A Framework for the Interoperable Specification and Verification of Encapsulated Data Structures

Wolfram Pfeifer[1][0000−0002−9478−9641], Werner Dietl[2][0000−0002−9316−6952], and Mattias Ulbrich[1][0000−0002−2350−1831]

[1] Karlsruhe Institute of Technology
wolfram.pfeifer@kit.edu, mattias.ulbrich@kit.edu
[2] University of Waterloo
wdietl@uwaterloo.ca

Abstract. Well-designed data structures are fundamental to the construction of robust programs, particularly when their correctness can be established using formal methods. Currently, various successful deductive verification techniques exist but necessitate distinct and non-interoperable specifications and verification methods for data structure implementations. This paper presents a technique enabling cross-paradigm interoperable specification and verification of encapsulated data structures. The novel approach builds on the shared mathematical foundations of algebraic data types (ADTs) across these diverse methodologies. The technique enables the coherent integration of components that have been verified using different deductive program verification approaches within a single project, provided that a regime of encapsulation is followed, which is slightly stricter than those originally imposed by the verification techniques.
We formally introduce and discuss the encapsulation principle using a simplified conceptual object-oriented language and subsequently instantiate it for the Java programming language. This integrates the approaches of KeY, VeriFast, and Universe Types, thereby enabling heterogeneous verification projects in Java that encompass Dynamic Frames, Separation Logic, and Ownership Types.
We demonstrate the applicability of our approach by conducting a cooperative verification of a client with three different data structures, each of which is verified with one of the aforementioned techniques.

Keywords: Deductive Program Verification · Collaborative Verification · Interoperability of Formal Methods · Algebraic Data Types (ADTs)

1 Introduction

Significant progress in deductive program verification has been made over recent decades [24], both theoretically and in verification tools capable of analysing complex data structures and algorithms. Notable case studies include the verification of the TimSort implementation in Java libraries [23], the competitively efficient sorting routine IPSO [8], and the parallel nested depth-first search [43].

Nonetheless, prevailing approaches and tools tend to evolve in isolation, building on expressive and powerful formalisation and modelling concepts that lack compatibility with other verification techniques. This incompatibility exists not only at the language level but also conceptually; for instance, there is no clear method to integrate a program verified with a Separation Logic tool [45] with a client verified using Dynamic Frames [34].

In the domain of fully automatic software verification tools employing advanced static analysis techniques like property-driven reachability (PDR), k-induction, or bounded model checking, considerable progress has been made towards cooperative verification [46,3,25,11,9]. Standard exchange formats enable the sharing of artifacts [13], like witnesses and path invariants between different verification engines, thereby enhancing collaborative efforts. Witnesses have become integral to the SV-COMP verification competition [12].

Intensive cooperation between diverse tools is currently not available in heavyweight deductive program verification of sophisticated properties. A key reason is the required level of abstraction. Automatic verification operates close to the actual program state, whereas the functional verification of sophisticated properties requires modularisation, decomposition, and abstraction to subdivide tasks into more manageable units. The main issue here is the need to model memory, particularly the heap, and to reason about independent, encapsulated data units occupying disjoint memory areas. Current deductive techniques mandate that the entire program be verified with the same memory model within a single tool. Modern deductive tools rely on diverse, incompatible memory management philosophies, notably Dynamic Frames (e.g. KeY [1,7], Dafny) and Separation Logic (e.g. VeriFast[30,47], VerCors[14]). Furthermore, subtle differences in language design hinder the direct exchange of VeriFast SL specifications with those of VerCors.

As an example, consider the Java class `Client` in Lst. 1. It uses three data structures: a mutable `int` cell, a tree set of `int`s, and a linked list of `Cell` references. While Java verification tools such as KeY, VeriFast, or VerCors could be used to verify this project, they require verification of the entire project within a single tool. This is unfortunate, as the `Cell` library may have been previously verified with KeY, the linked list with VeriFast, and the tree set with an Ownership Types approach. To our knowledge, no unifying method exists that integrates verification results across such diverse memory modelling paradigms.

In this paper, we overcome this limitation and propose an interoperable specification and verification technique that enhances compatibility among various memory management ecosystems. It enforces stricter notions of *encapsulation* to ensure data structures are "well-behaved" for heterogeneous verification projects. We define encapsulation within this context and give sufficient criteria and methods to demonstrate it across the major memory management methodologies: *Separation Logic*, *Dynamic Frames*, and *Ownership Types*. Our approach facilitates the distribution of tasks according to the strengths of different techniques and enables the transfer of proven specifications for library code between tools.

```
1  class Client {                                  // Client: proven with DF + KeY
2    void m() {
3      Cell c0 = new Cell(); c0.set(5);            // Cell:   proven with DF + KeY
4      Cell c1 = new Cell(); c1.set(9)
5      LinkedCellList l = new LinkedCellList(); // List:   proven with VeriFast
6      l.add(c0); l.add(c1);
7      IntTreeSet s = new IntTreeSet();            // Set:    proven with UET + KeY
8      s.add(5); s.add(9);
9      Cell last1 = l.getLast();   int r1 = last1.value();
10     assert s.contains(r1);      assert r1 == 9;
11   }
12 }
```

Listing 1: Example of a client class which uses three different data structures. In particular, references to objects of type Cell are passed to LinkedCellList.

We formally introduce the encapsulation requirement in Sect. 3, applying it subsequently to Java. We refine the encapsulation concept for three relevant Java verification tools: Separation Logic in VeriFast (Sect. 4.1), Dynamic Frames in KeY (Sect. 4.2), and an Ownership Type System (Sect. 4.3). For KeY and VeriFast, we demonstrate that strict encapsulation can be guaranteed by adhering to certain specification patterns. Additionally, for the type system, we achieved interoperability by introducing a new type annotation, `payload` for objects that should not be part of the data structure, but are only passed around via opaque references. Each refinement includes a corresponding theorem stating that adhering to the specification schema guarantees encapsulation.

The approach has been prototypically implemented using Contract-LIB [21] as the exchange language for functional specifications (see Sect. 5). The example from Lst. 1 can be verified cooperatively across three tools. Fig. 1 illustrates the heterogeneity of the project.[3] The classes in the hatched boxes are automatically derived from language-independent contract specifications and used in the three verification frameworks. Sect. 5 elaborates more on this.

An extended version of this paper [44] is available, containing more details about the programming language model as well as proofs for the lemmas and theorems.

