Software & system verification with KIV

Gerhard Schellhorn, Stefan Bodenmüller, Martin Bitterlich, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Schellhorn, Gerhard, Stefan Bodenmüller, Martin Bitterlich, and Wolfgang Reif. 2022. "Software & system verification with KIV." *Lecture Notes in Computer Science* 13360: 408–36. https://doi.org/10.1007/978-3-031-08166-8_20.

Software & System Verification with KIV*

Gerhard Schellhorn, Stefan Bodenmüller, Martin Bitterlich, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg, Germany
{schellhorn,stefan.bodenmueller,martin.bitterlich,reif}
@informatik.uni-augsburg.de

Abstract. This paper gives an overview of the KIV system, which in its long history has evolved from a prover for sequential programs using Dynamic Logic to a general purpose theorem prover. Today, KIV's main focus is the refinement-based development of sequential and concurrent software systems. In this paper we describe KIV's logic, highlighting recent developments such as support for polymorphism and for exceptions in programs. We show its proof engineering support that uses a graphical user interface and explicit proof trees, as well as KIV's support for the development of large-scale software systems using modular components and for the verification of concurrent algorithms using a rely-guarantee calculus. Finally, we give a short survey over the case studies that have been conducted with KIV.

1 Introduction

KIV was originally developed in the 80's by Maritta Heisel, Wolfgang Reif and Werner Stephan at the chair of Prof. Menzel in Karlsruhe [25]. The original focus was on developing proof support for the verification and synthesis of sequential programs using Dynamic Logic [17]. Underlying the work was the development of a specific functional "proof programming language" (PPL), that replaced LISP's basic data structure of s-expressions with proof trees.

Reiner Hähnle was one of the first students involved in the implementation of PPL (that used an instance of Cardelli's SECD machine [9] to interpret PPL) and the realization of first deduction concepts. He co-authored [19]. This work later influenced the design of the Key System [1], which also uses a version of Dynamic Logic to verify Java programs.

Since then KIV has evolved to a general-purpose theorem prover, however, still with a focus on developing verified software. KIV supports the refinement-based development of sequential as well as concurrent software systems. The logic has been extended to a higher-order temporal logic and recently polymorphism has been added. The programming language now has exceptions and a code generator supports generating Scala as well as C-Code.

The implementation language of KIV also changed a few years ago: KIV is now programmed entirely in Scala [43].

* Partly supported by the Deutsche Forschungsgemeinschaft (DFG), "Verifikation von Flash-Dateisystemen" (grants RE828/13-1 and RE828/13-2).

This paper gives an overview of the current concepts supported in KIV. It is organized as follows. Sec. 2 introduces polymorphic higher-order logic, which is the basis of our specification language. Exemplary specifications of the free data type of lists and the non-free data type of heaps with separation formulas are given.

Sec. 3 describes the core features of KIV's proof engineering. Explicit proof trees are used, that can be saved and manipulated, AC rewriting and heuristics are used to automate proving theorems. Interaction takes place via a graphical interface that allows context-sensitive rewriting by clicking on sub-expressions.

Sec. 4 introduces KIV's program logic for sequential, abstract programs. In contrast to many other theorem provers, which use (some variant of) higher-order logic only and embed programs as data structures, KIV always had programs as a native concept together with a calculus for symbolic execution of Dynamic Logic and wp-calculus formulas. The section then introduces KIV's concept of components, subcomponents, and refinement. Components are also called abstract state machines (ASMs) since they are close to ASMs as defined in [8]. Abstractly, a component can be viewed as an instance of an abstract data type, i.e. a collection of operations working on a common state. When operations are called sequentially, then refinement is essentially data refinement with the contract approach [12].

Section Sec. 5 introduces KIV's basic concepts for the verification of concurrent systems. This calculus has evolved over time. It started as a calculus that views programs as formulas of interval temporal logic (ITL; an extension of LTL) and is able to prove arbitrary temporal logic properties. Rely-Guarantee (RG) formulas were initially defined as abbreviations [55]. They became native concepts later on, with calculus rules that are stream-lined to the verification of partial and total correctness of concurrent programs.

KIV also implements extensions of the component concept to concurrency. Such components may either have concurrent internal threads (e.g. Garbage Collection) or offer a thread-safe interface such that operations can be called by several threads in parallel. They are proved to be linearizable [26] and deadlock-free using proof obligations from RG calculus. A discussion of concurrent components and the proof obligations that are necessary for their correctness is beyond the scope of this paper, the interested reader should look at [4, 52].

Sec. 6 gives an overview of some medium- to large-sized case studies that have been verified with KIV. Finally, Sec. 7 gives some ongoing work and concludes.

2 Basic Logic and Structured Specifications

The basic logic of the KIV system is higher order logic (HOL), recently extended from monomorphic to polymorphic types. The definition of the set of types Ty is based on a finite set of type constructors Tc with fixed arity l (we write $tc{:}l \in Tc$) and a countable set of type variables $'a \in Tv$. It is assumed that the type constant $bool$ is predefined, i.e. $bool{:}0 \in Tc$. We will usually leave the arity of type constructors implicit when writing types. Hence, a type $ty \in Ty$ is

an application of a type constructor, a type variable, a tuple type, or a n-nary function type

$$ty \;:=\; tc{:}l(\underline{ty}) \mid 'a \mid (\underline{ty}) \mid \underline{ty} \to ty'$$

where $\underline{ty}$ denotes a sequence $ty_1 \times \ldots \times ty_n$ of types (we will use underlining to denote sequences in general). The sequence must have l elements in the application of a type constructor, at least two elements for tuples, and at least one element for function types.

Expressions $e \in \mathit{Expr}$ are defined over a set of (typed) variables $x{:}ty \in X$ and a signature $\Sigma = (\mathit{Tc}, \mathit{Op})$ which in addition to type constructors contains (typed) operations $\mathtt{op}{:}ty \in \mathit{Op}$. Op always includes the usual boolean operations like $\mathtt{true} : \mathit{bool}$, $\neg\ .\ : \mathit{bool} \to \mathit{bool}$ (written prefix), or $.\ \wedge\ .\ : \mathit{bool} \times \mathit{bool} \to \mathit{bool}$ (written infix), equality $.\ =\ .\ :\ 'a \times\ 'a \to \mathit{bool}$, an if-then-else-operator $\supset$: $\mathit{bool} \times\ 'a \times\ 'a \to 'a$, as well as tuple constructors (written $(e_1, \ldots, e_n)$) and tuple selectors (written e.g. $e.\_3$). The basic set of higher-order expressions, which will be extended in Sec. 4 and Sec. 5, is defined by the grammar

$$e \;:=\; x \;\mid\; \mathit{op} \mid\; e_0(\underline{e}) \;\mid\; \lambda\, \underline{x}.\ e \;\mid\; \forall\, \underline{x}.\ \varphi \;\mid\; \exists\, \underline{x}.\ \varphi$$

Here, φ denotes a formula, i.e. an expression of type bool, and the variables of $\underline{x}$ must be pairwise disjoint. In the following we will also use t to denote terms, which are expressions without quantifiers, and ε to denote quantifier-free formulas, which are used e.g. for conditions of programs (see Sec. 4.1). The typing rules are standard, e.g. in an application the type of e_0 must be a function type, where the argument types are equal to those of $\underline{e}$. Operations are allowed to be used with an instantiated type in theorems, but not in axioms (definitions). Most types can be inferred by type inference, so we will leave the types of variables and the instance types of operations implicit in formulas. Application of the if-then-else-operator is written as $(\varphi \supset e_1; e_2)$. Its result is e_1 if φ is true and e_2 otherwise.

The semantics of an expression $[\![e]\!]$ essentially follows the semantics of HOL defined in [18]. It is based on algebras $\mathcal{A} = (\mathcal{U}, \{tc{:}l^{\mathcal{A}}\}, \{\mathtt{op}{:}ty^{\mathcal{A}}\})$.

The first component of an algebra is the universe $\mathcal{U}$, which is a set of non-empty (potential) carrier sets. The semantics $tc{:}l^{\mathcal{A}} : \mathcal{U}^l \to \mathcal{U}$ of a type constructor maps the carrier sets of its argument types to the one of the full type. The semantics of booleans, functions and tuples is standard, i.e. $\mathcal{U}$ is assumed to contain the set $\{\mathbf{tt}, \mathbf{ff}\}$ that interprets booleans, and to be closed against forming cartesian products and functions to interpret function and tuple types. Given a type valuation $w : Tv \to \mathcal{U}$, that maps each type variable to a carrier set, an algebra fixes the semantics $[\![ty]\!](\mathcal{A}, w)$ of a type as one of the carrier sets in $\mathcal{U}$.

The interpretation $\mathtt{op}{:}ty^{\mathcal{A}}$ of an operation over an algebra then yields an element of $[\![ty]\!](\mathcal{A}, w)$ for every possible type valuation w. Finally, the semantics of an expression $[\![e]\!](\mathcal{A}, w, v)$ refers to an algebra $\mathcal{A}$, to interpret type constructors and operations, a type valuation w, and a valuation v (compatible with w) that maps each variable $x : ty$ to an element of $[\![ty]\!](\mathcal{A}, w)$. The result of $[\![e]\!](\mathcal{A}, w, v)$ is an element of the carrier set $[\![ty]\!](\mathcal{A}, w)$. We write $\mathcal{A}, w, v \models \varphi$ when the

```
data specification
  using nat

data types
  list('a) = [] | . + . (. .head : 'a; . .tail : list('a)) ;

variables
  x, y, z : list('a);
  a : 'a;

size functions # . : list('a) → nat;
order predicates . < . : list('a) × list('a);

end data specification
```

Fig. 1: KIV specification of the generic free data type *list*.

semantics of a formula evaluates to **tt**. An algebra is a model of an axiom φ ($\mathcal{A} \models \varphi$), iff the formula evaluates to true for all w and v.

In the following we are interested in valuations v that are used as (the changing) states of programs, while algebra $\mathcal{A}$ and type valuation w are fixed. In this case we often drop these arguments and just write $[\![e]\!](v)$.

2.1 Structured Specifications of Algebraic Data Types

In KIV we use structured algebraic specifications to build a hierarchy of data type definitions, which may be generated freely or non-freely. Such data type definition specifications can be augmented by additional functions and can be combined using the usual structuring operations like enrichment, union, and renaming. KIV also supports a generic instantiation concept. It allows to replace an arbitrary subspecification P (the "parameter") of a generic specification G with an actual specification A using a *mapping*. A mapping is a generalized morphism that renames types and operations of P to types and expressions over A. Such an instantiation generates proof obligations that require to prove that the axioms of P instantiated by the mapping are theorems over A.

An example of a free data type specification is given in Fig. 1 for lists. A list $list('a)$ is defined using a constant constructor `[]` (representing the empty list) and a non-constant infix constructor `+`. Non-empty lists consist of an head element of generic type $'a$ and and a remaining tail list. These fields can be accessed via the selector functions `.head` and `.tail`, respectively.

