Haskell. It serves both as an executable implementation that can be tested, and as a functional specification that can be reasoned about in a theorem prover. Then, an efficient C version was manually implemented, and proved to refine the Haskell prototype, which, in turn, was shown to satisfy the abstract specification. For the refinement proof between the C program and the Haskell prototype, the Autocorres tool [8] was developed. Similar to our approach, it implements a refinement calculus on shallowly embedded monadic programs. However, it features bottom-up refinement, i.e., a concrete program is abstracted, while our approach uses top-down refinement, where an abstract program is concretized.

2 Foundations of the Refinement Framework

The Isabelle Refinement Framework [20, 15] provides a refinement calculus [2] that is based on a nondeterminism monad [25]. It features a stepwise refinement based development approach, where an algorithm is first specified on an abstract level, and then refined towards an efficient implementation in possibly many correctness preserving steps.

Note that nondeterminism is essential for specifying abstract algorithms: For example, a standard textbook presentation of a workset algorithm might contain the operation ,,pick some element from the workset". However, a precise description of which element is picked is not possible until the data structure for the workset has been fixed. Thus, abstractly, one has to nondeterministically choose an element, and prove the algorithm correct for any choice.

In the remainder of this section, we briefly introduce the Isabelle Refinement Framework and its theoretical foundations.

2.1 The Refinement Monad

The Monadic Refinement Framework represents programs inside a monad over the type $'a\ nres = \mathbf{res}\ 'a\ set \mid \mathbf{fail}$. A *result* **res** X means that the program non-

deterministically returns a value from the set X, and the result **fail** means that an assertion failed. The subset ordering is lifted to results as follows:

$$\textbf{res}\ X \leq \textbf{res}\ Y \equiv X \subseteq Y \quad | \quad \_ \leq \textbf{fail} \equiv \mathit{True} \quad | \quad \_ \leq \_ \equiv \mathit{False}$$

Intuitively, $m \leq m'$ (m *refines* m') means that all possible values of m are also possible values of m'. Note that this ordering yields a complete lattice on results, with smallest element **res** $\{\}$ and greatest element **fail**. The monad operations **return** and **bind** (notation $\gg=$) are then defined as follows:

$$\textbf{return}\ x \equiv \textbf{res}\ \{x\}$$
$$\textbf{res}\ X \gg= f \equiv \mathit{Sup}\ \{f\ x \mid x{\in}X\} \quad | \quad \textbf{fail} \gg= f \equiv \textbf{fail}$$

Intuitively, **return** x is the result that contains the single value x, and $m \gg= f$ is sequential composition: Choose a value from m, and apply f to it.

As a shortcut to specify values satisfying a given predicate Φ, we define **spec** $\Phi \equiv$ **res** $\{x \mid \Phi\ x\}$. Moreover, we use a Haskell-like do-notation, and define a shortcut for assertions: **assert** $\Phi \equiv$ **if** Φ **then return** () **else fail**. Recursion is defined by a fixed point: **rec** $B\ x \equiv$ **do** {**assert** (*mono* B)*; gfp* $B\ x$}. As we use the greatest fixed point, a non-terminating recursion causes the result to be **fail**. This matches the notion of total correctness. We assert monotonicity of the function's body. Note that the standard way of defining recursion is w. r. t. a flat ordering of results, where **fail** is the top element. Thus, we require monotonicity w. r. t. both, the refinement ordering and the flat ordering, in which case the greatest fixed points coincide. Note that monotonicity w. r. t. both orderings follows by construction [13] for any program that only uses the monad combinators.

On top of the **rec** primitive, we define loop constructs like **while** and **foreach**, with an explicit state threaded through the loop.

2.2 Data Refinement

In a typical refinement based development, one also wants to refine the representation of data.

A data refinement is specified by a *refinement relation* between concrete and abstract values. In many cases, this relation is single-valued, and can be expressed by an *abstraction function* from concrete to abstract values and an invariant on concrete values. Note, however, that refinement relations typically are neither left nor right total.

