Contract-LIB: A Proposal for a Common Interchange Format for Software System Specification *

Gidon Ernst[1], Wolfram Pfeifer[2], and Mattias Ulbrich[2]

[1] Ludwig Maximilian University, Munich, gidon.ernst@lmu.de
[2] Karlsruhe Institute of Technology, {wolfram.pfeifer,ulbrich}@kit.edu

Abstract. Interoperability between deductive program verification tools is a well-recognized long-standing challenge. In this paper we propose a solution for a well-delineated aspect of this challenge, namely the exchange of abstract contracts for possibly stateful interfaces that represent modularity boundaries. Interoperability across tools, specification paradigms, and programming languages is achieved by focusing on abstract implementation-independent behavioral models. The approach, called Contract-LIB in reminiscence of the widely-successful SMT-LIB format, aims to standardize the language over which such contracts are formulated and provides clear guidance on its integration with established methods to connect high-level specifications with code-level data structures. We demonstrate the ideas with examples, define syntax and semantics, and discuss the rationale behind key design decisions.

1 Introduction

Deductive verification [27] following the design-by-contract principle [36] allows proving software correct against expressive specifications that can capture strong properties. Today, users can choose from a variety of mature and powerful deductive verification tools such as Dafny [35], KeY [3], KIV [22], SecC [21], VerCors [6], Verifast [32], Viper [37], Why3 [37] that target many programming languages and are based on different verification paradigms.

Specifications are often built using mathematical modeling tools, in particular algebraic data types (ADTs), to formally represent non-trivial data structures and contracts for their operations. ADTs offer a high level of abstraction, are precisely defined, and are expressive enough to capture complex operations if a sufficiently extensive toolbox of predefined operations is available. Due to their

* This version of the contribution has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is not yet available. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.
This work reflects a contribution to the SpecifyThis track at ISoLA 2024, https://2024-isola.isola-conference.org/.

axiomatic foundations in mathematics, ADTs possess a well-defined semantics upon which the verification tools agree; regardless of the paradigms they follow.

A common specification pattern – which can be regarded as the prototypical case – is that the abstract state of a (well-encapsulated) data structure is formally and abstractly modeled via ADT values; and operation contracts are expressed formally by defining how the abstract state evolves.

In an ideal world, verification tools would be *interoperable* in the sense that they can freely exchange specifications and other verification artifacts. While approaches and tools agree on the semantics of ADTs, they differ considerably on how abstract state and implementation state are coupled, and on how the framing problem [17] is addressed.

Hence, to enable interoperation for more than trivial examples, an approach to exchange information is needed for sharing abstract specification artifacts. In this paper we introduce Contract-LIB, a new exchange infrastructure for sharing design-by-contract specifications between different verification tools and paradigms. Contract-LIB is a language to concisely and tool-independently represent abstract specifications. It is designed to allow verification tools to import abstract specifications as concrete tool-specific specifications (resp. export in the opposite direction).

Each verification paradigm and tool has specific strengths and limitations (stemming not only from the underlying theoretical foundations, but also from the emphasis placed by the developers of that tool). Being able to rely on a common abstract background specification in Contract-LIB and to thus exchange specifications allows the seamless integration of otherwise incompatible approaches.

There are numerous situations in formal software design that can benefit from an effective specification exchange technology: verified standard libraries (like JDK or libc) can be shared amongst tools, collaborating code base written in different languages can be verified in heterogeneous projects, and different parts serving different purposes in a program can be verified using different verification techniques. Sect. 2 elaborates on typical application scenarios for Contract-LIB.

One design goal of Contract-LIB is to have a standardized machine-readable (yet still human-readable) interchange format with clear and simple structures. Comparable exchange formats serve a similar purpose very successfully in other formal method communities: DIMACS, SMT-LIB [12], BTOR2 [38], and the TPTP format [40] have had a huge impact and unlocked unforeseen applications.