For a free data type specification KIV generates all necessary axioms: the constructor functions are injective, different constructor functions yield different results, selectors and update functions (written e.g. x`.head:=` *newhead*) get definitions. Selector (and update) functions are not given axioms for all arguments: `[].head` is left unspecified (as is `[].tail`). The semantic function in a model then is still a total function, and `[].head` may be any value, following the standard loose approach to semantics. However, for use in programs, KIV attaches a *domain* to the function, here given as $\lambda\, x.\ x \neq$ `[]`. Calling `.head` outside of its domain in a program (here: with `[]`, where it is "undefined") will raise an exception, explained in detail in Sec. 4.2.

A *size function* (`#` x counts the number of non-constant constructors in x, i.e. it calculates list length) and an *order predicate* ($x < y$ iff x is a suffix of y)

can be specified, for which axioms are generated as well. Note that we abbreviate functions with result type *bool* by omitting the result type and declaring them as *predicates*.

2.2 Modelling the Heap and Separation Logic

The specification of non-free data types requires more effort as axioms cannot be generated automatically. For example, we use the polymorphic non-free data type shown in Fig. 2 to reason about pointer structures in the heap. A heap can be considered as a partial function mapping references r to objects obj of a generic type $'a$, where allocation of references is explicit. As stated by the "induction" clause, the $heap('a)$ data type is inductively generated by the constant $\emptyset$ representing the empty heap, by allocating an new reference r (written h `++` r), or by updating an allocated location r with a new object obj (written $h[r, obj]$).

For a non-free data type one has to give an *extensionality axiom*: two heaps are considered to be equal if they have allocated the same locations and they store the same objects under their allocated locations. This definition requires two additional operations that also have to be axiomatized. A predicate $r \in h$ is defined for checking whether a reference is allocated in a heap and a function $h[r]$ is used for looking up objects in the heap (this corresponds to dereferencing a pointer). References can also be deallocated by the function h `--` r.

```
generic specification
   parameter ref
   using nat
target

sorts heap('a);

constants ∅ : heap('a);

predicates
    . ∈ . : ref × heap('a);

variables
    h : heap('a);
    r : ref;
    obj : 'a;

partial functions
    . ++ .  : heap('a) × ref → heap('a)
              with λ h, r. r ≠ null ∧ ¬ r ∈ h;
    . -- .  : heap('a) × ref → heap('a)
              with λ h, r. r ≠ null ∧ r ∈ h;
    . [ . ] : heap('a) × ref → 'a
              with λ h, r. r ≠ null ∧ r ∈ h;
    . [ . ] : heap('a) × ref × 'a → heap('a)
              with λ h, r, obj. r ≠ null ∧ r ∈ h;

induction
    heap('a) generated by ∅, ++, [ ];

axioms

extensionality:
⊢ h0 = h1 ↔ ∀ r.   (r ∈ h0 ↔ r ∈ h1)
                 ∧ (r ∈ h0 → h0[r] = h1[r]);

;; ... definitions of ∈, [ ], -- ...

end generic specification
```

Fig. 2: KIV specification of the polymorphic non-free data type $heap('a)$.

The constructor functions as well as lookup and deallocation are declared as partial functions in order to specify valid accesses to the heap. This requires to give domains for the functions (see lambda clauses in Fig. 2): Accesses to the heap with the `null` reference are always undefined ($r \neq$ `null`) and allocation is only allowed for new references ($\neg\ r \in h$). Lookups, updates, and deallocations are defined only for allocated references ($r \in h$).

This explicit specification of the heap is necessary since in KIV all parameters of procedures are explicit. Hence, when reasoning about pointer-based programs, like a pointer-based implementation of red-black trees, the heap must be an explicit parameter of the program as well. To facilitate the verification of such programs we built a library for Separation Logic (SL) [48] in KIV. SL formu-

las are encoded using heap predicates $hP : heap('a) \to bool$. A heap predicate describes the structure of a heap h. At its simplest, h is the empty heap `emp`:

$$\vdash \texttt{emp}(h) \leftrightarrow h = \emptyset$$

The maplet $r \mapsto obj$ describes a singleton heap, containing only one reference r mapping to an object obj. It is defined as a higher-order function of type $(ref \times 'a) \to heap('a) \to bool$:

$$\vdash (r \mapsto obj)(h) \leftrightarrow h = (\emptyset \texttt{ ++ } r)[r, obj] \wedge r \neq \texttt{null}$$

More complex heaps can be described using the separating conjunction $hP_0 * hP_1$ asserting that the heap consists of two disjoint parts, one satisfying hP_0 and one satisfying hP_1, respectively. Since it connects two heap predicates, it is defined as a function with type $(heap('a) \to bool) \times (heap('a) \to bool) \to (heap('a) \to bool)$:

$$\vdash (hP_0 * hP_1)(h) \leftrightarrow \exists\, h_0, h_1.\ h_0 \perp h_1 \wedge h = h_0 \cup h_1 \wedge hP_0(h_0) \wedge hP_1(h_1)$$

Besides the basic SL definitions, the KIV library contains various abstractions of commonly used pointer data structures like singly-/doubly-linked list or binary trees. These abstractions allow to prove the functional correctness (incl. memory safety) of algorithms on pointer structures against their algebraic counterparts.

3 Proof Engineering

Proof engineering (in analogy to software engineering) is the process of developing a verified software system. Since the goal is mechanized verification, the tool support a verification system can provide for speeding up the process is very important. The process includes various complex and time-consuming tasks. Structured specifications containing axioms and definitions must be set up or (better) reused from a library, properties must be formalized, and finally proved. Many revisions are necessary when proofs fail or definitions are found to be inadequate. Only a small part of the effort is verifying the final theorems with the correct axioms, where flexible features to automate proofs are crucial to avoid repeating the same interactive proof steps over and over. Most of the effort is spent restructuring specifications, revising axioms and theorems, and then particularly correcting proofs that become invalidated by these changes, so that maintaining a large lemma base is a critical factor in developing a verified system.

One key building block of KIV to address these challenges is the Graphical User Interface (GUI), which provides intuitive and interactive support for the proof engineer. Algebraic specifications and theorems can be managed via a graphical representation of the specification hierarchy (see Sec. 3.1). Theorems are proved semi-interactively in the GUI, where proof automation techniques like heuristics or the automatic rewriting of expressions support the user (see Sec. 3.2). Proofs are visualized explicitly as *proof trees* that give insight into every proof step and offer direct manipulation of the proof (see Sec. 3.3).

The KIV system is publicly available as an IDE plugin developed in Scala. More information about the setup can be found at [28].

3.1 Management of Specifications and Proofs

The basis of the development is a hierarchy of specifications that can contain data type definitions, components (see Sec. 4), and structuring operations (union, actualization, enrichment, etc.). This hierarchy is called the *development graph* and shown graphically. It is the starting point when developing a software project in KIV. A specification (a node in the graph) has a *theorem base* attached that contains axioms and theorems (some of them may be proof obligations) together with their proofs, as far as they have been done.

A specification can be in different *states*, depending on whether it has just been *created*, is *imported* from a library, or is currently *valid*. Changing a subspecification may yield an *invalid* specification that must be fixed, e.g. when some signature symbols used in axioms has changed. When all theorems over a specification have been successfully proved, it can be put in *proved state*, which asserts that all lemmas used in proofs that are from subspecifications are still present in *some* subspecification. Deferring this check for entering proved state allows to avoid invalidating proofs on restructuring specifications. It also avoids the need to enforce a bottom-up strategy for proving theorems. In general, the *correctness management* of KIV always minimizes the number of proofs that are invalidated when specifications or theorems are changed. When all specifications are in proved state, the correctness management guarantees that all theorems are proved and do not have cyclic dependencies.

The development graph also offers access to work on a particular specification. When selecting a specification, the theorem base is loaded and an overview is shown that gives information about its axioms, and its proved, unproved, and invalid lemmas. At this level, theorems can be added, edited, and deleted. Axioms and theorems can be declared as *simplifier rules*, which are then used automatically in proofs (see the following section). For the theorems of the specification, new proofs can be started, existing proofs can be deleted, and partial proofs can be continued. KIV projects can be combined by importing (parts of) other projects and using them as a library. The standard library included in the KIV plugin provides specifications for common data structures like lists, arrays, sets, and maps. Other libraries define support for separation logic (see Sec. 2.2), or provide basic locking constructs (mutexes, reader-writer locks, and conditions) to support concurrency. Providing an exhaustive library also has the benefit of adding many useful simplifier rules to the theorem base, such that verification effort will be reduced drastically.

Finally, the KIV application offers general features, like exporting projects to an HTML representation (presentations of many projects can be found at [30]) or viewing proof statistics for projects or individual specifications.

3.2 Proving Theorems

Proving theorems in KIV is done using a sequent-based calculus. A sequent $\Gamma \vdash \Delta$ abbreviates the formula $\forall \underline{x}.\ \bigwedge \Gamma \rightarrow \bigvee \Delta$ where Γ (the antecedent) and Δ (the succedent) are lists of formulas and $\underline{x}$ is the list of all free variables in

Δ and Γ. The rules of the calculus follow the structure of the formulas and are applied backwards in order to reduce the conclusion to simpler premises, until they can be closed using axioms.

KIV facilitates this process for proving theorems by using extensive automation combined with interactive steps performed by a proof engineer. Therefore the interface must help the proof engineer to understand the automated steps and to identify possible actions. The interface for proving consists of two parts: one for the proof tree (see Sec. 3.3) and another for performing proof interactions. The latter is the most important part of the GUI and is shown in Fig. 3. The large area in the middle contains the sequent of the current goal, i.e. what currently has to be proven. The left-hand side shows the list of calculus rules that can be applied to the goal. A status bar under the menu bar provides information about the proof: the name of the lemma, the number of open goals, the number of the current goal, and the total number of proof steps.

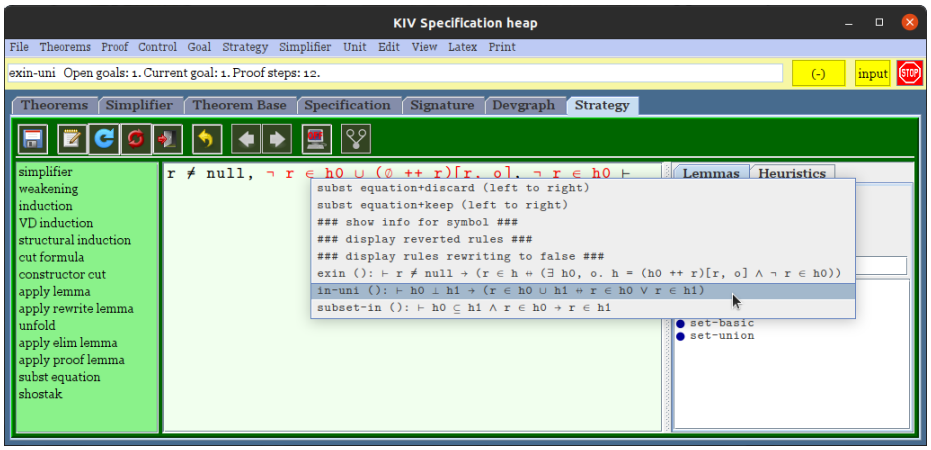

Fig. 3: Sequent of a proof goal with context sensitive choice.