A prototypical example is implementing sets by *distinct lists*, i. e. lists that contain no duplicate elements. Here, the refinement relation $\langle R \rangle \mathit{list\_set\_rel}$ relates a distinct[1] list to the set of its elements, where the elements are related by R. This relation is not left-total, as lists with duplicate elements have no abstract counterpart. This reflects the concrete data structure's invariant. Also, this relation is not right-total, as infinite sets cannot be implemented by lists.

[1] Assuming R is single-valued

Given a refinement relation R, we define the function $\Downarrow R$ to map results over the abstract type to results over the concrete type:

$$\Downarrow R\ (\mathbf{res}\ A) \equiv \mathbf{res}\ \{c \mid \exists a \in A.\ (c,a) \in R\} \quad | \quad \Downarrow R\ \mathbf{fail} \equiv \mathbf{fail}$$

Intuitively, $\Downarrow R\ m_2$ is the largest concrete result, such that all its values have abstract counterparts in m_2. Thus, $m_1 \leq \Downarrow R\ m_2$ (notation $m_1 \leq_R m_2$) states that m_1 is a refinement of m_2 w. r. t. the refinement relation R, i. e. all concrete values in m_1 correspond to abstract values in m_2.

Note that we originally [20] defined the refinement relation differently: By only including concrete elements for which all abstractions are contained in the abstract result, we made $\Downarrow_R$ an adjoint of a Galois connection, which seemed theoretically beautiful at first glance. However, with this definition, we could prove some refinement rules only for single valued relations. However, during our development of a DFS framework [19], we also required (non-single valued) projection relations to reason with ghost variables. Thus, we changed the definitions to better generalize to arbitrary relations, at the cost of loosing the Galois connection property, which required reworking some proofs.

2.3 Refinement Calculus

For each combinator of the nres-monad, we define two refinement rules. One for *specification refinement*, which proves properties of the form $m \leq \mathbf{spec}\ \Phi$, and one for *pure data refinement*, which proves properties of the form $m \leq \Downarrow R\ m'$, where m and m' have the same top-level combinator. Intuitively, a specification refinement replaces an abstract specification by an algorithmic implementation (e. g. „Some path from u to v" by a depth-first search algorithm), and a pure data refinement replaces abstract types by concrete data structures (e. g. set by distinct list). For example, the rules for **return** and $\gg\!=$ are the following:

$$\Phi\ x \Longrightarrow \mathbf{return}\ x \leq \mathbf{spec}\ \Phi$$
$$(x,\ x') \in R \Longrightarrow \mathbf{return}\ x \leq_R (\mathbf{return}\ x')$$

$$m \leq \mathbf{spec}\ (\lambda x.\ f\ x \leq \mathbf{spec}\ \Phi) \Longrightarrow m \gg\!= f \leq \mathbf{spec}\ \Phi$$
$$[\![m \leq_{R'} m';\ \textstyle\bigwedge x\ x'.\ (x,\ x') \in R' \Longrightarrow f\ x \leq_R (f'\ x')]\!] \Longrightarrow m \gg\!= f \leq_R m' \gg\!= f'$$

Consider a refinement goal of the form $m \leq_R m'$. If the programs are similar enough, i.e., they have the same structure, where m may contain an arbitrary expression at places where m' contains a **spec** and the refinement relation is Id (note that $\forall\ m.\ \Downarrow Id\ m = m$), resolution with the refinement rules leaves us with *verification conditions* over the basic operations in the program. The Isabelle Refinement Framework comes with a verification condition generator (VCG), which automates this process, and has some additional rules to tolerate certain structural changes.

2.4 Refinement Based Algorithm Development

In a typical development based on stepwise refinement, one specifies a series of programs $m_1 \geq \ldots \geq m_n$, such that m_1 has the form **assert** *pre;* **spec** *post*,

and m_n is the final implementation. In each refinement step (from m_i to m_{i+1}), some aspects of the program are refined.