First steps have already been made towards interoperability. For example, [30] translate specifications and auxiliary annotations between JML, VerCors, and Krakatoa [24] for Java. In contrast, [34] aim to integrate static analysis tools and software verifiers into more expressive deductive ones to increase proof automation. In full generality, however, tool integration remains challenging due to the heterogeneous and diverse feature set of the tools, i.e., exactly that aspect which makes innovation possible and integration worthwhile in the first place. Hence, it is questionable whether a fully standardized approach is even desirable.

Therefore, in this paper, we look at a particular, well-delineated aspect of tool integration, namely to find common ground for a specification formalism that is well-suited for interoperability of deductive verification tools in the sense that it is *useful* and at the same time *easy to adopt*. This issue has been discussed in-depth in the community over the past years and a conceptual opportunity has been identified that we now follow-up with a concrete technical proposal.

The key insight behind this opportunity is that the above mentioned ideal case specification pattern used in deductive tools *typically* encompass three different levels [44]: 1) code-level entities and assertions that capture implementation-level concepts, 2) data abstractions using ADTs to represent code-level values with behavior abstractions expressed using ADT data expressions, and 3) logical coupling machinery to formally connect these two. While 1) and 3) must inherently be specific to the used verification approach and programming language, it is the mathematical abstractions in 2) that are stable across tools and are hence the target to be exchanged in our approach. On the other hand, while Contract-LIB specifications only talk about abstractions, import and export functionalities from and into actual programming languages must ensure that necessary tool-dependent specification artifacts for 1) and 3) are provided, e.g. to guarantee that the modules are well encapsulated entities. Only thus can a sound modular interplay of the verification tools be guaranteed.

Contribution: This paper standardizes an approach to the integration of deductive verification tools, called Contract-LIB in reminiscence of SMT-LIB. We motivate and define Contract-LIB as a standardized interface modeling language as an extension of SMT-LIB that expresses those parts of behavioral contracts that are shareable. We describe the rationale behind its design that allows for easy adoption and integration into existing infrastructure. In Sec. 2 we identify a number of use-cases and in Sec. 3 we walk through some design requirements using a nontrivial example. We work out the conceptual approach for integration in the style of Data Refinement [28] as general guidelines for point 3) above. We discuss syntax, semantics, and showcase how different specification paradigms (Dynamic Frames, Separation Logic) can be connected to Contract-LIB (Sec. 4).

Data Availability: The tool-chain and examples are available as open source on Github: `https://github.com/gernst/contract-lib`.

2 Use Cases for Contract-LIB

The fundamental and guiding principle behind the design is to precisely describe interfaces of possibly stateful components/modules that are *well-encapsulated*. Intuitively, a module is well-encapsulated if the possible interactions, as observed at the interface can be explained fully and precisely in terms of an abstract mathematical model *without* referencing (pun intended) implementation-level concepts, notably mutable memory. The key observation behind Contract-LIB is that—despite all the heterogeneity in methodology and language-specific

aspects—the principles and constituents of such mathematical abstractions are well-understood, stable, semantically unambiguous and widely supported in the better part of their functionality. Being based on that observation, Contract-LIB has a well-defined scope which aspects of specifications it can describe and which we intentionally do not aim to incorporate. In the terminology of [44], Contract-LIB is a format that "formally and fully captures the requirements and nothing more" (Sec 1. point ②).

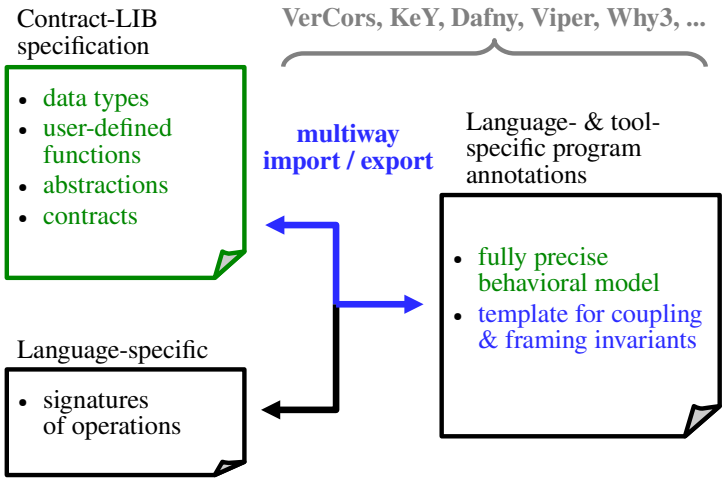

Fig. 1. Interoperability of Contract-LIB specifications.