2 Related Work

As mentioned above, an active research area focuses on the interoperability of automatic verification tools. But also in the domain of deductive software verification, the call for collaboration has a long-standing tradition, shown by numerous community challenges [28,27,38,29] over the decades. However, examples of tool collaborations or case studies using heterogeneous approaches remain scarce. The *Karlsruhe Java Verification Suite* [35] enables collaboration through a subset of JML [37,16] assertions but, unlike our work, does not address encapsulated

[3] The full code of the example can be found in the artifact accompanying this paper (DOI: 10.5281/zenodo.18607514).

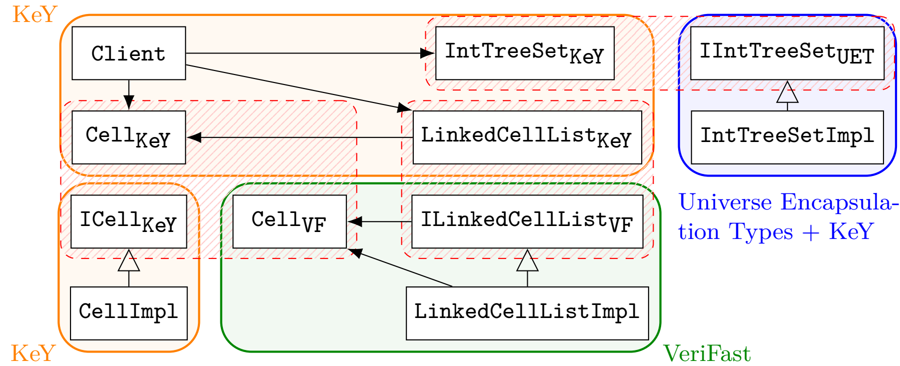

Fig. 1: Overview of the example, with the client from Lst. 1 in the top-left corner. Everything in the hatched red boxes is automatically generated from the corresponding Contract-LIB specifications. The subscripts indicate for which tool the specifications were generated. Filled arrows indicate usage in the code, open arrows denote implementation of an interface. The implementations are written manually and include the program code as well as additional specifications such as coupling invariants/predicates.

objects and relies on a direct memory model. Armborst et al. [2] propose a method to connect fully automatic and deductive software verification for C programs, with a focus on enabling tools from both verification types to collaborate by exchanging invariants, rather than integrating verification solutions across different program components as we aim to do.

Jakobs [31] presents a technique for constructing correctness witnesses from various partial analyses, aimed at integrating model checkers and static analysers that maintain explicit records of visited and checked states through abstract reachability graphs. Bao et al. [5] unify Separation Logic and Region Logic [4] into a single formal reasoning framework, showcasing reconciliation of different principles at the logical level. In contrast, our approach leverages algebraic data types (ADTs) as a common semantic foundation, facilitating interoperability of encapsulated components verified across paradigms without altering the underlying tools.

Unconstrained aliasing and mutations are causes of errors and complicate program reasoning. Object ownership and limitations on the effects of computations are well-studied proposed solutions [15]. Static and dynamic object ownership systems, in particular Universe Types, have been designed for program verification [20].

Intermediate verification languages (IVLs) such as Why3 [22], Boogie [39], Viper [41], and SV-LIB [10] decouple reasoning engines from source code languages, akin to intermediate representations in compilers. While IVLs effectively enable a shared backend for various verification engines, they only permit cooperation between tools that use compatible translations into the IVL logic.

3 Formal Foundations of Encapsulation

This section formally introduces the principle of data structure encapsulation in a conceptually reduced object-oriented programming language, focusing on the concepts essential for the notion of encapsulation. The definitions include deliberate simplifications and abstractions compared to real-world programming languages. In Sect. 4, we illustrate how insights gained from this abstract language can be transferred onto a concrete language—we chose Java—and be integrated with existing state-of-the-art verification frameworks.

3.1 Model of an Object-Oriented Programming Language

There are five mutually disjoint infinite sets of names: $\mathcal{F}$ for field names, $\mathcal{L}$ for local variable names, $\mathcal{P}$ for procedure names, $\mathcal{Q}$ for query names, and $\mathcal{C}$ for class names. In addition, $\mathcal{O}$ denotes the infinite set of object references, which is disjoint from the name sets. The set of *values* contains at least the integers, booleans, and object references: $\text{Val} = \mathbb{Z} \cup \mathbb{B} \cup \mathcal{O} \cup \ldots$. This leaves room for values of other types when instantiating the framework with concrete languages.

We need a memory model that is general enough to entail those of existing verification methodologies in an object-oriented context, such as Dynamic Frames and Separation Logic. Therefore, we define a *location* (o, f) as a pair of an object reference $o \in \mathcal{O}$ and a field name $f \in \mathcal{F}$. $\text{Loc} = \mathcal{O} \times \mathcal{F}$ is the set of all locations. These locations are then used as indices of memories: A *memory* $M : \text{Loc} \to \text{Val}$ is a mapping from locations to values. Mem denotes the set of all memories. Note that M is a total function: We assume that there is always *some* value defined for each location. With this general memory model, we can define objects and their operations. We assume that there are two clearly distinguished types of operations. A *procedure* $p : \text{Mem} \times \text{Val} \to \text{Mem}$ is a function that takes a memory and a value and returns a memory, but not a value. A *query* $q : \text{Mem} \times \text{Val} \to \text{Val}$ is a function that takes a memory and a value and returns a value, but not a memory. This is a simplification, but mixed operations could always be split up into a sequence of procedures and queries. In addition, for keeping the presentation simple, operations only have a single value argument. Again, this is not a conceptual limitation, since it could be easily overcome by using Godelization or introducing tuples into the values.

Definition 1 (Object). *An object is a tuple* $o = (\mathbb{F}_o, \mathbb{P}_o, \mathbb{Q}_o)$ *of a finite set of field names* $\mathbb{F}_o \subseteq \mathcal{F}$ *and finite partial maps from procedure names to procedures* $\mathbb{P}_o : \mathcal{P} \nrightarrow (\text{Mem} \times \text{Val} \to \text{Mem})$ *and query names to queries* $\mathbb{Q}_o : \mathcal{Q} \nrightarrow (\text{Mem} \times \text{Val} \to \text{Val})$*. The set of all objects is* Obj*.*

Note that this notion of an object is more general than usual in programming languages. For example, a `struct` in C together with functions working only on it could also be an object according to our definition. So far, we do not have a possibility to track created objects, or create new ones. Inspired by virtual function tables in OO languages, we introduce *vTables* and constructors for that. A *vTable* $T : \mathcal{O} \nrightarrow \text{Obj}$ is a finite partial map from object references to objects.