Refinement is modular, i.e., one can prove refinements for parts of a program in isolation. This is important for having libraries of standard algorithms, which can be used in the program to be developed. One such example is the Isabelle Collection Framework (cf. Section 3). Also, it allows to independently develop the components of larger programs, as we illustrate in Section 5.

Example 1. Given a finite set S of sets, the following specifies a set r that contains at least one element from every non-empty set in S:

$$sel_1\ S \equiv \textbf{do}\ \{\textbf{assert}\ (\textit{finite}\ S);\ (\textbf{spec}\ r.\ \forall s \in S.\ s \neq \{\} \longrightarrow r \cap s \neq \{\})\}$$

This specification can be implemented by iteration over the outer set. In each iteration step, the result set must not shrink, and it must contain an element from the current inner set, if this is not empty.

$$\begin{array}{l} sel_2\ S \equiv \textbf{do}\ \{ \\ \quad \textbf{assert}\ (\textit{finite}\ S); \\ \quad \textbf{foreach}\ S\ (\lambda s\ r.\ \textbf{spec}\ r'.\ r' \supseteq r \wedge (s \neq \{\} \longrightarrow r' \cap s \neq \{\}))\ \{\} \\ \} \end{array}$$

Using the verification condition generator, it is straightforward to show that sel_2 is a refinement of sel_1:

$$\begin{array}{l} \textbf{lemma}\ sel_2\ S \leq sel_1\ S \\ \quad \textbf{unfolding}\ sel_2\_def\ sel_1\_def \\ \quad \textbf{by}\ (refine\_vcg\ foreach\_rule[\textbf{where}\ I{=}\lambda it\ r.\ \forall s{\in}S{-}it.\ s{\neq}\{\} \longrightarrow r{\cap}s{\neq}\{\}]) \\ \qquad auto \end{array}$$

Note that the invariant for the foreach-loop is explicitly specified here. It is parametrized over the set *it* of elements still to be iterated over, and the current state r of the loop.

Next, we want to further refine the program: In each iteration, we want to pick an arbitrary element from the current inner set, and add it to the result set. We specify the new algorithm:

$$\begin{array}{l} sel_3\ S \equiv \textbf{do}\ \{ \\ \quad \textbf{assert}\ (\textit{finite}\ S); \\ \quad \textbf{foreach}\ S\ (\lambda s\ r. \\ \qquad \textbf{if}\ s{=}\{\}\ \textbf{then return}\ r \\ \qquad \textbf{else do}\ \{ \\ \qquad\quad x{\leftarrow}\textbf{spec}\ x.\ x{\in}s; \\ \qquad\quad \textbf{return}\ (insert\ x\ r) \\ \qquad \} \\ \quad)\ \{\} \\ \} \end{array}$$

Note that only the body of the foreach-loop has changed. Using the VCG, it is straightforward to show that this algorithm refines the previous one:

lemma $sel_3\ S \leq sel_2\ S$
unfolding $sel_3\_def\ sel_2\_def$ **by** (*rule refine_IdD,refine_vcg inj_on_id*) *auto*

Now assume that finding a representative element from a set is hard. Thus, every inner set comes with an pre-computed representative. We define a refinement relation between sets of sets with representatives, and sets of sets:

definition $repr\_set\_rel \equiv \{(S',S).$
$(*1*)\ S = snd`S'$
$(*2*) \wedge (\forall(b,s)\in S'.$ **case** b **of** $None \Rightarrow s=\{\} \mid Some\ x \Rightarrow x\in s)$
$(*3*) \wedge (single\_valued\ (S'\backslash<inverse>))$
}

Proposition (1) ensures that the abstract set can be obtained from the concrete set by projecting away the representatives. Proposition (2) ensures that the attached representatives are actual representatives, where an option-type is used to have *None* as representative for the empty set. Finally, proposition (3) ensures that we do not add more than one representative for each set. This is important to ensure that a finite abstract set must be represented by a finite concrete set, over which iteration is well-defined.