In this section, we discuss the use cases for interoperability and tool integration in which we see a potential for Contract-LIB to play a central role resp. that benefit and/or are enabled by it in the first place. Consider Fig. 1, which shows how information is communicated: At the top-left in green, Contract-LIB scripts contain all that is needed to describe behaviors of well-encapsulated modules and the underlying domain formalization (Sec. 3). This information can be combined and/or related to structural information that is specific to a programming language, such as `interface` specifications in Java or header files in C, as shown bottom-left in black.

A deductive verification tool can then import these, for example, by linking a given Contract-LIB specification with a suitable signature, to internalize resp. annotate the program with contracts in its own specification language. Part of this translation step could already be the scaffolding of the typical methodology-specific definitions, such as those of abstraction predicates or footprints, to be later filled in by a human.

The converse direction is to export from a tool's native specification language into Contract-LIB. In fact, its role is more that of an intermediate representation

rather than a first-class language directly written by humans, similar in spirit to SMT-LIB, which is also primarily designed for ease of machine readability in mind. Below, we give some concrete scenarios and small examples, deferring a larger example to Sec. 3.

Co-development of larger projects composed of modules: A first scenario for interoperability that comes to mind is to be able to use multiple deductive verification tools in the context of a larger project that consists of multiple components, all of which are written in the same programming language. Considering that a formal specification often touches different formal aspects, one would like to be able to use different tools with complementary strengths. For example, one could use VerCors to prove linearizability of a concurrent data structure and then communicate the established abstract behavior to the OpenJML [19] verifier that scales better for lightweight properties in order to establish the client's sequential correctness.

Applications comprised of multiple implementation languages: Cross-language verification projects are tricky to realize today [**?**] unless tools already support multiple programming languages. One scenario is when native code (compiled from C) is invoked from a higher-level programming language (such as Java). Reasons for relying on native code could be for example efficiency concerns, connecting to operating system services, or relying on a given library that happens to be written in the lower-level language. Besides native code, a prime use-case would be that Dafny is nowadays used at scale to implement critical infrastructure. Dafny has multiple back-ends into which the programs can be translated into, but of course, the interfaces between the Dafny parts and the parts written in the more conventional language are annotated with specifications on the Dafny side. Mistakes here can undermine the verification guarantees, which is why Dafny can insert runtime checks to validate assumptions about the integration points. For critical cases, perhaps one may want to instead verify these as guarantees, and Contract-LIB offers a way to mechanically translate the respective contracts between Dafny and the verifier used for the conventional language.

Shared specifications of standard libraries. Shareable abstractions of standard libraries for a given implementation language (such as Java's JDK or `libc`) are a promising prospective to converge as a whole field towards making tools more "industry-ready". While comprehensive support for such standard libraries is usually not needed to solve small but intricate verification challenges, scaling to larger and more realistic code bases at some point will rely on having specifications for example for strings, Java's collection classes like `java.util.List`, or C's low-level memory allocation and manipulation functions like `malloc` and `strlen`. So far, support for such libraries is rather underdeveloped, with some exceptions, e.g., specifications written in JML[3] or in VeriFast[4]. Contract-LIB will provide

[3] https://github.com/OpenJML/Specs
[4] https://github.com/verifast/verifast/tree/master/bin

the means to seamlessly exchange such library specifications across tools for the same implementation language, even across specification paradigms.