The set of all vTables is $\mathcal{T}$. Note that we do not show how classes and object creation are modelled, but just assume that there is a way to create objects, and the objects created match the source code (for instance, they only have the operations defined in their class). More details can be found in the extended version [44]. In addition to normal objects, there are instances of data structures, which we model as a special case of objects.

Definition 2 (Data Structure Instance (DSI)). *A* data structure instance *(DSI) is a special object $d = (\emptyset, \mathbb{P}_d, \mathbb{Q}_d)$ without (public) fields.*

This notion serves as the basic building block for defining encapsulation. Intuitively, a DSI is an entity supposed to be encapsulated, and can thus be abstracted more than a normal object. Having no public fields is necessary for encapsulation; however, it is not sufficient, and we still need to prove that. This brings us to the definition of a footprint, which intuitively is the set of locations that (semantically) "belong" to a DSI in memory. We split the footprint into a read and a write part, and determine them via queries and procedures.

Definition 3 (Footprints). *The read and write footprint* $\mathrm{rfp}(d, M), \mathrm{wfp}(d, M) \subseteq \mathrm{Loc}$ *for a DSI $d = (\emptyset, \mathbb{P}_d, \mathbb{Q}_d)$ in a memory $M \in \mathrm{Mem}$ are defined as:*

$$\begin{aligned}\mathrm{rfp}(d, M) := \big\{ l \in \mathrm{Loc} \mid\ & \exists n \in \mathbb{N}, \bar{p} \in \mathbb{P}_d^n, q \in \mathbb{Q}_d, v \in \mathrm{Val}, a \in \mathrm{Val}, \bar{a} \in \mathrm{Val}^n \\ & q(p_1(p_2(...(p_n(M \qquad\qquad , a_n), ...), a_2), a_1), a) \\ & \neq q(p_1(p_2(...(p_n(M[l \mapsto v], a_n), ...), a_2), a_1), a) \big\} \\ \mathrm{wfp}(d, M) := \big\{ l \in \mathrm{Loc} \mid\ & \exists p \in \mathbb{P}_d, a \in \mathrm{Val}\, .\ p(M, a)(l) \neq M(l) \big\} \\ \mathrm{fp}(d, M) :=\ & \mathrm{rfp}(d, M) \cup \mathrm{wfp}(d, M)\end{aligned}$$

The notation $M[l \mapsto v]$ denotes a function update, that is $M[l \mapsto v](l) = v$ and $M[l \mapsto v](x) = M(x)$ for all $x \neq l$.

A location is in the rfp of a DSI in a memory M, iff changing the location can make a difference in a later observed state of the DSI. Note that it is not sufficient to look at only a single procedure at a time, since the state change could be hidden and only have an effect on a query result much later. Therefore, we need to consider chains of procedure applications. A location is in the wfp of a DSI, iff it can be changed by any of its procedures.

We refrain from giving formal definitions for programs and their semantics for space reasons; these can be found in the extended version [44]. The intuition is, however, that it is sufficient to look at sequences of atomic statements (inspired by computation sequences in literature [26, Sec. 5.3]). In particular, our program model contains no control structures. In addition, the statements do not contain method calls on non-DSI objects. Our programs can be thought of as having all statements outside of a DSI "inlined". For DSIs, on the other hand, we stop this "inlining" when we first reach one of their methods from the outside. This gives us a clear separation between the outside world of the rest of the program, and the DSIs that are supposed to be encapsulated (which will be proven later) and

can thus be abstracted further. It suffices to consider only these restricted linear programs in our model, because later we always quantify over all terminating programs (for fixed code of the DSI under examination).

For understanding the definition of encapsulation, however, we need a notion of program state: A *program state* $s = (V, M, T) \in S$ is a triple of a partial function $V : \mathcal{L} \nrightarrow \text{Val}$ of local variable names to values, a memory $M : Loc \to \text{Val}$, and a vTable $T : \mathcal{O} \to \text{Obj}$. We define a program $P = (s_0, \sigma)$ as a pair of a starting state s_0 and a sequence σ of atomic statements, and denote the set of all states reached during the execution of P by $R(P)$.

3.2 Reachable Locations and Encapsulation

Since we conceptually "inline" object procedures and queries, we define their variant of a footprint as the locations that are reachable via chained field accesses.

Definition 4 (Reachable Locations, Relevant Locations). *We define the set of locations reachable from an object o in a memory M as the smallest fixpoint of the following recursive function (*fst *selects the first element of a tuple):*

$$\mathtt{reachLocs}(o, M) := \begin{cases} \bigcup_{f \in \text{fst}(o)} \{(o, f)\} \;\cup \mathtt{reachLocs}\big(M(o, f), M\big) & (o \in \mathcal{O}) \\ \emptyset & (o \notin \mathcal{O}) \end{cases}$$

The relevant locations *of an object o in a memory M are defined as follows:*

$$\mathtt{relLocs}(o, M) = \begin{cases} \text{fp}(o, M) & \textit{if } o \textit{ is a DSI} \\ \mathtt{reachLocs}(o, M) & \textit{if } o \textit{ is not a DSI} \end{cases}$$

Note that for convenience, we define the function `reachLocs` also for values that are not object references (for instance integers), which makes the definition significantly shorter and easier to read.

Intuitively, *reachLocs* of an object o contains all the fields that are either fields of the object directly, or recursively reachable after following one of the references stored in one of the fields. The definition of *relLocs* avoids the need of more case distinctions later.

Definition 5 (Encapsulation). *A DSI d is called* encapsulated *if for all programs in all possible states of the program the footprint of d is disjoint from the relevant locations of objects referenced in local variables:*

$$\forall \text{ Program } (s_0, \sigma).\ \forall \text{ State } (V, M, T) \in R(s_0, \sigma).\ \forall \text{ Object } o.$$
$$o \in dom(T) \land o{\neq}d \land (\exists v \in dom(V).\ V(v) = o) \to \text{fp}(d, M) \cap \mathtt{relLocs}(o, M) = \emptyset$$

Our central definition intuitively states that external references cannot access the footprint of an encapsulated DSI, nor can any external location be reached from within the DSI. Its procedures and queries serve as an interface, allowing

the state of the DSI to be queried and modified solely through this interface. Note that it is sufficient to consider references stored in local variables, since (reachable) references from the heap could be obtained and immediately stored in those. Since in the definition there is a quantifier ranging over all programs, this also includes the program where this heap read and assignment to a local variable happens. Thus, the condition also indirectly covers any objects that are referenced in fields on the heap.