Finally, we phrase the refined algorithm, and prove refinement:

definition $sel_4\ S \equiv$ **do** {
assert (*finite S*);
foreach $S\ (\lambda(b,\_)\ r.$
case b **of** $None \Rightarrow$ **return** $r \mid Some\ x \Rightarrow$ **return** $(insert\ x\ r)$
) {}
}

lemma $(S',S)\in repr\_set\_rel \Longrightarrow sel_4 x\ S' \leq sel_3\ S$
unfolding $sel_4 x\_def\ sel_3\_def$
apply (*rule refine_IdD*)
apply (*refine_rcg FOREACH_refine_rcg*[**where** α=*snd*])
[...] (* *Omitted 8 lines of standard Isabelle text to prove the VCs* *)
done

2.5 Automatic Refinement

Many refinements, which are typically performed at the end of a refinement based development, are pure data refinements, i. e. the overall structure of the program is preserved, and only some abstract types are refined to concrete data structures. Given which abstract types to refine to which concrete data structures, as well as refinement rules for the required operations, the concrete program and the refinement theorem can be automatically synthesized from the abstract program.

We have implemented such a synthesis procedure in the Autoref tool [16]. It is based on the idea to express data refinement by relators [24].

It contains various heuristics to automatically select appropriate data structures and algorithms for the types and operations in the abstract program. The most important ones are the homogeneity principle and priorities. The homogeneity principle intuitively states that the result of an operation should be implemented by the same data structure as the operands. This avoids frequent casts between different implementations, thus producing a cleaner and more predictable synthesis result. Priorities can be assigned to both, data structures and algorithms. They are used to prefer efficient data structures and algorithms over less efficient ones.

Moreover, Autoref supports instantiation of generic algorithms via recursive synthesis. A generic algorithm implements an operation in terms of other operations. For example, union of finite sets may be implemented by iterating over one set, and inserting its elements into the other. When Autoref encounters a union operation, and decides to use this generic algorithm, it will try to synthesize algorithms for iteration and insertion.

Using priorities, generic algorithms may be specialized. For example, there is a more efficient union-operation on red-black trees. Its rule has a higher priority than the generic algorithm, such that Autoref will try it first. Similarly, if one can prove that the sets to be joined are disjoint, union on distinct lists can be efficiently implemented by concatenation. This rule depends on an additional side condition, which our tool will try to prove using some standard Isabelle tactics. If the proof fails, the generic algorithm is used.

2.6 Code Generation

Once the program is refined to a deterministic program that only uses executable constructs, we have to generate actual code from it. This is done in two steps: In the first step, the program is transfered to a deterministic monad, and in the second step, it is translated to source code of an actual programming language.

Transfer to Deterministic Program The combinators of the nres-monad itself are defined using non-executable constructs. For execution, we define the dres-monad over the type $'a\ dres = dsucceed \mid dreturn\ 'a \mid dfail$.

The function $nres\_of :: 'a\ dres \Rightarrow 'a\ nres$ maps a result from the dres-monad to its corresponding result from the nres-monad. Given a deterministic program m in the nres-monad, it is straightforward to transport it to the dres-monad, i.e., automatically synthesize a program m' with $nres\_of\ m' \leq m$. Moreover, if m is tail recursive (i.e. does not contain the **rec** combinator), it can be transported to a plain HOL expression. That is, we can automatically synthesize a term m'' with **return** $m'' \leq m$.

Isabelle's Code Generator When the program is refined to the dres-monad or to a plain expression, and all functions used by the program are executable (i.e., the code generator knows how to generate code for them), the code generator of Isabelle/HOL [10] can be used to generate code in one of its supported languages,

which are currently SML, OCaml, Scala, and Haskell. Note that code generation happens outside the logic of Isabelle/HOL, and thus belongs to the trusted code base. However, there is a pen-and-paper proof of its correctness [10].