To give a concrete example, VeriFast specifies `memset` from `string.h` as follows (slightly adapted), whose purpose is to initialize each byte in a memory region of the given `size` starting at pointer `array` with a particular `value`.

```
void* memset(void *array, char value, size_t size);
    //@ requires chars(array, size, ?cs0);
    //@ ensures  chars(array, size, ?cs1) &*& result == array;
    //@ ensures  all_eq(cs1, value) == true;
```

The contract expresses this by a combination of pre-/postconditions that capture the methodology-specific validity of the memory region in terms of a separation logic predicate, which links in particular the array contents to abstract value `cs1` as a mathematical sequence of bytes. The tool-independent constraints then ensures that this sequence has the correct value in all places. Sharing such specifications can also tie deductive techniques closer to fully automated software model checkers that participate for example in SV-COMP [14].

Formal specifications of normative documents. RFCs are the typical format in which standards for our connected world are established. This includes widespread and standard protocols such as HTTP and SSL. Of course, there are many formalizations of protocols in the literature, for example based in the modeling languages of specialized tools such as Tamarin [13]. However, these are rarely connected to implementations, notable exceptions are for example Project Everest [15]. Here, exporting the corresponding formal models and sharing them as Contract-LIB specifications can ensure that we not only have RFCs as an informal basis for implementations, but also that we gradually move towards a standardized and fully formal set of models of protocols that can be realized by verified implementations.

Verification competitions and challenges. Verification competitions like the VerifyThis on-site events [20] and the longer running cooperative challenges [29,23] may benefit from a common interchange format. For example, verification tasks can be specified by the competition organizers so that participants get a clear goal for their verified implementations. Conversely, the respective abstract specifications could be a basis for a machine-assisted analysis and comparison of the results, which today are judged by humans [20]. With respect to the longer-running competitions, Contract-LIB enables tighter collaboration in the sense of the first few scenarios mentioned above. This takes on suggestions that are part of the respective challenge descriptions already [23].

Integration with other program analysis techniques. There are many other use-cases for Contract-LIB besides verification. Either can we generate specifications from traditional model-based development approaches, such as FOCUS [5] or maybe UML state-charts, or alternatively can we export behavioral models into other tools, such as automatic test generators or symbolic model checkers [26].

Possibly, there is some common kernel that can be translated losslessly between Contract-LIB and MoXI [39]. Finally, invariant inference techniques and lemma generators can be integrated to produce facts that at least give some proof support for properties that can be expressed at the abstract level already, which by experience is often relevant in practice.

3 Requirements for Contract-LIB

In this section we show how the specifications of desired behaviors are typically realized in deductive verification tools. We follow an example, which is taken from the VerifyThis long-term challenge [5], which has as its subject the in-memory cache server `memcached` [23]. We discuss the described functionality alongside the means of its formalization. The case-study is intentionally somewhat complex to talk about a wide range of features and we complement it with several additional smaller examples to discuss some features not present in the `memcached` specification.[6]

`Memcached` is a high-performance key-value cache which is used to speed up cloud-native applications, e.g., by avoiding database lookups. It is realized as a small daemon that keeps an in-memory mapping from keys (sequences of bytes) to entries. What distinguishes `memcached` from say a hash-map is that entries have a lifetime after which the association implicitly expires, and moreover, entries can be evicted due to memory pressure. `Memcached` is accessed via a simple text-protocol as exemplified in Fig. 3. Here, an entry is stored under key `token` with flags `0`, a timeout of 30s, and a length of 6 bytes, the subsequently line denotes the data associated to this entry, here the arbitrary number `162596`. Lines 6ff and 12ff show the responses when accessing the entry before resp. after expiry of the timeout. Note, we will not model the protocol representation of data types itself but instead formulate a higher-level abstraction.[7]

The discussion hinges on Fig. 2, in which the green part above the double horizontal line represents the mathematical abstraction, i.e., all that information which we wish to transfer across tools and implementation languages using Contract-LIB. Below, the figure visualizes the respective implementation-level data structures, which are grouped into logical units that independently can be coupled to the respective mathematical abstractions. This is essentially the idea behind Data Refinement [28], which is reflected in modern tools for example by the use of `ghost` state, abstraction predicates, data structure invariants, and so on. The verification itself ensures that such abstractions are compatible with the dynamic behavior of the implementation, typically in the form of a simulation proof.