4 Encapsulation in Java

The formalizations in the previous section are agnostic to the concrete programming language, and only require some basic object-oriented features to be present. We instantiate the framework now for Java, and demonstrate how it ties in with different verification methodologies. The rather abstract DSI from the previous section is more concrete here: It is, in the simplest form, a Java object of a data structure class. We support a subset of the Java language. In the spirit of soundiness [40], we do not support Java features that involve dynamic change to the classes and code available, in particular reflection, `invokedynamic`, and code generation at runtime. Furthermore, we do not support methods declared as `native` and the Java Native Interface (JNI). However, these are restrictions commonly made in static program analysis. More importantly, we do not support subclassing—therefore, all classes are final—and generics at the moment, although we are confident that our methodology can be expanded to them. We require that all fields of data structure classes are private, which fits together with their abstraction (DSI) not having any public fields. For the sake of presentation, we ignore other visibility modifiers (`protected`, package private). In addition, we require that there are no unchecked downcasts in the program that could lead to runtime exceptions. Finally, for readability, we assume that procedures have only a single parameter, and constructors and queries do not have any. This is only for presentation, not a conceptual restriction.

In this section, we present schemata for the three verification and framing methodologies Separation Logic, Dynamic Frames, and Ownership Types. For each of them, we provide a theorem stating that if the data structure implementation adheres to the schema, all objects of the data structure are encapsulated according to Def. 5. For space reasons, we do not show proofs for the theorems and lemmas in this section, but instead refer to the extended version [44].

4.1 Separation Logic (VeriFast)

Separation Logic [42] (SL) is a deductive verification disclipline with a rather strict memory regime: Only those locations whose values have been explicitly captured in the preconditions are allowed to be read from or written to. This paradigm enforces strict memory separation, as the same locations must never be part of the memory footprint of more than one DSI at a time. SL hence enforces a strict notion of encapsulation—if the entire code base adheres to SL proofs. We

will elaborate what this entails for approach-spanning verification attempts and describe a pattern that must be obeyed to guarantee encapsulation if not the entire code base adheres to SL. The challenge here is to remain open to “plugging in” data structures verified with other techniques.

We focus on the SL variant with permissions supported by VeriFast [30], which does not support quantifiers and magic wand in formulas. Data structure abstraction in SL is usually done using *abstraction predicates* which take a reference to the data implementation and the abstract value, capture the memory locations of the footprint of a DSI, and hold if the value computed from the data in the data structure is the same as the abstract argument value.

In interoperable contexts where some program components do not conform to SL, a significant challenge arises: an object may reference other objects in different data structures that should not be externally accessed. If the program were verified in one SL framework, access clauses for the relevant memory locations would be required, but these clauses cannot be generated. Without SL rigor, arbitrary Java code can access fields and traverse references, bypassing the abstract predicates defined by SL. The solution is to identify abstract predicates that include *all* reachable memory locations, thereby preventing access to prohibited memory.

Definition 6 (Deep Abstraction Predicates). *An abstraction predicate p for a class C is called* deep *if it refers to all fields of C and asserts a deep abstraction predicate for every field of reference type.*

Deep abstraction predicates represent heap chunks maximal wrt. reachability.

We introduce an abstract predicate `payload(x)` that captures the locations reachable from the Java object `x` via field dereferences. The predicate does not have a definition and is therefore intentionally underspecified: it exposes no concrete points-to facts and yields no functional guarantees about `x`. Its purpose is framing: with separating conjunction, $\texttt{payload}(x) * R$ asserts that R must be disjoint from the footprint of x. Because `payload` does not have a definition, clients cannot create initial instances. We therefore provide an introduction axiom `payloadCreate`, permitting `payload(x)` to be asserted without preconditions. This grants the `payload` token (for disjointness/framing), not any concrete field permissions. It reflects the ability of a client to pass arbitrary references into data structures as payloads. For soundness, we need to make sure that it is only used in clients, not in the data structure itself, which can be done with a simple syntactic check.

The encapsulation schema for a data structure class `C` is as follows (a template is shown in Lst. 2):

1) There is a deep abstraction predicate for `C` (`absC` in the example) with a parameter of type `C` and a second one of an immutable type (usually an ADT). It connects the heap-related type `C` to its heap-independent abstraction.
2) Each constructor establishes well-formedness with a postcondition `absC(this,` ϕ_1`())`, where ϕ_1 is an expression of the ADT type.
3) Each method requires and establishes the well-formedness of the data structure with a precondition `absC(this, ?x)` (the operator “`?`” binds `x` in the

```
1*  /*@ predicate absC(C c, <adt> a) = ...; @*/
2   class C {
3     C()
4       //@ requires true;
5*      //@ ensures absC(this, ϕ1());
6     { ... }
7
8     <ret_type> query()          // obtains payload ref. from data structure
9       //@ requires absC(this, ?x);
10*     //@ ensures absC(this, ϕ2(x)) &*& payload(result);
11    { ... }
12
13    void procedure(<param_type> p) // stores payload ref. in data structure
14      //@ requires absC(this, ?x) &*& payload(p);
15*     //@ ensures absC(this, ϕ3(x, p));
16    { ... }
17  }
```

Listing 2: Schematic example for encapsulation with VeriFast Separation Logic. The ϕ_i are placeholders for expressions of the ADT type, possibly depending on the given parameters as well as fields of this. The $*$ indicates lines dealing with functional properties covered in Sect. 5.

contract) and a postcondition `absC(this,` ϕ_i`(x))`, where ϕ_i is an expression of ADT type possibly depending on the input parameters and fields.

4) Each procedure receiving a parameter `p` adds a precondition `payload(p)`.
5) Each query returning a payload adds a postcondition `payload(result)`.

An observation is that an abstraction predicate of a DSI carries a set of permissions to memory locations, which is a superset of its footprint.

Definition 7 (Permissible Locations). *For an SL formula ϕ, we define the set $\texttt{pLocs}(\phi, d, M)$ as:*

$$\texttt{pLocs}(\phi, d, M) := \begin{cases} \emptyset & \phi \in \{\textbf{\textit{false}}, \textbf{\textit{true}}, \textbf{\textit{emp}}\} \\ \{(d, f)\} & \phi = f \mapsto y \\ \bigcup_{\psi} \texttt{pLocs}(\psi, d, M) & \textit{otherwise (with } \psi \textit{ subformula of } \phi) \end{cases}$$

Here, ***emp*** *denotes the SL formula that states that the heap is empty, and $f \mapsto y$ ("points-to") expresses that the heap is a single cell with address f and value y.*

We assume that there is exactly one abstraction predicate α_d defined for each DSI d and use $\texttt{pLocs}(d, M)$ as a shorthand for $\texttt{pLocs}(\alpha_d, d, M)$.