The proof engineer must choose the next proof step, if no rules are applicable by the automation. This is usually done context-sensitively, as shown in Figure 3. When the user moves the mouse over the sequent, the KIV system permanently determines the smallest (sub-)expression and highlights it. Right-clicking on a formula opens a context menu with applicable rules. If the leading symbol is a logical connective, the rules for dealing with this connective are offered, e.g. "quantifier instantiation" for quantifiers. If the leading symbol is a predicate, a list of applicable rules is shown.

Figure 3 shows the result of a click on the predicate $\in$ in the expression $\neg\, r \in h0 \cup (\emptyset$ ++ $r)[r, obj]$ in the antecedent. It shows the three rewrite lemmas matching on the selected part of the expression out of the hundreds of lemmas in the theorem base. When the user selects the rewrite lemma *in-uni*: $\vdash h0 \perp h1 \rightarrow (r \in h0 \cup h1 \leftrightarrow r \in h0 \vee r \in h1)$ from the context menu, all occurrences of the selected expression are rewritten to $\neg r \in h0 \vee r \in (\emptyset$ ++ $r)[r, obj]$. The precondition generates a side goal where one has to prove $h0 \perp (\emptyset$ ++ $r)[r, obj]$. The context-

sensitive computation of applicable lemmas restricts the theorem base to the relevant cases, so that even very large verification projects remain manageable.

Besides applying rules manually, KIV offers different levels of automation: the user can switch between different sets of heuristics that automatically apply calculus rules. The base heuristic that is used in every proof is the *simplifier* The simplifier rewrites the current goal by applying rules and lemmas that simplify the sequent. This includes applying all the propositional rules of sequent calculus with at most one premise as well as the two rules that eliminate quantifiers (universal quantifier right and existential quantifier left). The simplifier also applies simplifier rules that are lemmas following the pattern of *conditional rewrite rules* and of *forward rules* if enabled by the user: Rewrite rules replace terms with simpler terms, forward rules are typically used to add transitive information to the goal. Application of these rules is performed modulo associativity and commutativity, the strategy is currently enhanced to include matches modulo neutral elements as well. Additionally, specialized heuristics are available, depending on the content of the sequent. E.g., for sequents containing programs (see Sec. 4), different heuristics performing *symbolic execution* can be selected. Furthermore, KIV offers heuristics that call the SMT solvers Z3 [10] and CVC4 [2].

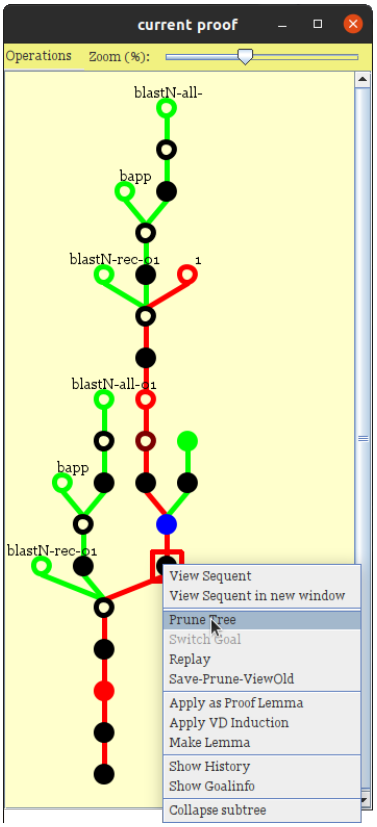

Fig. 4: Example of a proof tree.

3.3 Proof Trees

A proof tree like the one shown in Figure 4 is a graphical representation that shows the applied rules in a particular proof step. Typically a proof tree consists of 20 up to 300 steps. For every step of the proof, the current sequent, the applied proof rule, used lemmas and simplifier rules, the used substitution for the variable instantiation, the active heuristics, and a lot of internal information are stored and accessible. Characteristic proof steps are easy to identify, similarities between different parts of the proof can be recognized, and the proof can be saved in any state and continued later. If a premise of a proof cannot be closed, the proof tree allows to easily trace back the problem to its cause, e.g. a problematic branch of a program present in the original conclusion.

The nodes of the proof tree have different forms and colors. Filled circles represent steps performed by a heuristic; open circles represent interactive steps. The color indicates the different kinds of steps. For instance, induction steps are red. The edges of the proof also have different colors. Edges leading to an incomplete subtree are red, the edges of completely proved subtrees are green. At the nodes representing the interactive usage of a lemma, the name of the lemma is added to the node.

Clicking on nodes allows to view information about the goal or to inspect which simplifier rules have been used by the simplifier. When a premise is not provable, tracing through the branch is regularly used to identify where or why something went wrong.

Incorrect decisions can be undone by standard chronological backtracking. But since the incorrect step is usually not the last one, it is typically done by pruning the proof tree, i.e. removing the complete subtree above a selected node. Often fixing the incorrect step allows to successfully reapply the steps of the subtree (without its incorrect first step) that has been pruned away, since the approach for proving the branch is not affected by the fix.

Such a reapplication is directly supported in KIV with the *replay* functionality. The mechanism allows the user to reuse (parts of) proofs, even if the goal has changed, using a sophisticated strategy to adjust proof steps. Thus, replaying is also a powerful tool for re-validating invalid proofs: if a proof becomes invalid due to a minor change in one (or some) of its used theorems, most of the time, the old proof can be replayed automatically (the same applies if the theorem goal itself has changed).

All in all, KIV offers helpful features to support the proof engineer in the process of developing specifications and proofs. Among them are a direct interaction with the proof structure through proof trees, assistance for automation using heuristics, and context-sensitive rule selection. Finally, KIV's correctness management supports maintaining many proofs split into theorem bases for many specifications. More detailed information on the KIV GUI can be found in [20].

4 Components, Refinement and Program Logic

4.1 Sequential Programs

KIV features an imperative programming language with recursive procedures. Recently, the programming language was extended by constructs for exception handling. The syntax of sequential programs in KIV is given by the grammar

$$\begin{aligned}\alpha := \quad & \textbf{skip} \mid \textbf{abort} \mid \underline{x} := \underline{t} \mid \alpha_1; \alpha_2 \mid \textbf{if } \varepsilon \textbf{ then } \alpha_1 \textbf{ else } \alpha_2 \\ & \mid \textbf{let } \underline{x} = \underline{t} \textbf{ in } \alpha \mid \textbf{choose } \underline{x} \textbf{ with } \varphi \textbf{ in } \alpha_1 \textbf{ ifnone } \alpha_2 \mid \alpha_1 \textbf{ or } \alpha_2 \\ & \mid \textbf{while } \varepsilon \textbf{ do } \alpha \mid \textbf{proc\#}(\underline{t}; \underline{x}; \underline{y}) \mid \textbf{throw } \mathtt{op} \mid \textbf{try } \alpha \textbf{ catch } \underline{\eta}\end{aligned}$$

The program **skip** does nothing and **abort** is the program that never terminates. Assignments $\underline{x} := \underline{t}$ assign each variable x_i the value of term t_i simultaneously. This allows for swaps of the form $x, y := y, x$ in a single statement with no additional variables necessary. Two programs α_1 and α_2 can be executed in order

by sequential composition (;). The **if-then-else** and **while** programs function as usual.

The **let** program is used to introduce new variables $\underline{x}$ initialized with values $\underline{t}$. **choose** also introduces new local variables $\underline{x}$ but with a random valuation that satisfies a condition φ. If such a valuation can be found, the program α_1 is executed using these variables. If no values that satisfy φ exist, the program α_2 is executed. The **or** program makes an nondeterministic choice between two programs α_1 and α_2.

For **if**-programs the **else**-branch can be omitted if α_2 is the program **skip**. Similarly, for **choose**-programs α_2 can be dropped if it is the program **abort**. A '**with** true' clause in a **choose** can be omitted too.

A procedure **proc#** can be called with input arguments $\underline{t}$, reference arguments $\underline{x}$, and output arguments $\underline{y}$ by the statement **proc#**$(\underline{t}; \underline{x}; \underline{y})$, where the different argument types are separated by semicolons. Input arguments may be arbitrary terms while only variables are allowed to be passed as reference and output arguments. Typically, a procedure has a declaration of the form **proc#**$(\underline{x}; \underline{y}; \underline{z})\{\alpha\}$ with disjoint formal parameters $\underline{x}$, $\underline{y}$, and $\underline{z}$. α must set all output parameters $\underline{z}$, only the input parameters $\underline{x}$ and the reference parameters $\underline{y}$ can be read in α. Updates to $\underline{x}$ and $\underline{y}$ are allowed, however, updates to $\underline{x}$ are invisible to the caller of **proc#**.

The recently added **try-catch** and **throw** constructs enable exception handling in KIV programs. A program statement may now raise an exception if a partial function (see Sec. 2) is applied to arguments outside of its domain. Each exception is thus coupled with an operation `op`. For example, the subtraction operation $-$ on natural numbers throws its exception in the program $m := n_0 - n_1$ when $n_0 < n_1$. Additionally, the exception of operation `op` can be thrown explicitly by the **throw** program. **try-catch** blocks can be used to catch exceptions within a program α by giving exception handlers

$$\underline{\eta} \equiv \textbf{case } \mathtt{op}_1 :: \alpha_1;\ \ldots\ ;\ \textbf{case } \mathtt{op}_n :: \alpha_n;\ \textbf{default} :: \alpha_{default}$$

$\underline{\eta}$ can contain exception handlers for any number of operations $\mathtt{op}_1, \ldots, \mathtt{op}_n$ as well as an **default** exception handler (both are optional). If the leading statement of α throws an exception $\mathtt{op}_i$ for which a matching handler **case** $\mathtt{op}_i :: \alpha_i \in \underline{\eta}$ exists, the remaining statements of α are discarded and α_i is executed. If no matching handler exists but there is a **default** handler, then $\alpha_{default}$ is executed, otherwise the exception is not caught.

A small-step semantics of concurrent programs is explained in Sec. 5. For sequential programs the semantics can be abstracted to a relation $[\![\alpha]\!] \subseteq ST \times ST \times (Op \cup \{\top\})$ between states ST (valuation of program variables), augmented with Op and $\top$ to express termination with or without an exception. $(v, v', \zeta) \in [\![\alpha]\!]$ iff there is a terminating execution of the program α starting in state $v \in ST$ and finishing in state $v' \in ST$, without raising an exception (if $\zeta = \top$) or finishing with an exception $\zeta \in Op$.