[5] https://verifythis.github.io/

[6] The specification is also available at https://gist.github.com/gernst/eb0028af1961b6df4740b5ceca628cf9

[7] The procotol is documented at https://github.com/memcached/memcached/blob/master/doc/protocol.txt

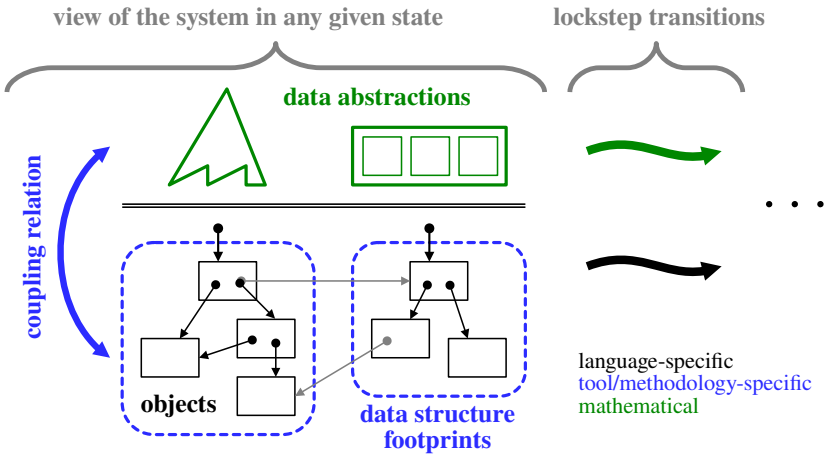

Fig. 2. Deductive verification of systems with respect to behavioral abstractions. Black: specific to the implementation language, green: sharable abstractions, blue: specific to the tool/methodology.

Ingredient 1: Expressive Standardized Mathematical Theories. Data abstractions require expressive mathematical background theories, including user-declared algebraic data types and mathematical sequences, sets, and maps. Moreover, besides the typical built-in operations such as cardinality, membership test, union/intersection/difference, subset, map update, user-defined functions are often needed to capture certain concepts concisely.

Fig. 4 shows the data type declarations of over which the system model for `memcached` is defined, in the specification language of Dafny [35]. It starts by declaring some unspecified types to represent parts of the protocol messages, i.e., we choose to abstract from the underlying byte representation.

`Memcached` associates to each key of type `Key` a value of datatype `Entry`, which tracks the respective `data` and metadata comprised of `flags`, `expiry` timeout, and a version counter `unique`, which is is picked to be globally fresh by the system for every transaction and which supports CAS-style synchronization among concurrent clients (not discussed further here). Datatype `Result` is used to encode return values of operations, further described below, where the `Values` is used to encode both responses shown in Fig. 3, once with a singleton sequence and once with the empty sequence; remarking that there is an operation `gets`, too, which possibly returns multiple values. Predicate `live` determines whether an entry must be considered expired when compared to a given time point `now`, where `none` represents that the entry does not expire.

We therefore derive the following requirements: While tools in practice are roughly compatible in their support for mathematical sstructures, the key goal associated with this ingredient is to *standardize* which data structures are deemed available, what operations should be supported out-of-the-box, and what the

```
// entry with timeout 30s
set token 0 30 6
162596

// before timeout
get token
VALUE token 0 6
162597
END

// after timeout
get token
END
```

Fig. 3. Example interaction with `memcached` at the protocol level

```
type Key(==) // Keys support equality
type Flags
type Time = nat
type Data

datatype Entry
  = Entry(data: Data, flags: Flags, unique: nat,
          expiry: option<Time>)

datatype Result
  = Values(entries: seq<(Key,Entry)>)
  | Stored | Touched | Deleted | Exists | NotFound

predicate live(expiry: option<Time>, now: Time) {
  match expiry {
    case none => true
    case some(time) => now < time
  }
}
```

Fig. 4. Dafny model of the data structures of `memcached`.

mechanism to communicate user-defined operations should be. We address this in Sec. 4 by basing Contract-LIB on the widely established SMT-LIB standard, complemented with the definition of those theories that are missing from the official standard (using notation already supported by cvc5).