Lemma 1. *The set $\texttt{pLocs}(d, M)$ of a DSI d is a superset of its footprint* $\mathrm{fp}(d, M)$*:*
$\forall\ \mathrm{DSI}\ d.\ \forall\ M \in \mathrm{Mem}\,.\ \texttt{pLocs}(d, M) \supseteq \mathrm{fp}(d, M)$

Theorem 1 (Encapsulation with the VeriFast Separation Logic Schema). *A verified DSI d that adheres to the above schema is encapsulated. That is, the permissible locations of the DSIs are disjoint from the relevant locations of all*

objects referenced in local variables:

$$\forall \text{ Program } (s_0, \sigma).\ \forall \text{ State } (V, M, T) \in R(s_0, \sigma).\ \forall \text{ Object } o.$$
$$o \in dom(T) \wedge o{\neq}d \wedge (\exists v \in dom(V).\ V(v){=}o) \rightarrow \mathtt{pLocs}(d, M) \cap \mathtt{relLocs}(o, M) {=} \emptyset$$

The theorem is proven by induction on the program length (for details see [44]). With Lemma 1, this gives us that the DSI is encapsulated.

4.2 Dynamic Frames (KeY)

Dynamic Frames [34,48] provide a flexible methodology for specifying and reasoning about shared structures. However, this flexibility entails significant specification overhead. Additionally, proofs can be challenging due to the need for extensive reasoning about potentially overlapping memory locations.

We present the conditions for encapsulation with Dynamic Frames in the syntax of the Java Modeling Language (JML) [37,16], more specifically in the JML dialect of the Java verifier KeY [1,7]. We present two variants of the schema for encapsulation. First, we show the original pattern known from literature [33,48], where only Dynamic Frames are used for all classes. Second, we present a generalized version that is suitable for our goal of combining multiple different specification and reasoning methodologies.

Pure Dynamic Frames With only Dynamic Frames, an encapsulated data structure has to adhere to the following schema (a template class is shown in Lst. 3):

1) There is a virtual (ghost) field of type locations set (`\locset`), which at least contains all fields of `this` (`this.*`). It is defined manually by the user to contain all heap locations which "belong" to the data structure.
2) The footprint locations are freshly created by each constructor (`ensures \fresh(fp)`), and no existing locations are written (`assignable \nothing`).
3) Procedures and constructors write only to locations in `fp` (`assignable fp`) or add freshly created objects to it (`ensures \new_elems_fresh(fp)`)[4]. This ensures that references of objects passed as parameters are not captured.
4) Query results depend only on the footprint (`accessible fp`).
5) No query can leak pointers into the data structure (`ensures \disjoint(fp, \result.fp)`).
6) Nothing is leaked or captured via any method parameter `p` of reference type (`ensures \disjoint(fp, p.fp)`). For proving, we usually need to know that this also holds in the pre-state (`requires \disjoint(fp, p.fp)`).

Hybrid approach for Dynamic Frames combined with other techniques To enable collaboration with other verification techniques, we cannot assume that each class has a clearly defined footprint field. Therefore, we require alternative methods to express the disjointness conditions (points 5/6 above). We introduce the function `\reachLocs`(o) on JML level, which has the semantics of *reachLocs* as

[4] This is known as the *swinging pivot condition* in literature [32].

```
1    class C {
2      //@ ghost \locset fp;
3      //@ invariant fp == this.* ∪ ...
4*     //@ ghost <adt> absVal;
5
6*     //@ ensures \fresh(fp) && φ1(absVal);
7      //@ assignable \nothing;
8      C() { ... }
9
10     //@ ensures \disjoint(fp, \result.fp);          // ⟵ \reachLocs(\result)
11*    //@ ensures φ2(\result, absVal);
12     //@ assignable \strictly_nothing;
13     //@ accessible fp;
14     <ret_type> query() { ... }
15
16     //@ requires \disjoint(fp, p.fp);               // ⟵ \reachLocs(p)
17     //@ ensures  \disjoint(fp, p.fp);               // ⟵ \reachLocs(p)
18*    //@ ensures \new_elems_fresh(fp) && φ3(absVal, \old(absVal));
19     //@ assignable fp;
20     void procedure(<param_type> p) { ... }
21   }
```

Listing 3: Schematic example for encapsulation with Dynamics Frames. When Dynamic Frames shall be used in cooperation with other techniques, the underlined parts can be replaced as indicated with the expressions in the comments to obtain the hybrid schema. The $*$ indicates lines dealing with functional properties (expressed by the placeholder formulas ϕ_i) covered in Sect. 5.

defined in Def. 4 applied to the current heap state. The location set returned by `\reachLocs` is an overapproximation of the actual footprint, representing the "worst possible" footprint of the data structure while adhering to the programming language's capabilities, including visibility modifiers. With the `\reachLocs` function, we can now formulate the conditions for a data structure to be encapsulated (a template is shown as an alternative in Lst. 3):

1-4) These conditions are the same as in the pure Dynamic Frames technique.

5) No query can leak pointers into the data structure (`ensures \disjoint(fp, \reachLocs(\result))`).

6) Nothing is leaked or captured via any method parameter `p` of reference type (`ensures \disjoint(fp, \reachLocs(p))`). For proving, we usually need to know that this also holds in the pre-state (`requires \disjoint(fp, \reachLocs(p))`).

Lemma 2. *For a DSI d adhering to the constraints above, the footprint field holds a superset of* $\mathrm{fp}(d, M)$*:* $\forall$ DSI $d.\ \forall \mathrm{Mem}\ M.\ M(d, fp) \supseteq \mathrm{fp}(d, M)$

Theorem 2 (Encapsulation with the Hybrid Dynamic Frames Schema). *If a DSI d specified with the hybrid Dynamic Frames schema is verified successfully in KeY, it is encapsulated. That is, the locations stored in the virtual footprint field* fp *of the DSI are disjoint from the relevant locations of all objects referenced*

in local variables:

$$\forall \text{ Program } (s_0, \sigma).\ \forall \text{ State } (V, M, T) \in R(s_0, \sigma).\ \forall \text{ Object } o.$$
$$o {\in} dom(T) \wedge o {\neq} d \wedge (\exists v \in dom(V).\ V(v) = o) \rightarrow M(d, fp) {\cap} \mathtt{relLocs}(o, M) {=} \emptyset$$

To give an intuition, this means that all the outside objects cannot have a reference into the data structure.

The proofs of Lemma 2 and Theorem 2 are very similar to the SL case, and can be found in the extended version [44]. Combined, we obtain the guarantee that the DSI is encapsulated if it adheres to either variant of the DF schema.