4.2 Weakest Precondition Calculus with Exceptions in KIV

Reasoning about sequential programs in KIV is done with a weakest-precondition calculus, borrowing notation from Dynamic Logic (DL) [24] including its two standard modalities: formula $[\alpha]\varphi$ (*box*) denotes that for every terminating run the final state must satisfy φ, corresponding to the weakest liberal precondition $wlp(\alpha, \varphi)$. The formula $\langle\alpha\rangle\varphi$ (*diamond*) guarantees that there is a terminating execution of α that establishes φ. Finally, the formula $\langle\!|\alpha|\!\rangle\,\varphi$ (*strong diamond*) states that the program α is guaranteed to terminate and that all final states reached satisfy φ. This is equivalent to the weakest precondition $wp(\alpha, \varphi)$. For deterministic programs the two formulas $\langle\alpha\rangle\varphi$ and $\langle\!|\alpha|\!\rangle\,\varphi$ are equivalent. As a sequent, partial and total correctness of α with respect to pre-/post-conditions $Pre/Post$ are written as $Pre \vdash [\alpha]Post$ and $Pre \vdash \langle\!|\alpha|\!\rangle\, Post$. To handle exceptions the modalities are extended by exception specifications

$$\underline{\xi} \equiv \mathtt{op}_1 :: \varphi_1,\ \ldots\ ,\ \mathtt{op}_k :: \varphi_k,\ \textbf{default} :: \varphi_{default}$$

which yields program formulas of the form $\langle\!|\alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi})$, $\langle\alpha\rangle(\varphi\ ;\ \underline{\xi})$, and $[\alpha](\varphi\ ;\ \underline{\xi})$, respectively. The exception specifications allow to give additional postconditions for executions that terminate with a specific exception, e.g. φ_1 must hold if α terminates with exception $\mathtt{op}_1$, or to give a generic exception postcondition $\varphi_{default}$ for executions of α that terminate with an exception $\mathtt{op} \notin \mathtt{op}_1, \ldots, \mathtt{op}_k$. If one wants to show the absence of exceptions, the exception specifications $\underline{\xi} \equiv \textbf{default} :: \mathtt{false}$ can be chosen, which is the default and omitted from program formulas.

The main proof technique for verifying program correctness in KIV is *symbolic execution*. Basically, a symbolic execution proof step executes the first statement of the program and calculates the strongest postconditions from the preconditions. When the symbolic execution of the program is completed, the proof of the postcondition is performed purely in predicate logic. In the following we will present an excerpt of the calculus rules for program formulas in KIV, with a focus on the recent addition of exceptions and exception handling.

Fig. 5 shows the rule for parallel assignments for total correctness (the rule is identical for the other modalities). The rule uses a vector $\underline{x}'$ of fresh variables to store the values of $\underline{x}$ before the assignment. The assignment is removed and instead the formula $\underline{x} = \underline{t}^{\underline{x}'}_{\underline{x}}$ is added to the antecedent ($\underline{t}^{\underline{x}'}_{\underline{x}}$ denotes the renaming of $\underline{x}$ to $\underline{x}'$ in $\underline{t}$). Note that renaming is possible on all program formulas, while substitution of $\underline{x}$ by general terms $\underline{t}$ in $\langle\!|\alpha|\!\rangle\,\varphi$ is not possible and just yields $\langle\!|\underline{x} := \underline{t}; \alpha|\!\rangle\,\varphi$. Only when the assignment is the last statement of the program and α is missing, the standard premise of Hoare calculus, which replaces the program formula in the premise with $\varphi^{\underline{t}}_{\underline{x}}$, can be used.

Since the expressions $\underline{t}$ can contain partial operations, this is correct only when the evaluation of $\underline{t}$ does not raise any exception: we use $\delta(\underline{e})$ to describe the condition that the expressions $\underline{e}$ are defined, i.e. that all partial operations in $\underline{e}$ are applied to arguments within their respective domain. For example, a heap constructor term $(h\ \mathtt{++}\ r_0)[r_1, obj]$ (see Sec. 2.2) produces the definedness

$$\frac{\Gamma^{\underline{x}'}_{\underline{x}},\ \delta(\underline{t}^{\underline{x}'}_{\underline{x}}),\ \underline{x} = \underline{t}^{\underline{x}'}_{\underline{x}} \vdash \langle\!|\alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta^{\underline{x}'}_{\underline{x}} \quad \text{where } \underline{x}' \text{ fresh} \qquad Exc(\Gamma, \Delta, \underline{t}, \varphi, \underline{\xi})}{\Gamma\ \vdash \langle\!|\underline{x} := \underline{t}; \alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta}$$

Fig. 5: Calculus rule for assignments.

condition

$$\delta((h \texttt{ ++ } r_0)[r_1, obj]) \ \equiv\ (r_0 \neq \texttt{null} \wedge \neg\, r_0 \in h) \wedge (r_1 \neq \texttt{null} \wedge r_1 \in (h \texttt{ ++ } r_0))$$

Exception premises $Exc(\Gamma, \Delta, \underline{t}, \varphi, \underline{\xi})$ need to be shown for all potential violations of $\delta(\underline{t})$, i.e. for all partial operations $\texttt{op}_1, \ldots, \texttt{op}_m$ in $\underline{t}$, *throw premises* are generated. For each partial operation an exception condition is calculated following a bottom-up approach: the exception is thrown if and only if an application of $\texttt{op}_i$ violates the domain of $\texttt{op}_i$ and the evaluation of its arguments does not throw an exception. E.g. the assignment $h := (h \texttt{ ++ } r_0)[r_1, obj]$ would yield the two exception premises

$$\begin{aligned}
&Exc(\Gamma, \Delta, (h \texttt{ ++ } r_0)[r_1, obj], \varphi, \underline{\xi}) \equiv \\
&\quad (1)\ \Gamma \vdash r_0 = \texttt{null}\ \vee\ r_0 \in h \rightarrow \langle\!|\textbf{throw ++ }|\!\rangle\,(\varphi;\ \underline{\xi}),\ \Delta \\
&\quad (2)\ \Gamma \vdash (r_0 \neq \texttt{null}\ \wedge\ \neg\, r_0 \in h)\ \wedge\ (r_1 = \texttt{null}\ \vee\ \neg\, r_1 \in (h \texttt{ ++ } r_0)) \\
&\qquad\qquad \rightarrow \langle\!|\textbf{throw [ ]}|\!\rangle\,(\varphi;\ \underline{\xi}),\ \Delta
\end{aligned}$$

Computing exception clauses uses the standard left-to-right strategy for evaluating arguments, and a shortcut semantics for boolean connectives (like Java and Scala). The test $y \neq 0 \wedge x/y = 1$ of a conditional will never throw an exception, while switching the conjunction will throw the division exception, when $y = 0$.

The rules for **throw** are quite simple. If an operation is thrown for which a specific exception specification $\texttt{op} :: \varphi_\xi$ is given in $\underline{\xi}$, the program formula is discarded and replaced by the exception postcondition φ_ξ. If there is no exception specification in $\underline{\xi}$ for the thrown operation op, the default exception postcondition $\varphi_{default}$ must hold.

The **try-catch**-rule shown in Fig. 6 takes advantage of the fact that exception postconditions can also contain program formulas. The program α can be executed symbolically with adjusted exception specifications $\underline{\xi}'$ that include the exception handling of $\underline{\eta}$. $\underline{\xi}'$ contains exception specifications for all operations that have either already a specification in $\underline{\xi}$ or have a case in $\underline{\eta}$. For an operation op the adjusted exception postcondition φ'_ξ is defined as

$$\varphi'_\xi = \begin{cases} \langle\!|\alpha'|\!\rangle\,(\varphi\ ;\ \underline{\xi}) & \text{if } \textbf{case } \texttt{op} :: \alpha' \in \underline{\eta} \\ \varphi_\xi & \text{if } \texttt{op} \notin \underline{\eta} \wedge \texttt{op} :: \varphi_\xi \in \underline{\xi} \end{cases}$$

If an exception handler was given for op, φ'_ξ is set to a program formula with the exception handler program α' and the original postcondition φ and exception specifications $\underline{\xi}$. This has exactly the desired effect: if an op exception is thrown within α, symbolic execution continues with α' while the postcondition is still φ. For operations without a case in $\underline{\eta}$, the exception postcondition remains un-

$$\frac{\Gamma \vdash \langle\!|\alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}'),\ \Delta}{\Gamma \vdash \langle\!|\textbf{try}\ \alpha\ \textbf{catch}\ \{\ \underline{\eta}\ \}|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta}$$

Fig. 6: Calculus rule for try-catch programs.

$$\frac{\begin{array}{l}(1)\ \Gamma \vdash INV,\ \Delta\\ (2)\ Exc(INV, \langle\rangle, \varepsilon, \varphi, \underline{\xi})\\ (3)\ INV,\ \varepsilon,\ \delta(\varepsilon),\ z = t \vdash \langle\!|\alpha|\!\rangle\,(INV \wedge t \ll z;\ \underline{\xi})\\ (4)\ INV,\ \neg\,\varepsilon,\ \delta(\varepsilon) \vdash \varphi\end{array}}{\Gamma \vdash \langle\!|\textbf{while}\ \varepsilon\ \textbf{do}\ \alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta}$$

Fig. 7: Invariant rule for while loops.

changed ($\varphi'_\xi = \varphi_\xi$). The adjusted default exception postcondition $\varphi'_{default}$ is constructed similarly. If there is a default exception handler **default** :: $\alpha_{default} \in \underline{\eta}$, a program formula with $\alpha_{default}$ is built, otherwise the default postcondition $\varphi_{default}$ from $\underline{\xi}$ is used.

Proofs about recursive procedures are typically done using (well-founded) induction. For **while**-loops, typically the invariant rule shown in Fig. 7 is used, though the more general induction rule occasionally leads to simpler proofs (see e.g. the proof online at [30] for Challenge 3 at the VerifyThis2012 competition).

The rule requires an invariant formula INV as an input from the user from which multiple premises need to be proven: the invariant must hold at the beginning of the loop (1), INV must be stable over the loop body α (3), and INV must be strong enough to prove the postcondition φ after the loop was exited (4). Similar to the assignment rule, exception premises are generated for the loop condition ε (2). These cannot include Γ or Δ as ε is evaluated again after each iteration. Instead, the invariant INV can be assumed as well as $\delta(\varepsilon)$ for premises (3) and (4). The figure shows the rule for total correctness which requires to give a variant t that decreases with every iteration of the loop ($t \ll z$ in premise (3)). The rule functions analogously for diamond formulas, for partial correctness no decreasing variant is necessary.