Ingredient 2: Declarations of Data Abstractions. In order to view some module of a given system in terms of a data abstraction, we need some mechanism to bundle together those state variables that represent this abstraction in such a way that it integrates well with typical coupling mechanisms found in practice.

To continue the example, Fig. 5 the interface specification of `memcached`, realized as **`trait`** `Code`, in terms of two attributes variables, `entries` and `uniques`, which in being marked as **`ghost`** are not part of the implementation but instead whose purpose is to keep track of the current state of the abstraction. While `entries` simply maintains the associations between keys and values, set `uniques` accumulates all version counters used so far, including those of entries not present in `entries` any longer.

In addition, we provision two typical features that connect these to potential implementations of this interface, i.e., a set `footprint` of memory locations (object references) on which the abstraction depends on, and an abstract predicate `Valid`, to be realized by each implementation, which captures all necessary invariants on both the abstract and concrete state space. Here, we may already manifest some of these, such as the fact that at least all current `unique` version counters must be tracked by set `uniques` and that these must be distinct.

We emphasize that `footprint` and `Valid` are *not* to be understood as part of the actual behavioral model—Contract-LIB will in fact not represent these but rather it is designed in such a way that it honors a wide range of similar mechanisms. In VeriFast [?] for the C programming language for example, the coupling would be expressed as a Separation Logic abstraction predicate, where

```
trait Cache {
  // Abstract model of the internal state
  ghost var entries: map<Key, Entry>
  ghost var uniques: set<nat>

  // The coupling to the implementation is specified in terms of
  // a dynamic frame of relevant memory locations ...
  ghost var footprint: set<object>

  // ... and a predicate describing invariants and coupling to code.
  // Note that this predicate is not defined here but as part of instances,
  // yet, we already state some desired consequences of its definition.
  ghost predicate Valid()
    reads this, footprint
    ensures forall key :: key in entries
        ==> entries[key].unique in uniques
    ensures forall key1, key2 :: key1 in entries && key2 in entries && key1 != key2
        ==> entries[key1].unique != entries[key2].unique

  // ...
}
```

Fig. 5. Dafny specification of the state of the data abstraction, expressed over the data types from Fig. 4 as ghost attributes of an interface specification realized as a **`trait`**. Moreover, the trait already provisions the connection to possible implementations.

`struct` `cache` is the implementation-level data structure being related to the abstract values given by `entries` and `uniques` in a similar manner (the `footprint` is implicit in this methodology).

```
predicate Valid(struct cache *cache, map entries, set uniques)
```

In order to achieve this compatibility, we need to identify the requirements and conditions formally that control aliasing just sufficiently that changes at the abstract level are not interferred with by aliases that are representable in the implementation only—this leads to strong independence assumptions in Sec. 4, which essentially formalize some important aspects of a module being *well-encapsulated*, and therefore justifying the scope of Contract-LIB as described earlier with a technical point. We emphasise that within this argumentation underpins the key rationale, why Contract-LIB is an actionable approach to integration in the first place.

Ingredient 3: Contract Definitions for Operations. Having established the other two ingredients, we can now formulate contracts that describe behavior in terms of them. Contracts are pre-/postcondition pairs that relate the input parameters and return values to a transition between a pair of states.