Example 2. Reconsider the program from Example 1. We want to implement the input by a distinct list of distinct lists. As retrieving a representative element from a non-empty list is simple, let's drop our last refinement step and start at program sel_3 again.

As Autoref is often applied in the last refinement step before code generation, it can combine the data refinement and the transportation to the dres-monad or plain expression. Thus, an executable version of sel_3 is generated as follows:

```
schematic_lemma sel4'_aux:
  assumes [autoref_rules]: (Si,S)∈⟨⟨Id⟩list_set_rel⟩list_set_rel
  shows (?c::?'c,sel3 S)∈?R
  unfolding sel3_def by (autoref_monadic (plain))
concrete_definition sel4' uses sel4'_aux
prepare_code_thms sel4'_def
export_code sel4' in SML
```

Here, the **assumes**-line is an annotation that the parameter S should be refined by a list of lists. The relation $?R$ for the result type is left unspecified. Autoref also decides to use a distinct list, as it knows nothing about the (polymorphic) element type, and thus cannot derive an ordering or hash function, which would be required for more efficient data structures. The (plain) option indicates to transfer to a plain function, instead of the default transfer to the dres-monad. Finally, the **concrete_definition** command extracts the concrete program from the refinement theorem and names it sel_4'. The last two lines then generate the following SML-code:

```
fun sel4' A_ si =
  Foldi.foldli si (fn _ => true)
    (fn x => fn sigma =>
      (if Autoref_Bindings_HOL.is_Nil x then sigma
       else let
              val xa = List.hd x;
            in
              Impl_List_Set.glist_insert (HOL.eq A_) xa sigma
            end))
    [];
```

The code has the same structure as the original program. The foreach-loop has been replaced by a fold function[2], and the set operations have been replaced by corresponding list operations. The extra parameter $A\_$ contains the equality operation on the polymorphic element type, which is required by the insert operation.

[2] The variant *foldli* has an additional break condition, which, however its not used here, and thus set to **fn** _ ⇒ *true*.

As the generated code lives outside the logic of Isabelle, we cannot prove that it coincides with sel_4'. However, by chaining all the refinement theorems we have obtained on our way from the specification sel_1 down to the executable version sel_4', we can prove that sel_4' is actually correct w. r. t. the specification:

$$(S',S) \in \langle\langle Id\rangle list\_set\_rel\rangle list\_set\_rel \Longrightarrow \forall s{\in}S{-}\{\{\}\}.\ set\ (sel_4'\ S') \cap s \neq \{\}$$

3 The Isabelle Collection Framework

Having a library of re-usable standard data structures greatly reduces the effort required to produce efficient implementations. In this section, we briefly describe the Isabelle Collection Framework (ICF), which provides such a library.

It is seamlessly integrated into Autoref, such that many collection data structures are readily available, without any further setup. The current ICF is a defacto reimplementation of the original framework [14], to support nested data structures (e.g. distinct lists of distinct lists), and make use of the Autoref tool to instantiate generic algorithms.

The ICF is based on the concepts of interfaces, generic algorithms, and implementations. Its main features are easy usability and extensibility, which is achieved through seamless integration into the Autoref tool: Its heuristics select appropriate data structures that the user do not even have to know about. Moreover, new interfaces, generic algorithms, and implementations can be added to the ICF easily and without changing the original code base.

3.1 Interfaces

An interface describes an abstract data type and the operations on it. The default interfaces which come with the ICF are map, set, priority queue, and list. All the interfaces come with a large set of pre-defined operations, and the setup required for Autoref to identify those operations in the abstract program.

For example, the map interface comes with an emptiness check operation, and the abstract expressions $m = Map.empty$ and $dom\ m = \{\}$ may be identified as emptiness check by Autoref.