4.3 Ownership/Universe Types

In contrast to Separation Logic and Dynamic Frames, which combine reasoning about memory with reasoning about functional properties, Ownership Types only give guarantees about memory. Therefore, it is necessary to complement them with an additional technique for functional reasoning. One advantage of this is that the memory checks are done by the type system independently, and can then be assumed in the functional verifier, instead of being mixed-in with the rest of the verification. In our case, we use a type checker for a variant of Universe Types[5], implemented using the Checker Framework [17], in collaboration with KeY with simplified proof obligations only for the functional part[6].

Universe Encapsulation Types (UET) We define the new type and effect system *Universe Encapsulation Types (UET)*, which builds upon the existing Universe Types [18,19], but replaces their `any` types with `payload` types to obtain stronger guarantees. The type system adds a second dimension to the standard Java reference types. A type in UET is a pair (U, T), where T is the standard Java type and $U \in \{\mathtt{rep}, \mathtt{peer}, \mathtt{payload}\}$ is an ownership modifier. The subtype relation is defined element-wise: $(U_s, T_s) \leq (U, T)$ iff $U_s \leq_{\mathrm{UET}} U$ and $T_s \leq_{\mathrm{Java}} T$. Here, $\leq_{\mathrm{Java}}$ is the standard Java subtype relation, and $\leq_{\mathrm{UET}}$ is the relation shown in Fig. 2.

Universe Types classify objects into contexts, allowing an object to read or modify everything in its context (directly or via methods). Contexts are indicated by ownership modifiers. The modifier `rep` denotes that the referenced object is (directly) owned by `this`, while `peer` indicates that the object shares the same owner as `this`. We replace the UT modifier `any` with `payload`, enhancing guarantees from the type system by prohibiting method calls or de-referencing of them (only equality comparison is allowed), in contrast to calls to `pure` methods in UT. They are treated as opaque, not depending on the state of underlying objects, allowing for storage and passing around. Similar to other type and effect systems, such as Universe Types, there are annotations in addition to the type modifiers. In UET, these are `DSI`, a marker for classes subject to encapsulation

[5] https://github.com/WolframPfeifer/universe/tree/pfeifer/encapsulation
[6] https://github.com/KeYProject/key/tree/pfeifer/universeEncapsulation

```
1   @DSI class C {
2*    //@ ghost <adt> absVal;
3     @Rep <field_type> f;
4
5*    //@ ensures φ1(absVal);
6     @RepOnly C() { ... }
7
8*    //@ ensures φ2(result, absVal);
9     @RepOnly @Payload <ret_type> query() { ... }
10
11*   //@ ensures φ3(absVal, \old(absVal, p));
12    @RepOnly void procedure(@Payload <param_type> p) { ... }
13  }
```

Listing 4: Schematic example for encapsulation with UET. The functional part is expressed in JML and can be proven with KeY. The $*$ indicates lines with functional properties (see Sect. 5), and the ϕ_i stand for formulas expressing these properties.

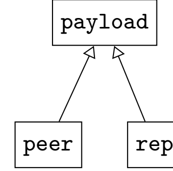

Fig. 2: Type hierarchy of UET.

checking, and `RepOnly`, which denotes a method that operates solely on the `rep` footprint of the DSI. This is ensured by allowing assignments only to fields of `this`, and method calls where the receiver is `this` or a `rep` reference.

Encapsulation with UET We can now use this type system for proving encapsulation. However, well-typedness alone is not sufficient. Instead, as for Separation Logic and Dynamic Frames, the DSI needs to comply to a specific schema. A DSI o (marked as such by the user with the corresponding annotation) of a class C is encapsulated if it adheres to the following schema, and it is UET well-typed (a template class following the schema is found in Lst. 4):

1) All methods and constructors in class C are `repOnly`.
2) All fields of reference type in C are `rep` or `payload`.
3) All parameters (of reference type) of methods are `payload`.
4) All (reference) return types of methods are `payload`.

Note that the internal classes of the DSI (for instance a class `Node` that is used internally in an implementation of `List`) do not have these restrictions. There, it is for example also allowed to have `peer` references and methods that operate on peers. Only the top-level "interface" class has to comply to the schema, internally, the full expressive power of the original Universe Types can be used.

Definition 8 (Rep Footprint). *The* rep footprint $\texttt{repFP}(d, M)$ *of a DSI d is the set of locations that are directly or transitively owned by d. It is the smallest fixpoint of the following function, where x is used as a shortcut for $M(d, f)$:*

$$\texttt{repFP}(d, M) := \bigcup_{f \in \mathrm{fst}(d)} (d, f) \cup \bigcup_{\substack{f \in \mathrm{fst}(d), \\ f \text{ is } \texttt{rep}}} \Big(\bigcup_{g \in \mathrm{fst}(x)} (x, g) \cup \bigcup_{\substack{g \in \mathrm{fst}(x), \\ g \text{ is not } \texttt{payload}}} \texttt{repFP}(M(x, g), M) \Big)$$

Lemma 3. *For a DSI d adhering to the constraints above, the rep footprint of d is a superset of* $\mathrm{fp}(d, M)$*:* $\quad \forall\ \text{DSI}\ d.\ \forall\, \text{Mem}\, M.\ \texttt{repFP}(d, M) \supseteq \mathrm{fp}(d, M)$

Theorem 3 (Encapsulation with Universe Encapsulation Types). *If a DSI d is annotated with UET, and checked with the UET checker, it is encapsulated. That is, the rep footprint of the DSI is disjoint from the relevant locations of all objects referenced in local variables:*

$$\forall \text{ Program } (s_0, \sigma).\ \forall \text{State } (V, M, T) \in R(s_0, \sigma).\ \forall \text{ Object } o.$$
$$o \in dom(T) \land o \neq d \land (\exists v \in dom(V).\ V(v)=o) \rightarrow \mathtt{repFP}(d, M) \cap \mathtt{relLocs}(o, M) = \emptyset$$

As with SL and DF, for the proofs we refer to the extended version [44], and obtain encapsulation of the DSI with Lemma 3.

5 Cooperation Between Verification Techniques

Up to here, we have demonstrated the verification of encapsulated data structures within a single verification framework. However, the strict encapsulation notion aims to verify heterogeneous client programs with parts verified using different verification tools. Achieving cooperative deductive verification also requires the exchange of functional specifications beyond framing across different tools. The programming language in Sect. 4 is consistently Java, such that code can be readily shared between tools. However, sharing functional specifications is challenging due to significant conceptual differences between the specification languages of KeY, VeriFast, and Universe Types.

The presented approach mainly focuses on integrating various seemingly incompatible memory management approaches, such that we only briefly outline how functional properties can be shared across verification paradigms.