Of the many operations supported by `memcached`, Fig. 6 defines the contracts of the methods implementing protocol commands **`get`** and **`set`**, respectively, as shown in Fig. 3. The contracts of these two operations have two parts: The first part manifests that predicate `Valid` serves as a class invariant, i.e., may be assumed upon method calls and must be re-established by methods that change the state (such as `Set`). In addition, we impose some typical frame conditions

```
trait Cache {
  // ...

  method Get(key: Key, ghost now: Time)
    returns (result: Result)
    requires Valid() // assume invariant Valid()

    // effect of the operation on the abstract state
    ensures key  in entries &&  live(entries[key].expiry, now)
                ==> result == Values([(key, entries[key])])
    ensures key !in entries || !live(entries[key].expiry, now)
                ==> result == Values([])


  method Set(key: Key, data: Data, flags: Flags, expiry: option<Time>, ghost now: Time)
    returns (result: Result, ghost unique: nat)

    modifies this, footprint
    requires Valid() // preserve invariant Valid()
    ensures  Valid()

    // typical frame assumption wrt. existing objects
    ensures  old(footprint) <= footprint && fresh(footprint - old(footprint))

    // effect of the operation on the abstract state
    ensures  unique !in old(uniques) && uniques == old(uniques) + {unique}
    ensures  entries == old(entries)[key := Entry(data, flags, unique, expiry)]
    ensures  result == Stored
}
```

Fig. 6. Contract definitions of two operations of `memcached`.

on `footprint`, i.e., that it can only grow in terms of freshly allocated objects to guarantee independence between this and other objects.

While the first part very generic and appears in this form in virtually all case-studies that embrace Dafny's methodology, the second part is case-study specific and explains the behavior of `memcached` precisely in terms of the abstract state representation. The contract specification of `Get` is split up into two behaviors. In case `key` denotes a valid entry that has not expired yet the result is the singleton sequence that pairs the key with the respective entry, wrapped in the `Values` constructor of type `Result`. Otherwise, the operation similarly returns the empty sequence of keys (constructor `NotFound` of `Result` serves a different purpose). Operation `Set` has two return values, which is supported first-class in Dafny: `result`, which is always `Stored`, and `unique`, which is the unique version number assigned to the entry after its creation or modification. While `unique` may not be accessed (directly) by the caller of this interface, we use it to specify the contract: Set `uniques` was disjoint to and is enlarged by that value and `entries` at index `key` is updated accordingly. Note the use of keyword `old` to express the relation between the state of the abstraction before and after the operation.

Another interesting situation is when the abstractions of two objects change simultaneously by the same method call. For example, we could have a class for storing lists of integers, with a method that moves some values to another collection, such as:

```
class IntList {
    ghost var content: seq<int>

    method moveNegativeValuesTo(IntList that)
        modifies this, that
        ensures this.content == pos(old(this.content))
        ensures that.content == old(that.content) + neg(old(this.content))
}
```

This specifies that both objects are **`modified`** and how the respective lists' content changes (where `pos` and `neg` are some mathematical functions that retain resp. drop the positive elements only).

The requirement for Contract-LIB therefore to be able to associate with each method/top-level function two key information. First, which abstractions are passed in and out including the receiver `this` (or `self`) in object-oriented languages. In Fig. 6, for example, change is reserved to the trait itself, as indicated by the **`modifies`** clause, but in the second example, the abstractions of two objects is changed at the same time. In Sec. 4 we represent this by annotating each parameter with an input/output mode, which describes whether the respective datum is an input or output, or whether it is a dynamic object whose abstraction is changed accordingly.

4 Technical Realization of Contract-LIB

Contract-LIB intends to specify software components that have a well-defined external interface and an encapsulated internal state. It is designed to be agnostic to the methodology and tool used to verify the correspondence between the specification expressed in Contract-LIB and potential implementations or refinements towards any implementation language. The format realizes the the three ingredients discussed in Sec. 3, namely expressive standardized mathematical domain models, compatibility with established approaches for data abstraction, and support for denoting the dynamic behavior arising from method calls on the objects that are affected, but abstractly. Note that the structure of this section cross-cuts these three ingredients with respect to their syntactic representation, the semantic foundations, and practical concerns.

4.1 Syntax for Defining Abstractions and Contracts

The syntax of Contract-LIB is based on SMT-LIB. It is based on S-expressions and therefore easy to parse. More importantly, all the machinery for ingredient 1, i.e., the definition of rich mathematical background theories, is already in place. This means that all of the following SMT-LIB commands [11] are valid in Contract-LIB too: **`declare-sort`**, **`define-sort`**, **`declare-fun`**, **`define-fun`**, **`define-funs-rec`**, **`declare-datatypes`**, and **`assert`**.