3.2 Generic Algorithms

The ICF heavily relies on generic algorithms as a tool to avoid code duplication and allow rapid prototyping of new data structures. For example, the ICF has generic algorithms to derive most operations on (finite) maps from five basic operations: empty-map, lookup, update, remove, and fold. Moreover, it has generic algorithms to derive a set implementation from a map implementation, by instantiating the value type to unit. This allows for rapid prototyping of a new data structure, as all operations on sets and maps become available once one has implemented the five basic map operations. Moreover, in many cases

the generic algorithms are reasonably efficient and match the default implementation of the operation for this data structure. This way, code duplication is avoided, as the generic algorithm is shared between many data structures. If a data structure supports a more efficient version of an operation, specialization is used to override the generic algorithm.

3.3 Implementations

An implementation provides a concrete data structure for an interface. It consists of a refinement relation and implementations of some of the operations, along with their correctness lemmas.

Note that an implementation needs not provide all operations. Some of the operations may be filled in by generic algorithms, and others may not be supported at all. The Autoref tool will only select implementations that support all operations required by the program to be refined.

Available Implementations Examples for data structures provided by the ICF are red-black trees and hash tables for sets and maps, distinct lists for sets, association lists for maps, characteristic functions for sets, bit-vectors for (dense) sets of natural numbers, and arrays for (dense) maps from natural numbers.

While the red-black tree and list based data structures are purely functional, hash-tables, bit-vectors, and arrays are based on mutable arrays with undo-history (called DiffArray in Haskell) which behave like functional arrays, but use destructive update internally.

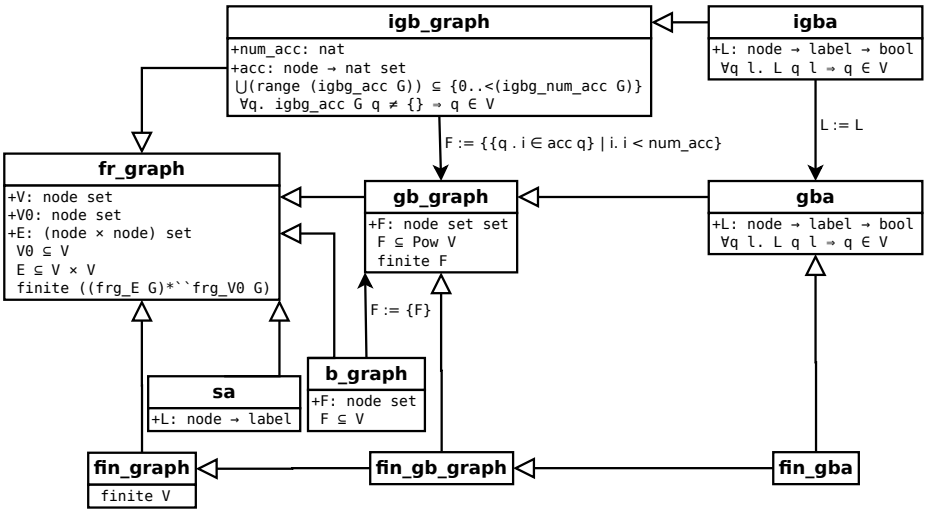
For those arrays, access to the latest version is always efficient, while access to earlier versions gets more expensive as older the accessed version is. However, many algorithms access their data in a linear fashion, and for linear access, the array-based implementations are considerably faster than purely functional implementation.

One drawback is that the mutable arrays with undo-history have to be implemented outside the logic, and thus contribute to the trusted code base. In [18], we presented an alternative approach that allows to reason about imperative features inside the logic.

4 The CAVA Automata Library

While the ICF organizes abstract types and their implementations, it has only limited support to establish a hierarchy on the interfaces: Type classes can be used to define specialized interfaces, which support additional operations: For example, the interface *ordered-set* constrains its elements to be in a linear order type class, and then provides additional operations like minimum.