4.3 Hierarchical Components and Refinement

For the development of complex software systems in KIV we use the concept of hierarchical components combined with the contract approach to data refinement [12]. A component is an abstract data type $(ST, \texttt{Init}, (\texttt{Op}_j)_{j\in J})$ consisting of a set of states ST, a set of initial states $\texttt{Init} \subseteq ST$, and a set of operations $\texttt{Op}_j \subseteq In_j \times ST \times ST \times Out_j$. An operation $\texttt{Op}_j$ takes inputs In_i and outputs Out_j and modifies the state of the component. Operations are specified with contracts using the operational approach of ASMs [6]: for an operation $\texttt{Op}_j$, we give a precondition pre_j and a program α_j (that establishes the postcondition of the operation) in the form of a procedure declaration $\textbf{op}_j\#(in_j; st; out_j)$ **pre** pre_j $\{\alpha_j\}$. Instead of defining initial states directly, we also give a procedure declaration $\textbf{init}\#(in_{init}; st; out_{init})$ $\{\alpha_{init}\}$ where in_{init} are initialization parameters, that determine the intial state, and out_{init} is an error code, that may indicate that initialization with these parameters failed.

Components are distinguished between specifications and implementations. The former are used to model the functional requirements of a (sub-)system and are typically kept as simple as possible by heavily utilizing algebraic functions and nondeterminism. The approach is as general as specifying with pre- and postconditions, since **choose** st', out' **with** $post(st', out')$ **in** $st, out := st', out'$ can be used to establish any postcondition *post* over state *st* and output *out*. Implementations are typically deterministic and only use constructs that allow to generate executable Scala- or C-code from them with our code generator.

The functional correctness of implementation components is then proven by a data refinement of the corresponding specification components (we write $\mathtt{C} \leq \mathtt{A}$ if $\mathtt{C} = (ST^{\mathtt{C}}, \mathtt{Init}^{\mathtt{C}}, (\mathtt{Op}_i^{\mathtt{C}})_{j \in J})$ is a refinement of $\mathtt{A} = (ST^{\mathtt{A}}, \mathtt{Init}^{\mathtt{A}}, (\mathtt{Op}_i^{\mathtt{A}})_{j \in J})$ where $\mathtt{C}$ and $\mathtt{A}$ have the same set of operations J). Proofs for such a refinement are done with a forward simulation $R \subseteq ST^{\mathtt{A}} \times ST^{\mathtt{C}}$ using commuting diagrams. This results in correctness proof obligations for all $j \in J$ (an extra obligation ensures that $\mathtt{Init}^{\mathtt{A}}$ and $\mathtt{Init}^{\mathtt{C}}$ establish matching states).

$$R(st^{\mathtt{A}}, st^{\mathtt{C}}),\ pre_j^{\mathtt{A}}(st^{\mathtt{A}})$$
$$\vdash \langle\!|\mathbf{op}_j^{\mathtt{C}}\#(in_j; st^{\mathtt{C}}; out_j)|\!\rangle\ \langle \mathbf{op}_j^{\mathtt{A}}\#(in_j; st^{\mathtt{A}}; out'_j)\rangle (R(st^{\mathtt{A}}, st^{\mathtt{C}}) \wedge out_j = out'_j)$$

Informally, one has to prove that, when starting in R-related states, for each run of an operation $\mathbf{op}_j^{\mathtt{C}}\#$ of $\mathtt{C}$ there must be a matching run of $\mathbf{op}_j^{\mathtt{A}}\#$ of $\mathtt{A}$ that maintains $R(st^{\mathtt{A}}, st^{\mathtt{C}})$ with the same inputs and outputs. The obligations also require to show that the precondition $pre_j^{\mathtt{A}}(st^{\mathtt{A}})$ is strong enough to establish the precondition $pre_j^{\mathtt{C}}(st^{\mathtt{C}})$ if $R(st^{\mathtt{A}}, st^{\mathtt{C}})$ holds. This obligation is implicit as the call rule creates this premise for a procedure with a precondition.

For each component invariant formulas $inv(st)$ over the state *st* can be given, which must be maintained by all $(\mathtt{Op}_j)_{j \in J}$. This simplifies (or even makes it possible in the first place) to prove the correctness proof obligations of a refinement as invariants $inv^{\mathtt{A}}(st^{\mathtt{A}})$ and $inv^{\mathtt{C}}(st^{\mathtt{C}})$ are added as assumptions. If an invariant is given for a component, additional proof obligations for all its operations are generated that ensure that the invariant holds. Additionally, one can give an individual postcondition $post_j(st)$ for an operation which extends its invariant contract.

$$pre_j(st),\ inv(st) \vdash \langle\!|\mathbf{op}_j\#(in_j; st; out_j)|\!\rangle\ (inv(st) \wedge post_j(st))$$

These invariant contracts can be applied when proving the refinement proof obligations and may further simplify the proofs since symbolic execution of the operation can be avoided. The theory has also been extended with proof obligations for crash-safety, see [16] for more details.

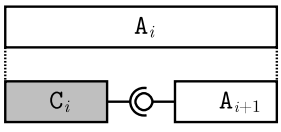

Fig. 8: Data refinement with subcomponents.

To facilitate the development of larger systems, we introduced a concept of modularization in the form of *subcomponents*. A component (usually an implementation) can use one or more components as subcomponents (usually specifications). The client component cannot access the state of its subcomponents directly but only via calls to the interface operations of the subcomponents. Using subcomponents, a refinement

hierarchy is composed by multiple refinements like the one shown in Fig. 8. A specification component $\mathtt{A}_i$ is refined by an implementation $\mathtt{C}_i$ (dotted lines in Fig. 8) that uses a specification $\mathtt{A}_{i+1}$ as a subcomponent (—©— in Fig. 8, we write $\mathtt{C}_i(\mathtt{A}_{i+1})$ for this subcomponent relation). This pattern then repeats in the sense that $\mathtt{A}_{i+1}$ is refined further by an implementation $\mathtt{C}_{i+1}$ that again uses a subcomponent $\mathtt{A}_{i+2}$ and so on. $\mathtt{A}_i$ may also be used as a subcomponent of an implementation $\mathtt{C}_{i-1}$ if it is not the toplevel specification. The complete implementation of the system then results from composing all individual implementation components $\mathtt{C}_0(\mathtt{C}_1(\mathtt{C}_2(...)))$. In [16] we have shown that $\mathtt{C} \leq \mathtt{A}$ implies $\mathtt{M}(\mathtt{C}) \leq \mathtt{M}(\mathtt{A})$ for a client component $\mathtt{M}$ which ensures that the composed implementation is in fact a correct refinement of its toplevel specification $\mathtt{A}_0$, i.e. $\mathtt{C}_0(\mathtt{C}_1(\mathtt{C}_2(...))) \leq \mathtt{A}_0$. This allows us to divide a complex refinement task into multiple, more manageable ones.

5 Concurrency, Temporal Logic and Rely-Guarantee Calculus

5.1 Concurrent Programs and Their Semantics

KIV supports concurrency in the form of weak fair and non-fair interleaving of sequential programs. Concurrent programs β extend the syntax of sequential ones (α) by the following constructs

$$\begin{aligned}\beta := \quad & \alpha \mid \textbf{if*}\ \varepsilon\ \textbf{then}\ \beta_1\ \textbf{else}\ \beta_2 \mid \textbf{atomic}\ \varepsilon\ \{\beta\} \mid \beta_1 \parallel_{\mathrm{nf}} \beta_2 \\ & \mid \beta_1 \parallel \beta_2 \mid \textbf{forall} \parallel \underline{x}\ \textbf{with}\ \varphi\ \textbf{do}\ \beta \mid \textbf{forall} \parallel_{\mathrm{nf}} \underline{x}\ \textbf{with}\ \varphi\ \textbf{do}\ \beta\end{aligned}$$

Programs execute atomic steps, consisting of one assignment, the evaluation of the test of a condition, binding a variable with **let** or calling/returning from a procedure. To enable thread-local reasoning, they are now assumed to execute their steps in an environment that may modify the state in between program steps. The environment may consist of other interleaved programs or a global environment, e.g. a physical environment that changes sensor values read by the program.

There are several new constructs that may be freely mixed with the sequential constructs. The first is the variant **if*** of a conditional, which evaluates the test together with the first step of the branch taken in one atomic step. It is used to model test-and-set, or CAS (compare-and-swap) instructions.

The construct **atomic** $\varepsilon\ \{\beta\}$ is assumed to (passively) wait for the environment to make the test ε true. While ε is false, the program is *blocked*. A program where the environment never enables the test is deadlocked. When the test becomes true, the program β is executed in a single step. The typical use of the construct is to model locking with **atomic** $\mathit{lock} = \mathtt{free}\ \{\mathit{lock} := \mathtt{locked}\}$. Atomic programs are also used in Lipton's reduction strategy [14, 36], which proves that program instructions can be combined to larger atomic blocks when specific commutativity conditions hold.

Both $\beta_1 \parallel_{\mathrm{nf}} \beta_2$ and $\beta_1 \parallel \beta_2$ interleave steps of β_1 and β_2 non-deterministically. The interleaving is blocked only, when both programs are blocked. The first as-

sumes no fairness, so when β_1 does not terminate and is never blocked, one possible run executes steps of β_1 only. The second $\beta_1 \parallel \beta_2$ has a weak fairness constraint: if β_2 is enabled continuously (i.e. is never blocked) then it will eventually execute a step, even if β_1 is always enabled. Weak fairness is typically assumed for programs which use locks while CAS-based programs often ensure the progress condition of lock-freedom that does not assume fairness.

The programs **forall** $\|$ and **forall** $\|_{\mathrm{nf}}$ generalize binary interleaving to interleaving instances of β for all values that satisfy φ bound to local variables $\underline{x}$. The typical use would be **forall** $\|$ n **with** $n < m$ **do** β where the body β uses variable n as the thread identifier.

Earlier versions of RG calculus in KIV used an equivalent recursive **spawn#**-program (called with m and the variables $\underline{x}$ used in β) which is defined as

$$\textbf{spawn\#}(n;\underline{x})\{\ \textbf{if*}\ n = 0\ \textbf{then skip else}\ \{\ \beta \parallel \textbf{spawn\#}(n-1;\underline{x})\ \}\ \}$$

They imported a theory which verified the correctness of the forall-rule given at the end of the next section.

Formally, the semantics of programs $[\![\beta]\!]$ is defined as a set of finite or infinite state sequences also called *intervals* following the terminology of interval temporal logic (ITL) [41]. Formally an interval is of the form

$$I = (I(0), I(0)_b, I'(0), I(1), I(1)_b, I'(1), I(2), \ldots, \zeta)$$

where every $I(k)$ and $I'(k)$ are valuations which map variables to values. The transitions from $I(k)$ to $I'(k)$ are program transitions and the transitions from $I'(k)$ to $I(k+1)$ from a primed to the subsequent unprimed state are environment transitions. Hence, intervals alternate between program and environment transitions, similar to the reactive sequence semantics in [11].