Despite diverse specification mechanisms, many formalisms share a common foundational concept: *Algebraic Data Types* (ADTs). Rooted in the mathematical notion of inductive structures, ADTs are recognised across many verification methods, sharing identical semantics (namely that of free generated algebras). As encapsulated DSIs possess an independent state unaffected by external operations, ADTs are an ideal means to represent abstract values in encapsulated data types.

Contract-LIB [21] is a programming-language-agnostic interface specification language allowing the specification of data abstractions and operations via contracts. This makes it suitable for data exchange between various verification approaches, as it enables the declaration of a data abstraction for the data structure along with contracts that express the effects of operations in terms of these abstractions. Algebraic and co-algebraic data types, either as built-in types of SMT-LIB [6] or defined using `declare-datatypes`, are supported as types for abstract data. The exchange language Contract-LIB and the specification schemata in Lsts. 2, 3, and 4 allow us to verify well-encapsulated data structures across different verification paradigms, using the following workflow:

1) Data structures and their operations are specified via ADTs in Contract-LIB.
2) For each data structure, one verification approach from Sect. 4 is selected.
3) The Contract-LIB specifications are used to generate a *verification interface* containing the translated specifications in the respective target language.

```
; Abstractions
(declare-abstractions
    ((example.Cell 0))
    (((Cell (absVal Int)))))

(declare-abstractions
    ((example.LinkedCellList 0))
    (((LinkedCellList (absVal (Seq (Ref Cell)))))))

; Contracts
(define-contract example.LinkedCellList.init      ; constructor
    ((result (out LinkedCellList)))
    ((true
     (= (absVal result) seq.empty))))

(define-contract example.LinkedCellList.add       ; procedure
    ((this (inout LinkedCellList))
     (v (in (Ref Cell))))
    ((true
     (= (absVal this) (seq.++ (old (absVal this)) (seq.unit v))))))

(define-contract example.LinkedCellList.getLast   ; query
    ((this (inout LinkedCellList))
     (result (out (Ref Cell))))
    (((not (= (absVal this) seq.empty))
     (= result (seq.nth (absVal this) (- (seq.len (absVal this)) 1))))))
```

Listing 5: Specification of LinkedCellList in Contract-LIB.

4) The manually written implementations are then verified against the interfaces.
5) The Contract-LIB specifications can also be used to produce *verification stubs*, verification-only classes bearing the same specifications that can be used on the client side to verify against them.

If we come back to the initial heterogeneous code example from Lst. 1, the overview in Fig. 1 shows how the Contract-LIB specification e.g. for the class `LinkedCellList` (as shown in Lst. 5), induces the verification interface $\texttt{ILinkedCellList}_{\texttt{VF}}$ providing the specification for the implementation in the class `LinkedCellListImpl`. Separately from verification of the data structure in VeriFast, the class is used in the context of the client, hence the verification stub $\texttt{LinkedCellList}_{\texttt{KeY}}$ is produced for client verification in KeY.

We implemented automation support for Contract-LIB in a framework named Contract-Chameleon[7], which encompasses a parser and a translator to/from the specification languages used by KeY (JML with Dynamic Frames) and VeriFast (permission-based Separation Logic). Contract-Chameleon also generates stub Java files within which the actual implementation and auxiliary specifications (such as coupling and loop invariants) can be integrated.

As established by Theorems 1, 2, and 3, verification entails encapsulation. Thus, it is sound to translate specifications from one language to another and employ the appropriate one for the client's verification proof. The functional guarantees are transported via abstraction predicates (SL) and ghost fields (DF

[7] https://github.com/Contract-LIB/contract-chameleon

and UET), as well as through the method contracts that describe how the abstract state evolves with each operation. In the schematic examples, this can be found in the lines marked with $*$ in Lsts. 2 to 4, where the expressions/formulas $\phi_1, \ldots, \phi_4$ describe the effects of the operations in terms of the abstract value. With the guarantee of encapsulation, it allows us to exchange the functional specifications soundly between the different verification methodologies.

6 Future Work and Conclusion

The encapsulation patterns discussed in this paper are restrictive. Some data structures may *behave* encapsulated, while sharing memory internally, for example, copy-on-write lists. We plan to extend our approach towards *observational encapsulation* to support such cases. Other structures are only partially encapsulated, such as a tree-set that uses hash codes for logarithmic-time operations, while remaining functionally indistinguishable from an abstract set. Techniques similar to declassification in information-flow verification may address this. Future work also includes support for subclassing and parametric polymorphism to make `payload` annotations of UET more useful. Combining UET with deductive verification [36] could further improve the expressiveness and usability of the system. We plan to extend current prototype tools, for example, to import VeriFast specifications into Contract-LIB, to support other Separation Logic variants (most notably the one supported by VerCors), and with a dedicated proof management tool.

To sum up, in this paper, we formalized encapsulation, and provided sufficient conditions in three common verification methodologies: Separation Logic (for the tool VeriFast), Dynamic Frames (KeY), and an Ownership Type system. We have proven that they are sufficient for each of the methodologies. In addition, we described how functional guarantees can be exchanged via Contract-LIB. We demonstrated the applicability of our framework by conducting a cooperative verification effort of a small example with three data structures, each verified with one of the methodologies.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback on this paper. This work was supported by the DFG projects BE 2334/9-1, UL 433/3-1 and the KIT International Excellence Fellowships Program with funds granted to the University of Excellence concept of Karlsruhe Institute of Technology. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants program, RGPIN-2020-05502, and an Early Researcher Award from the Government of Ontario.

Data Availability Statement. The full code of the example can be found in the artifact accompanying this paper (DOI: 10.5281/zenodo.18607514).