When we developed the CAVA Automata Library [17], which formalizes the various graph and automata types that occur in the CAVA Model Checker, we realized that there are many redundancies between the various types, which we eliminated by structuring them in a class hierarchy:



Each class inherits the fields and invariants of its base classes, and may add new fields and invariants. Moreover, some of the classes may be specializations of other classes, as indicated by solid arrows. For example, Büchi automata can be seen as generalized Büchi automata with a single acceptance class, as indicated by the solid arrow from class *b_graph* to *gb_graph*.

Internally, classes are implemented by a mixture of locales [11] and records [22]. The records provide a mechanism to declare the fields of the classes, and exploit polymorphism to have subtyping, i.e., the type of a base class matches on the type of its subclasses. However, they are restricted to single inheritance, which was not a problem for our design[3].

Locales provide a mechanism to capture the invariants of a class. Moreover, inside a class' locale, concepts can be defined and theorems can be proven, which are inherited to the subclasses. For example, the class *fr_graph* defines the concept of a path between two nodes, and proves theorems about it. These are available in all subclasses.

Methods with static binding correspond to functions that take a parameter of a class' record type. Inside such a method, we may re-use the corresponding method from the superclass. For example, renaming the states of an automaton is implemented as first renaming the nodes of the underlying graph, and then renaming the set of accepting states.

Definition of methods with dynamic binding (i. e. virtual methods) is more tricky. We avoided this in our Automata library, and leave it to future research to evaluate the different possible approaches.

Implementation is done via the Autoref-Tool, defining the classes as abstract types, and relating them with implementations. The implementations are also

[3] Note that we actually support multiple inheritance as long as all the fields can be added by following a single path up the hierarchy.

structured via records, such that implementations of base classes may be reused to implement subclasses. For example, a *gb_graph* may be implemented by augmenting an implementation of an *fr_graph* with an acceptance set.

5 The CAVA LTL Model Checker

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 sets of atomic propositions annotated at the nodes.

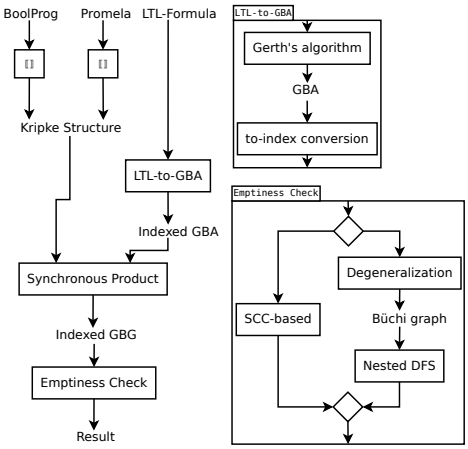


Fig. 1: Structure of the CAVA Model Checker

The LTL formula is converted to a generalized Büchi automaton, which accepts all infinite words that do *not* satisfy the formula.

Then, the synchronous product of the Kripke structure and the generalized Büchi automaton is computed, resulting in an generalized Büchi graph. Finally, the generalized Büchi graph is checked for emptiness by either using a strongly connected component algorithm, or by degeneralizing it and using 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.

The different components of CAVA are implemented and maintained by different developers. Thus, it is important to decouple them as much as possible. The interfaces between the components are based on the classes of the CAVA Automata Library.

The components are linked on two levels: the specification level and the implementation level. The specification level describes the abstract components' effect on the abstract automata data structures, using nondeterminism to leave room for different implementations. For example, the result of the intersection may be any automaton whose language is the intersection of the Büchi-automata's language with the system's runs.

The link at the implementation level is realized as generic algorithm: Given consistent implementations of the components which satisfy their specifications, a model-checker is constructed and proven correct. To obtain the actual model-

checker and correctness proof, the generic algorithm is instantiated with the actual implementations.