To model passive waiting, the boolean flag $I(k)_b$ denotes whether the program transition from $I(k)$ to $I'(k)$ is *blocked*, i.e. the program waits to continue its execution. In this case, when $I(k)_b = \mathbf{tt}$, then $I'(k) = I(k)$.

To model exceptions when running a program, the final state of a finite run carries information ζ whether an exception has happened. This may be either $\top$ to indicate regular termination without exception or the information that an operation $op \in OP$ has thrown its exception. To have uniform notation, we assume that $\zeta = \infty$ for infinite (non-terminating) runs, so $\zeta \in OP \cup \{\top, \infty\}$.

The semantics of programs is *compositional*, i.e. the semantics of complex programs can be constructed by combining intervals that are members of the semantics of its parts. This has been explained in detail for programs without exceptions in [55]. Adding exceptions for the sequential case is straightforward, for interleaving the combined run ends with an exception if one of the interleaved intervals does. Unlike in programming languages, where exceptions are thread-local, we therefore have *global* exceptions, which abort the whole interleaved program. If necessary, the global effect must be avoided by exception handlers in the interleaved programs.

5.2 Temporal and Program Formulas

To verify concurrent programs, KIV's logic is based on the idea of having *programs as formulas*. To do this, the semantics of expressions e is generalized from $[\![e]\!](v)$ to $[\![e]\!](I)$ using an interval I instead of a single state v. The expressions considered so far refer to the initial valuation $I(0)$ of the interval only. The extended semantics makes it possible to view a program β with free variables $\underline{z}$ as a formula $[: \underline{z} \mid \beta]$ (expression with boolean result) which returns **tt** iff the semantics of β includes the interval I. That β has a temporal property ψ is then simply expressed as the implication $[: \underline{z} \mid \beta] \rightarrow \varphi$.

The resulting calculus is again based on symbolic execution of programs as well as of temporal formulas. The resulting calculus has been described in detail in [55]. The calculus is strong enough to define rely-guarantee formulas as abbreviations of temporal logic formulas and to formally derive the rules of rely-guarantee calculus. Since rely-guarantee calculus is the one we predominantly use in the practical verification of programs, we have explicitly added rely-guarantee formulas and derived rules for them.

The extended expression language partitions variables into flexible variables, that may be modified by concurrent programs, and static ones, that always have the same value in all states $I(0)$, $I'(0)$, ... of an interval. In KIV, flexible variables start with uppercase letters, all others are static. The abstract syntax definition uses y to denote a static and z for a flexible variable. Flexible variables are allowed in quantifiers, but not as parameters of λ-expressions.

The extended language allows to use primed and double primed flexible variables y' and y'' in predicate logic expressions. $[\![y']\!](I)$ and $[\![y'']\!](I)$ are defined as $I'(0)(y)$ and in $I(1)(y)$, except for the case where the interval consists of a single state. For such an *empty* interval the value of both is $I(0)(y)$. Formulas like $y' = y$ or $y'' \geq y'$ therefore talk about the relation of the first program step (y is not changed) and about the first environment step (y is not decremented). They are used as guarantee and rely formulas that constrain program and environment steps. We use G to denote a predicate logic formula over unprimed and primed variables and R to denote one over primed and double primed variables. We write φ' and φ'' for predicate logic formulas where one resp. two extra primes are added to every free variable that is flexible.

The extended syntax then extends higher-order and wp-calculus formulas that may use any primed or double primed variables written χ (φ still denotes a formula without primed variables) to temporal expressions ψ, including program expressions. These have the following syntax:

$$\psi := \chi \mid [: \underline{z} \mid \beta] \mid \Box\ \psi \mid \Diamond\ \psi \mid \psi_1\ \textbf{until}\ \psi_2 \mid \textbf{last} \mid \textbf{lastexc} \mid \textbf{lastexc}(\textbf{op}) \mid \textbf{blocked} \mid [: \underline{z} \mid R, G, \varphi, \beta](\psi; \underline{\xi}) \mid \langle : \underline{z} \mid R, G, \mathit{Runs}, \varphi, \beta\rangle(\psi; \underline{\xi})$$

A formal semantics is given in [55], here we just give an informal explanation.

- $[: \underline{z} \mid \beta]$ is a program formula, where the used variables of β are required to be a subset of $\underline{z}$. The formula holds over an interval (i.e. the semantics evaluates to **tt**) if the program steps are steps of the program. The environment steps between program steps are not constrained: they may arbitrarily change

the values of variables. The frame assumption $\underline{z}$ indicates that non-program variables can change arbitrarily in program steps.
- $\Box\,\psi$, $\Diamond\,\psi$ and ψ_1 **until** ψ_2 are the standard formulas of linear temporal logic.
- **last** is true, if the interval is empty, i.e. consists of just the state $I(0)$.
- **lastexc(op)** resp. **lastexc** is true on an empty interval where $\zeta = op$ resp. $\zeta \neq \top$. The formula $\Diamond(\mathbf{last} \wedge \neg\ \mathbf{lastexc})$ therefore states, that the interval is finite (terminates) without exception.
- **blocked** is true for non-empty intervals, where the first step is blocked, i.e. $I(0)_b = \mathbf{tt}$. The formula $\Box$ **blocked** is used to express a deadlock.

The last two formulas are used to express rely-guarantee (RG) properties of programs, and we assume the reader to be familiar with the basic ideas. In [60] such an RG assertion would be written as β **sat** $\{Pre, R, G, Post\}$, another common notation is as an extended Hoare-quintuple $\{Pre, R\}\ \beta\ \{G, Post\}$. Often, whether partial or total correctness is intended, depends on the context. In KIV, partial and total correctness are expressed as a sequents

$$Pre \vdash [:\ \underline{z}\ |, R, G, \beta]Post \qquad\qquad Pre \vdash \langle :\ \underline{z}\ |\ R, G, Runs, \alpha\rangle Post$$

Like for the wp-formulas we leave away the default exception condition $\xi \equiv$ **default** :: `false` that forbids any exceptions for the final state.

Partial correctness asserts that if precondition Pre holds in the initial state, then program steps will not violate the guarantee G and the final state will not violate $Post$ (and have no exception) unless an earlier environment step violated R. The formula implies that partial correctness holds: if all environment steps are rely steps then all program steps will be guarantee steps and in final states the postcondition will hold.

Total correctness guarantees two additional properties. First, when the rely is never violated, then the program is guaranteed to terminate. Second, in all states, where predicate $Runs$ holds, the next program step is guaranteed not to be blocked. The additional $Runs$-predicate is used to verify deadlock-freedom: when an interleaved program satisfies total correctness with $Runs =$ `true`, then the program is deadlock-free: there is always at least one of its threads that is not currently waiting to acquire a lock.

At the end of this section we want to note that the interval semantics of programs can used to define the semantics of wp-calculus formulas as well by abstracting to initial and final states of the interval and assuming an "empty" environment that does not change program variables via the equivalences

$$[\alpha]\varphi \equiv [:\ \underline{z}\ |\ \underline{z}'' = \underline{z}', \mathtt{true}, \alpha]\varphi \qquad \langle\!|\alpha|\!\rangle\ \varphi \equiv \langle :\ \underline{z}\ |\ \underline{z}'' = \underline{z}', \mathtt{true}, \alpha\rangle\varphi$$

In the formulas, $\underline{z}$ are the flexible variables that occur free in α (all variables, except those bound by **let**, **choose** or **forall**) or free in φ.

5.3 Rely-Guarantee Calculus

Since rely-guarantee formulas can be viewed as abbreviations of temporal formulas, we could just use the calculus defined in [55] for deduction (and earlier

case studies have done so). Since rely-guarantee formulas are now available directly, it is unnecessary for a user to get familiar with temporal logic formulas. Knowledge of the calculus given in this section is sufficient to do partial or total correctness proofs for concurrent programs. The embedding into temporal logic is useful however to prove results about progress conditions such as lock-freedom [58] or starvation-freedom [57], which would not be possible with pure RG calculus only.

Symbolic execution using the rely-guarantee calculus resembles symbolic execution in wp-calculus. The extra effort needed when proving RG formulas can be seen when looking at the rule for assignment:

$$\frac{\begin{array}{l} Pre(\underline{y}_0), \underline{y}_1 = \underline{t}(\underline{y}_0) \vdash G(\underline{y}_0, \underline{y}_1) \\ Pre(\underline{y}_0), \underline{y}_1 = \underline{t}(\underline{y}_0), R(\underline{y}_1, \underline{z}) \vdash \langle : \underline{z} \mid R, G, \beta \rangle Post \end{array}}{Pre(\underline{z}) \vdash \langle : \underline{z} \mid R(\underline{z}', \underline{z}''), G(\underline{z}, \underline{z}''), \underline{z} := \underline{t}(\underline{z}); \beta \rangle Post}$$

For easier notation the rule assumes that all variables of the frame assumption are assigned. We also leave away the side conditions for possible exceptions when evaluating $\underline{t}$. These are the same as in wp-calculus. The rule makes explicit that Pre and t may contain variables from $\underline{z}$ by writing them as arguments in the conclusion. Analogously, R and G may depend on $\underline{z}$, $\underline{z}'$, and $\underline{z}''$.

In the semantics, symbolic execution of the assignment reduces the interval $I = (I(0), I(0)b, I'(0), I(1), \ldots)$ to the one shorter interval $(I(1), \ldots)$. The values of variables $\underline{z}$ in states $I(0)$ and $I'(0)$ are now stored in two vectors $\underline{y}_0$ and $\underline{y}_1$ of fresh static variables, the remaining program β again starts with the values of the variables in $\underline{z}$. The old precondition now holds for $\underline{y}_0$, the values stored in $\underline{y}_1$ are equal to the ones of the terms $\underline{t}(\underline{y}_0)$. As the assignment rule shows, the RG calculus has two differences to wp-calculus. First there is an additonal premise that asserts that executing the assignment satisfies the guarantee (which is simple usually). Second, the main premise needs two vectors of fresh variables instead of one to store old values: one before the assignment and one after the assignment but before the environment step.

The other rules of RG-calculus (e.g. the invariant rules for partial and total correctness) look very similar to wp-calculus. The only difference is again that a premise is generated that ensures that the step is a guarantee step. Finally, we need a rule for parallel composition. We give the rule for one flexible variable u, the generalization to several variables should be obvious.