Disclosure of Interests. The authors have no competing interests to declare.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book, Lecture Notes in Computer Science, vol. 10001. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Armborst, L., Beyer, D., Huisman, M., Lingsch-Rosenfeld, M.: AutoSV-Annotator: Integrating deductive and automatic software verification. In: 30th International Conference on Formal Methods for Industrial Critical Systems. Lecture Notes in Computer Science, vol. 16040, pp. 59–77. Springer, Berlin, Heidelberg (2025). https://doi.org/10.1007/978-3-032-00942-5_4
3. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Model Checking Software. pp. 184–203. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-66149-5_11
4. Banerjee, A., Naumann, D.A.: Local reasoning for global invariants, Part II: dynamic boundaries. J. ACM **60**(3), 19:1–19:73 (2013). https://doi.org/10.1145/2485981
5. Bao, Y., Leavens, G.T., Ernst, G.: Unifying separation logic and region logic to allow interoperability. Formal Aspects of Computing **30**(3-4), 381–441 (2018). https://doi.org/10.1007/s00165-018-0455-5
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.7. Tech. rep., Department of Computer Science, The University of Iowa (2025)
7. Beckert, B., Bubel, R., Drodt, D., Hähnle, R., Lanzinger, F., Pfeifer, W., Ulbrich, M., Weigl, A.: The Java verification tool KeY: A tutorial. In: 26th International Symposium on Formal Methods, FM 2024. pp. 597–623. Springer, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-71177-0_32
8. Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.: Formally verifying an efficient sorter. In: Tools and Algorithms for the Construction and Analysis of Systems: 30th International Conference, TACAS 2024. pp. 268–287. Springer, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-3-031-57246-3_15
9. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses. In: Tests and Proofs. pp. 3–23. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-92994-1_1
10. Beyer, D., Ernst, G., Jonáš, M., Lingsch-Rosenfeld, M.: SV-LIB 1.0: A standard exchange format for software-verification tasks (2025)
11. Beyer, D., Kanav, S., Richter, C.: Construction of verifier combinations based on off-the-shelf verifiers. In: Fundamental Approaches to Software Engineering. pp. 49–70. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-99429-7_3
12. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Tools and Algorithms for the Construction and Analysis of Systems: 31st International Conference, TACAS 2025. pp. 151–186. Springer, Berlin, Heidelberg (2025). https://doi.org/10.1007/978-3-031-90660-2_9
13. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles. pp. 143–167. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_8
14. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: FM 2014: Formal Methods. pp. 127–131. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9

15. Clarke, D., Noble, J., Wrigstad, T. (eds.): Aliasing in Object-Oriented Programming. Types, Analysis and Verification, Lecture Notes in Computer Science, vol. 7850. Springer Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-36946-9
16. Cok, D.R., Leavens, G.T., Ulbrich, M.: JML Reference Manual. Second edition edn. (2025)
17. Dietl, W., Dietzel, S., Ernst, M.D., Muşlu, K., Schiller, T.W.: Building and using pluggable type-checkers. In: International Conference on Software Engineering (ICSE). pp. 681–690. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1985793.1985889
18. Dietl, W., Drossopoulou, S., Müller, P.: Generic universe types. In: ECOOP 2007 – Object-Oriented Programming. pp. 28–53. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73589-2_3
19. Dietl, W., Ernst, M.D., Müller, P.: Tunable static inference for generic universe types. In: ECOOP 2011 – Object-Oriented Programming. pp. 333–357. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22655-7_16
20. Dietl, W., Müller, P.: Object ownership in program verification. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification, pp. 289–318. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36946-9_11
21. Ernst, G., Pfeifer, W., Ulbrich, M.: Contract-LIB: A proposal for a common interchange format for software system specification. In: Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification - 12th International Symposium, ISoLA 2024. Lecture Notes in Computer Science, vol. 15221, pp. 79–105. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-75380-0_6
22. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Programming Languages and Systems. pp. 125–128. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
23. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. Journal of Automated Reasoning **62**(1), 93–126 (2019). https://doi.org/10.1007/s10817-017-9426-4
24. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science: State of the Art and Perspectives, pp. 345–373. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_18
25. Haltermann, J., Jakobs, M.C., Richter, C., Wehrheim, H.: Parallel program analysis on path ranges. Science of Computer Programming **238**, 103154 (2024). https://doi.org/10.1016/j.scico.2024.103154
26. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. The MIT Press (2000). https://doi.org/10.7551/mitpress/2516.001.0001
27. Hoare, C., Misra, J., Leavens, G.T., Shankar, N.: The verified software initiative: A manifesto. ACM Comput. Surv. **41**(4), 22:1–22:8 (2009). https://doi.org/10.1145/1592434.1592439
28. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM **50**(1), 63–69 (2003). https://doi.org/10.1145/602382.602403
29. Huisman, M., Monti, R., Ulbrich, M., Weigl, A.: The VerifyThis collaborative long term challenge. In: Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY, Lecture Notes in Computer Science, vol. 12345, chap. 10, pp. 246–260. Springer, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-64354-6_10

30. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA Formal Methods, vol. 6617, pp. 41–55. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
31. Jakobs, M.C.: PARTPW: From partial analysis results to a proof witness. In: Software Engineering and Formal Methods. pp. 120–135. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_8
32. Kassios, I.T.: The dynamic frames theory. Formal Aspects of Computing **23**(3), 267–288 (2011). https://doi.org/10.1007/s00165-010-0152-5
33. Kassios, I.T.: Dynamic frames and automated verification
34. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: FM 2006: Formal Methods. pp. 268–283. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11813040_19
35. Klamroth, J., Lanzinger, F., Pfeifer, W., Ulbrich, M.: The Karlsruhe Java verification suite. In: The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday, pp. 290–312. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-08166-8_14
36. Lanzinger, F., Weigl, A., Ulbrich, M., Dietl, W.: Scalability and precision by combining expressive type systems and deductive verification **5**(OOPSLA) (2021). https://doi.org/10.1145/3485520
37. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML reference manual (2013)
38. Leino, K.R.M., Moskal, M.: VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In: Proceedings of Tools and Experiments Workshop at VSTTE (2010)
39. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 312–327. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_26
40. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: A manifesto. Commun. ACM **58**(2), 44–46 (2015). https://doi.org/10.1145/2644805
41. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Verification, Model Checking, and Abstract Interpretation (VMCAI). Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer, Berlin, Heidelberg (2016)
42. O'Hearn, P.: Separation logic. Commun. ACM **62**(2), 86–95 (2019). https://doi.org/10.1145/3211968
43. Oortwijn, W., Huisman, M., Sebastiaan J. C. Joosten, Jaco van de Pol: Automated verification of parallel nested DFS. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 247–265. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_14
44. Pfeifer, W., Dietl, W., Ulbrich, M.: A framework for the interoperable specification and verification of encapsulated data structures. Tech. rep., Karlsruhe Institute of Technology (KIT) (2026). https://doi.org/10.5445/IR/1000191071
45. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817

46. Richter, C., Chalupa, M., Jakobs, M.C., Wehrheim, H.: Cooperative software verification via dynamic program splitting. In: 47th International Conference on Software Engineering (ICSE). pp. 2087–2099 (2025). https://doi.org/10.1109/ICSE55347.2025.00092
47. Smans, J., Jacobs, B., Piessens, F.: VeriFast for Java: A tutorial. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification, pp. 407–442. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36946-9_14
48. Weiß, B.: Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction (2011). https://doi.org/10.5445/KSP/1000021694