These are the only points where the different components of the model checker are connected. Thus, changes to the components remain local, and do not affect the rest of the system. This greatly increases the maintainability and extensibility of the system. For example, we added the SCC-based emptiness check algorithm to the system later. After formalizing and proving the new algorithm correct, we could simply replace the original emptiness check component by a dispatcher component, which selects the algorithm based on a flag.

6 Conclusion

We have presented an infrastructure to develop large-scale verified software systems. It is based on stepwise refinement, which reduces proof complexity by splitting the correctness proof into independent parts. Our verification process is done entirely inside the Isabelle/HOL theorem prover. Thus, our correctness theorems only depend on the small inference kernel of Isabelle/HOL, which gives them a very high confidence. The user of our framework is supported by a tool chain which simplifies the proving process by automating canonical tasks.

Using the fully verified CAVA LTL model checker as a case study, we have shown how to adapt standard engineering techniques like object orientation and modularization to our development process.

References

1. Back, R.J.R., von Wright, J.: Refinement concepts formalized in higher order logic. Formal Aspects of Computing 2 (1990)
2. Back, R.J., von Wright, J.: Refinement Calculus — A Systematic Introduction. Springer (1998)
3. Boulmé, S.: Intuitionistic refinement calculus. In: Typed Lambda Calculi and Applications, LNCS, vol. 4583, pp. 54–69. Springer (2007)
4. Butler, M., Långbacka, T.: Program derivation using the refinement calculator. In: TPHOLs, LNCS, vol. 1125, pp. 93–108. Springer (1996)
5. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: Gonthier, G., Norrish, M. (eds.) CPP, LNCS, vol. 8307, pp. 147–162. Springer (2013)
6. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. SIGPLAN Not. 50(1), 689–700 (Jan 2015), http://doi.acm.org/10.1145/2775051.2677006
7. 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)
8. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: Formal verification of C code without the pain. In: PLDI. pp. 429–439. ACM (jun 2014)
9. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in isabelle/hol. In: ITP, LNCS, vol. 7998, pp. 100–115. Springer (2013)

10. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Functional and Logic Programming (FLOPS 2010). LNCS, Springer (2010)
11. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales - A sectioning concept for isabelle. In: TPHOLs. pp. 149–166 (1999)
12. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proc. ACM Symp. Operating Systems Principles. pp. 207–220. ACM (2009)
13. Krauss, A.: Recursive definitions of monadic functions. In: PAR. vol. 43, pp. 1–13 (2010)
14. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: ITP. LNCS, vol. 6172, pp. 339–354. Springer (2010)
15. Lammich, P.: Refinement for monadic programs. In: Archive of Formal Proofs. http://afp.sf.net/entries/Refine_Monadic.shtml (2012), formal proof development
16. Lammich, P.: Automatic data refinement. In: Interactive Theorem Proving, LNCS, vol. 7998, pp. 84–99. Springer Berlin Heidelberg (2013)
17. Lammich, P.: The CAVA automata library. In: Isabelle Workshop (2014)
18. Lammich, P.: Refinement to imperative/hol. In: Interactive Theorem Proving, LNCS, vol. 9236, pp. 84–99. Springer (2015)
19. Lammich, P., Neumann, R.: A framework for verifying depth-first search algorithms. In: CPP. pp. 137–146 (2015)
20. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: ITP. LNCS, vol. 7406, pp. 166–182. Springer (2012)
21. Leroy, X.: A formally verified compiler back-end. J. Automated Reasoning 43, 363–446 (2009)
22. Naraschewski, W., Wenzel, M.: Object-oriented verification based on record subtyping in higher-order logic. In: TPHOLs. pp. 349–366 (1998)
23. Preoteasa, V., Back, R.J.: Invariant diagrams with data refinement. Formal Aspects of Computing 24(1), 67–95 (2012)
24. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523 (1983)
25. Wadler, P.: Comprehending monads. In: Mathematical Structures in Computer Science. pp. 61–78 (1992)