The rule assumes that a lemma for the body β is available, that is applied by the rule and shown as its first premise. The precondition Pre of the goal can depend on $\underline{z}$, and we write $\mathrm{Pre}(\underline{y})$ for the assertion, where $\underline{z}$ has been replaced by $\underline{y}$. Similar conventions apply for the other formulas. E.g. Rely R_0 can depend on $u', \underline{z}', u'', \underline{z}''$, so write $R_0(u, \underline{y}_0, u, \underline{y}_1)$ for an instance. Formula φ may use u and $\underline{z}$. Since u is a local variable, the environment steps of β can not change its value, so the lemma can (and should) include the rely condition $u'' = u'$. Since the program is also prohibited from modifying variable u (otherwise it could not be used to identify the thread executing), instances of R_0 (and similarly: G_0) always use the same static variable v to instantiate u' and u''. The possible

values for u are chosen in the inital state before the **forall** executes, using new static variables $\underline{y}$ for this initial state and static variables v, v_1, v_2 for the values that all satisfy $\bar{P}re(\underline{u}) \wedge \varphi(v, \underline{u})$.

(1) $Pre_0 \vdash \langle : u, \underline{z} \mid R_0, G_0, Runs_0, \beta \rangle Post_0$

(2) $Pre(\underline{y}), \varphi(v, \underline{y}) \vdash G_0(v, \underline{y}_0, v, \underline{y}_0)$

(3) $Pre(\underline{y}), \varphi(v, \underline{y}) \vdash R_0(v, \underline{y}_0, v, \underline{y}_1) \wedge R_0(v, \underline{y}_1, v, \underline{y}_2) \rightarrow R_0(v, \underline{y}_0, v, \underline{y}_2)$

(4) $Pre(\underline{y}), \varphi(v_1, \underline{y}), \varphi(v_2, \underline{y}), v_1 \neq v_2 \vdash G_0(v_1, \underline{y}_0, v_1, \underline{y}_1) \rightarrow R_0(v_2, \underline{y}_0, v_2, \underline{y}_1)$

(5) $Pre(\underline{y}), \varphi(v, \underline{y}) \vdash Pre_0(v, \underline{y}_0) \wedge R_0(v, \underline{y}_0, v, \underline{y}_1) \rightarrow Pre_0(v, \underline{y}_1)$

(6) $Pre(\underline{y}), \varphi(v, \underline{y}) \vdash Post_0(v, \underline{y}_0) \wedge R_0(v, \underline{y}_0, v, \underline{y}_1) \rightarrow Post_0(v, \underline{y}_1)$

(7) $Pre(\underline{y}) \vdash APre(\underline{y})$

(8) $Pre(\underline{y}), Rely(y_0, y_1) \vdash AR(\underline{y}, \underline{y}_0, \underline{y}_1)$

(9) $Pre(\underline{y}), EG(\underline{y}, \underline{y}_0, \underline{y}_1) \vdash Guar(\underline{y}_0, \underline{y}_1)$

(10) $Pre(\underline{y}), Runs(\underline{y}_0) \vdash \ ERuns(\underline{y}, \underline{y}_0)$

(11) $Pre(\underline{y}), APost(\underline{y}, \underline{z}) \vdash \langle : \underline{z} \mid R, G, Runs, \beta_0 \rangle Post$

$Pre \vdash \langle : \underline{z} \mid R, G, Runs, \{\textbf{forall} \parallel u \textbf{ with } \varphi \textbf{ do } \beta\}; \beta_0 \rangle Post$

Conditions (1) to (6) ensure that the forall satisfies

$$Pre(\underline{z}), APre(\underline{z}) \vdash \langle : \underline{z} \mid AR, EG, ERuns, \textbf{forall} \ldots \rangle APost$$

where

$$\begin{aligned}
APre(\underline{y}) &\equiv \forall\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \rightarrow Pre_0(v, \underline{y}) \\
APost(\underline{y}, \underline{z}) &\equiv \forall\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \rightarrow Post_0(v, \underline{z}) \\
EG(\underline{y}, \underline{z}, \underline{z}') &\equiv \exists\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \wedge G_0(v, \underline{z}, v, \underline{z}') \\
ERuns(\underline{y}, \underline{z}) &\equiv \exists\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \wedge Runs_0(v, \underline{z}) \\
AR(\underline{y}, \underline{z}', \underline{z}'') &\equiv \forall\ v.\ Pre(\underline{y}) \wedge \varphi(v, \underline{y}) \rightarrow R_0(v, \underline{z}', v, \underline{z}'')
\end{aligned}$$

The remaining conditions (7) to (11) ensure that the given predicates used in the conclusion are weaker (for rely, precondition, and runs) resp. stronger (for guarantee) than the most general ones defined above, and that the remaining program β_0 is correct with the established postcondition $APost$ of the **forall**.

The rule for interleaving two programs $\beta_1 \parallel \beta_2$ is essentially the special case we get from the equivalent program **forall** $\parallel b$ **with** `true` **do if*** b **then** β_1 **else** β_2 where b is a boolean variable. Two lemmas are now required, one for β_1 and one for β_2 (corresponding to the two boolean values).

6 Applications

This section gives a short overview over applications that have been modeled and verified using KIV. We start with a brief overview over historically important case studies on sequential systems, described in the next subsection. Case studies

on concurrency are described in Sec. 6.2. Finally, Sec. 6.3 gives a brief overview over concepts used in the development of a verified file system for flash memory, developed in the Flashix project, which is by far the largest project we have tackled so far.

6.1 Overview

Initially, KIV had a focus on verifying single programs with Dynamic Logic, which is more flexible than just generating verification conditions from a standard Hoare-calculus [23] (see Challenge 3 of the VerifyThis competition 2012 in [15] for a recent example).

The focus however shifted early on to the development of modular, sequential software. An early concept there were algebraic modules [46], which were used to e.g. verify the correctness of Dynamic Hashtables or AVL trees [47]. Algebraic modules are stateless, and are nowadays integrated with the instantiation concept: a parameter P may be instantiated by placing a restriction and a congruence on the instance A. A simple example would be to use duplicate-free (restriction) lists, where the order of elements is ignored (congruence), to implement sets. This concept can be used to formally verify the consistency of non-free data type specifications.

Later on the focus shifted to the verification of components with state. The biggest case study in this area was the verification [51] of a compiler for Prolog, that compiles to the Warren Abstract Machine (WAM), formalizing the development in [7] as a hierarchy of a dozen refinements. The case study uses the complete theory of ASM refinement [50], which generalizes the theory of data refinement: the main correctness condition for a 1:1 diagram involving one abstract and one concrete operation (cf. Sec. 4.3) is generalized to using m:n diagrams, which are useful in particular for compiler verification.

Another area where component-based development was important, was the Mondex case study, which initially was about mechanizing a proof of of a protocol for the secure transfer of money between Mondex smartcards [59]. This case study became a part of the development of a strategy of model-based development for security protocols in general, starting with a UML model and ending with verified Java Code. For Mondex the original single refinement was extended to three refinements that end with verified Java code, see [21] for an overview.

6.2 Case studies on Concurrency

Research in this area has focused on the verification of efficient low-level implementations of components, in particular on CAS-based implementations, which are harder to verify and need different concepts for progress than standard lock-based implementations. The most complex proof using extensions of RG calculus to the thread-local verification of linearizability [26] and lock-freedom was a proof of Treiber's stack that used Hazard pointer with non-atomic reading and writing [39, 56, 58]. A number of other examples, e.g. one with fine-grained locking and formal proof obligations for starvation-freedom [57] can be found on [29].

The Web page also shows proofs for two of Cliff Jones' original examples that demonstrate the use of RG calculus: FINDP [22] (a parallel search over an array) and SIEVE [34] (parallel version of the Sieve of Erathostenes).

As part of the VerifyThis Competition 2021 [45], we used RG reasoning to verify ShearSort, a parallel sorting algorithm for matrices. The case study uses the **forall**‖ construct presented in Sec. 5.3 and also requires to solve the tricky problem of how to formalize the 0-1-principle, which is often used in informal proofs. A web presentation of the case study can be found at [32].

For some implementations thread-local verification of linearizability is difficult (including Herlihy and Wing's simple, but hard to verify queue implementation [26]), since their linearization points (LPs) are not fixed to specific steps of the thread itself, but are in steps of other threads and "potential": whether an LP has been passed depends on future execution. With colleagues we developed a complete approach [53] that uses a formalization of IO Automata [37] (using HOL only) in KIV that has recently been extended to an automatic translation of programs as steps of an IO automaton together with proof obligations generated from assertions [13].

The formalization of IO Automata provided one of the main motivations for adding polymorphism to KIV, IO-Automata are specified as data structures, with states and actions as parameters. Though the content of proofs does not change, using states and actions as type parameters instead of instantiating specifications (with e.g. "concrete" and "abstract" states for refinement, more instances are necessary to prove transitivity of refinement, or to define product automata) roughly halved the number of specifications required.

The most complex case study verified in this setting was the Elimination queue [40], an extension of the Michael-Scott queue [38] with an elimination array, that has very complex potential LPs. This case study, several others and a number of papers in this area can be found again online at [30].

Recently we also have looked at opacity [35] as the correctness criterion for STMs (implementations of software transactional memory, formalizing the theory as IO automaton refinement [3].

6.3 The Flashix Project

Flashix [4, 54] has been proposed as a pilot project for Hoare's Grand Challenge [27] with the goal to develop and verify a realistic file system for flash memory. The project motivated many enhancements in KIV, especially in component modularization and refinement methodology (cf. Sec. 4.3). To tackle the complexity we decomposed the file system into a deep hierarchy of components connected by 11 refinements. We generate C and Scala code from this hierarchy (we currently work on the generation of Rust code), the generated C code is about 18 kLOC and can be integrated into the Linux kernel or run via the FUSE interface.

A major focus in the project was to ensure *crash-safety*, i.e. the dealing with arbitrary power cuts. In [16] we added crash behavior to the semantics of our

components and proposed a verification methodology to prove crash-safety for modular refinement hierarchies.

The project required the use of our Separation Logic library (see Sec. 2.2) on multiple occasions. In the lower levels of Flashix, a verified pointer-based implementation of red-black trees is used (the KIV code and proofs can be found online [31]). There, we combined Separation Logic with KIV's concept of components and refinement (see Sec. 4.3) to "separate Separation Logic": the proof is split into the verification of a version based on algebraic trees, where the properties of red-black trees are verified, and a separate refinement that shows that copying branches on modifications can be avoided by replacing the algebraic tree with a pointer structure, that is updated in place. Only the latter, simple refinement has to deal with pointer structures, aliasing, and the avoidance of memory leaks by using Separation Logic formulas. Similarly, this concept was applied to the top-level refinement of Flashix, where the abstract representation of the file system as an algebraic tree is broken down to linked file and directory nodes.

The last phase of the project was mainly about introducing performance-oriented features like caching and concurrency. We added different caches to the file system, proposed novel crash-safety criteria for cached file systems, extended our refinement approach with corresponding proof obligations, and proved that the crash-safety criteria apply to the Flashix file system (see [44] and [5]). To add concurrency to the existing (purely sequential) refinement hierarchy, another kind of refinement was introduced: *atomicity refinement* [52] allows to incrementally reduce lock-based concurrent implementations to their sequential counterparts. This allowed to reuse large parts of the sequential proof work, in particular complex data refinements. The generation of proof obligations for atomicity refinements was implemented in KIV, they are based on rely-guarantee formulas (cf. Sec. 5.3), ownerships, and Lipton reductions [14, 36]. With this methodology, Flashix now allows concurrent calls to its toplevel interface and features concurrent Wear Leveling [52] as well as concurrent Garbage Collection [4].

7 Conclusion

This paper has given an overview of the concepts that are used in KIV to model and verify software systems. A lot of the support for software development was overhauled since [15] was published in 2014, starting with switching to Scala as KIV's implementation language. The basic logic has been extended to support polymorphism and exceptions in programs, rely-guarantee calculus has been revised to have native formulas and rules in favor of using abbreviations and general temporal rules. Support for components and for proof obligations has been significantly enhanced, and a code generator has been added that supports generation of Scala- and C-code.

Many of these developments have been motivated by the requirements to develop a realistic, concurrent file system for flash memory in the Flashix project. As an example, adding exceptions has uncovered several errors, that would have

gone unnoticed with the standard unspecified values semantics that is used in HOL, and that would also have been likely escaped testing, as they happened only on certain combinations of rare hardware errors (one was e.g. in formatting a flash device). Others, like adding assertions to programs or the forall-rule, have been motivated by the wish to be able to do smaller experiments, e.g. the challenges of the VerifyThis competition series, more quickly than before.

There is still lots of room for improvement though. Support for automating separation logic proofs is still far less developed than in native separation logic provers like Verifast [33] or Viper [42]. There is still lots of potential to optimize code generation from KIVs programs, using a careful dataflow analysis, where destructive updates and aliasing can be allowed in the generated C-code. The programming language is still (like Java) based on having programs ("statements") and expressions separately. We currently work on extending the language to have program expressions, where programs are the special case of program expressions of type unit. In the semantics this will generalize the void result $\zeta = \top$ on regular termination to a value of any type. This will allow methods with results and returns, and the use of Scala syntax for KIV programs (using Scalameta [49] for parsing). This will hopefully lead to increased expressiveness as well as an easier learning curve.

Acknowledgement We would like to thank our student Kilian Kotelewski who has added the forall-construct and its rules to the calculus.

References

1. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *LNCS*. Springer, 2016.
2. C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification*, pages 171–177. Springer, 2011.
3. E. Bila, S Doherty, B. Dongol, J. Derrick, G. Schellhorn, and H. Wehrheim. Defining and Verifying Durable Opacity: Correctness for Persistent Software Transactional Memory. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 39–58. Springer, 2020.
4. S. Bodenmüller, G. Schellhorn, M. Bitterlich, and W. Reif. *Flashix: Modular Verification of a Concurrent and Crash-Safe Flash File System*, pages 239–265. Springer, 2021.
5. S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Integration of Crashsafe Caching into a Verified Virtual File System Switch. In *Proc. of 16th International Conference on Integrated Formal Methods (IFM)*, volume 12546 of *LNCS*, pages 218 – 236. Springer, 2020.
6. E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 1 5(1–2):237–257, 2003.
7. E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence 11, pages 20–90. Elsevier, 1995.

8. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
9. L Cardelli. The Functional Abstract Machine. AT&T Bell Laboratories Technical Report. Technical report, TR-107, 1983.
10. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
11. W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.
12. J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. FACIT. Springer, 2001. second, revised edition 2014.
13. J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim. Verifying correctness of persistent concurrent data structures: a sound and complete method. *Formal Aspects Comput.*, 33(4):547–573, 2021.
14. T. Elmas, S. Qadeer, and S. Tasiran. A Calculus of Atomic Actions. In *Proceeding POPL 2009*, pages 2–15. ACM, ACM, 2009.
15. G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV: Overview and VerifyThis Competition. *Int. J. Softw. Tools Technol. Transf.*, 17(6):677–694, 2015.
16. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, Crash-Safe Refinement for ASMs with Submachines. *Science of Computer Programming*, 131:3 – 21, 2016. Abstract State Machines, Alloy, B, TLA, VDM and Z (ABZ 2014).
17. R. Goldblatt. *Axiomatising the Logic of Computer Programming*. LNCS 130. Springer, Berlin, 1982.
18. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
19. R. Hähnle, M. Heisel, W. Reif, and W. Stephan. An Interactive Verification System Based on Dynamic Logic. In *Proceedings of the 8th International Conference on Automated Deduction*, page 306–315. Springer, 1986.
20. D. Haneberg, S. Bäumler, M. Balser, H. Grandy, F. Ortmeier, W. Reif, G. Schellhorn, J. Schmitt, and K. Stenzel. The User Interface of the KIV Verification System — A System Description. In *Proceedings of UITP 05*, 2005.
21. D. Haneberg, N. Moebius, W. Reif, G. Schellhorn, and K. Stenzel. Mondex: Engineering a provable secure electronic purse. *International Journal of Software and Informatics*, 5(1):159–184, 2011. `http://www.ijsi.org`.
22. I. J. Hayes, C. B. Jones, and R. J. Colvin. Laws and semantics for rely-guarantee refinement. Technical Report CS-TR-1425, Newcastle University, 2014.
23. M. Heisel, W. Reif, and W. Stephan. Program Verification by Symbolic Execution and Induction. In *GWAI-87 11th German Workshop on Artifical Intelligence*, pages 201–210. Springer, 1987.
24. M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In *Logical Foundations of Computer Science*, LNCS 363, pages 134–145. Logic at Botik, Pereslavl-Zalessky, Russia, Springer, 1989.
25. M. Heisel, W. Reif, and W. Stephan. Tactical Theorem Proving in Program Verification. In *10th International Conference on Automated Deduction. Proceedings*, volume 449 of *LNCS*. Springer, 1990.
26. M.P. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

27. T. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. In *Modular Programming Languages*, pages 25–35. Springer, 2003.
28. Institut für Software & Systems Engineering - Universität Augsburg. Introduction and Setup of KIV. `https://www.uni-augsburg.de/en/fakultaet/fai/isse/software/kiv/`.
29. Institut für Software & Systems Engineering - Universität Augsburg. KIV Proofs of Starvation Freedom. URL: `https://kiv.isse.de/projects/Starvation-Free.html`.
30. Institut für Software & Systems Engineering - Universität Augsburg. Web Presentation of KIV Projects. `https://kiv.isse.de/projects/`.
31. Institut für Software & Systems Engineering - Universität Augsburg. KIV Proofs of Red-Black Trees, 2021. `https://kiv.isse.de/projects/RBtree.html`.
32. Institut für Software & Systems Engineering - Universität Augsburg. KIV Proofs of ShearSort, 2021. `https://kiv.isse.de/projects/shearsort.html`.
33. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *NASA Formal Methods*, 6617:41–55, 2011.
34. C. B. Jones, I. J. Hayes, and R. J. Colvin. Balancing Expressiveness in Formal Approaches to Concurrency. *Formal Aspects of Computing*, 27(3):465–497, 2015. `doi:10.1007/s00165-014-0310-2`.
35. Mohsen Lesani, Victor Luchangco, and Mark Moir. Putting opacity in its place, 2012. URL: `http://www.cs.ucr.edu/~lesani/downloads/Papers/WTTM12.pdf`.
36. R. J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, 1975.
37. N. Lynch and F. Vaandrager. Forward and Backward Simulations. *Information and Computation*, 121(2):214–233, 1995.
38. J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.
39. M.M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.
40. M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using Elimination to Implement Scalable and Lock-Free FIFO Queues. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, page 253–262. ACM, 2005.
41. B. Moszkowski and Z. Manna. Reasoning in Interval Temporal Logic. In *Logics of Programs*, pages 371–382. Springer, 1984.
42. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.
43. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Incorporation, 3rd edition, 2016.
44. J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Verification of Order-Preserving Write-Back Caches. In *IFM: 13th International Conference, 2017, Proceedings*, volume 10510 of *LNCS*, pages 375–390. Springer, 2017.
45. Programming Methodology Group - ETH Zürich. VerifyThis 2021. URL: `https://www.pm.inf.ethz.ch/research/verifythis/Archive/20191.html`.
46. W. Reif. Correctness of Generic Modules. In *Symposium on Logical Foundations of Computer Science*, LNCS 620, Berlin, 1992. Logic at Tver, Russia, Springer.
47. W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *COMPASS'95 – Tenth Annual Conference on Computer Assurance*. IEEE press, 1995.

48. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.
49. scalameta - Library to Read, Analyze, Rransform and Generate Scala Programs. URL: `https://scalameta.org/`.
50. G. Schellhorn. Completeness of ASM Refinement. *Electron. Notes Theor. Comput. Sci.*, 214:25–49, 2008.
51. G. Schellhorn and W. Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In *Automated Deduction — A Basis for Applications*, volume III: Applications, chapter 3: Automated Theorem Proving in Software Engineering, pages 165 – 194. Kluwer Academic Publishers, 1998.
52. G. Schellhorn, S. Bodenmüller, J. Pfähler, and W. Reif. Adding Concurrency to a Sequential Refinement Tower. In *Proc. of International Conference on Rigorous State-Based Methods (ABZ)*, volume 12071 of *LNCS*, pages 6–23. Springer, 2020.
53. G. Schellhorn, J. Derrick, and H. Wehrheim. A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures. *ACM Trans. Comput. Log.*, 15(4):31:1–31:37, 2014. URL: `https://doi.org/10.1145/2629496`.
54. G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a Verified Flash File System. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 9–24. Springer, 2014. Invited Paper.
55. G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, and W. Reif. RGITL: A Temporal Logic Framework for Compositional Reasoning about Interleaved Programs. *Annals of Mathematics and Artificial Intelligence*, 71, 2014.
56. G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. Interleaved Programs and Rely-Guarantee Reasoning with ITL. In *Proc. of the 18th Int. Symposium on Temporal Representation and Reasoning (TIME)*, IEEE Computer Society Press, pages 99–106, 2011.
57. G. Schellhorn, O. Travkin, and H. Wehrheim. Towards a Thread-Local Proof Technique for Starvation Freedom. In *Integrated Formal Methods IFM 2016*, volume 9681 of *LNCS*, pages 193–209. Springer, 2016.
58. B. Tofan, G. Schellhorn, and W. Reif. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *Theoretical Aspects of Computing – ICTAC 2011*, pages 239–255. Springer, 2011.
59. J. Woodcock, S. Stepney, D. Cooper, J. Clark, and J. Jacob. The Certification of the Mondex Electronic Purse to ITSEC Level E6. *Formal Aspects Comp.*, 20:5–19, 12 2008.
60. Q. Xu, W.-P. de Roever, and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects Comp.*, 9(2):149–174, 1997.