As an example, function `live` can be encoded in SMT-LIB either in terms of a function definition and pattern matching

$$
\begin{aligned}
\langle command \rangle &::= \texttt{(declare-abstractions (}\langle sort\_dec \rangle^{n+1}\texttt{) (}\langle datatype\_dec \rangle^{n+1}\texttt{))} \\
&\quad \mid \texttt{(define-contract } \langle symbol \rangle \texttt{ (}\langle formal \rangle^{+}\texttt{) (}\langle contract \rangle^{+}\texttt{))} \\
&\quad \mid \ldots \\
\langle formal \rangle &::= \texttt{(}\langle symbol \rangle \texttt{ (}\langle mode \rangle \ \langle sort \rangle\texttt{))} \\
\langle mode \rangle &::= \texttt{in} \mid \texttt{out} \mid \texttt{inout} \\
\langle contract \rangle &::= \texttt{(}\langle term \rangle \ \langle term \rangle\texttt{)} \\
\langle term \rangle &::= \texttt{(old } \langle term \rangle\texttt{)} \mid \ldots
\end{aligned}
$$

Fig. 7. Grammar of Contract-LIB, where nonterminals $\langle symbol \rangle$ and $\langle sort \rangle$ are defined as in SMT-LIB, but for $\langle term \rangle$ we add an additional form `(old ...)` which is allowed to appear in contracts to express transitions as a relation over a pair of states.

```
(define-sort Time () int)
(define-fun live ((expiry (Option Time)) (now Time))
     (match expiry
        ((none        true)
         ((some time) (< now time)))))
```

or semantically equivalently using a function declaration and two axioms.

The grammar for extensions specific to Contract-LIB is shown in Fig. 7 For ingredient 2, we introduce a new command, **`declare-abstractions`**, which has exactly the same structure as **`declare-datatypes`** but which assigns a different purpose to the sorts declared. As an example, the two ghost variables in Fig. 5 are grouped into an abstraction `Cache` as follows:

```
(declare-abstractions
    ((Cache 0))
    ((Cache
        ((entries (Map Key Entry))
         (uniques (Set Int))))))
```

The intended interpretation is that sort `Cache` encodes the abstraction of a module with the same name, such as a class or a C API. Note that having multiple constructors may reflect that the state of the module can in one of multiple operational modes, each with their respective abstract data associated. However, note also that while this is naturally supported in approaches using Separation Logic predicates, it is less clear how the connection to a fixed set of top-level **`ghost`** variables without introducing a nested data type.

We introduce a second new command **`define-contract`**, which captures possible state transitions on abstractions introduced by **`declare-abstractions`**. Such a contract specification refers to the name of the operation, given by the $\langle symbol \rangle$, followed by a list of formal parameters and a list of pre-/postcondition. A formal parameter is analogous to that of say a **`define-fun`** command, however, instead of pairing names just with types as in `(x Int)` we additionally annotate an

input/output mode as, for example `(key (in Key))` declares the `key` parameter to be an input argument, whereas `(this (inout Cache))` declares the (implicit) receiver argument as being a parameter that is both read and updated by the corresponding call (for `Set`). This means that while the *reference* `this` in Dafny remains pointing to the same object, the *abstraction* of the underlying object does change.

```
(define-contract Cache.get
    ((this (in   Cache) (key (in Key))))
    ((; first behavior: the key is exists and the entry is live
         (and (set.contains (map.keys (entries this)))
              (live (map.get (entries this)) now))
         ...)
     (; second behavior here
         ...)))

(define-contract Cache.set
    ((this (inout Cache) (key (in Key)))
     (data Data) (flags Flags) (expiry (Option Time)) (now Time))
    ((...)))
```

4.2 Standardized Theories and Polymorphism

As we have seen in Sec. 3, finite sequences, sets, and maps are indispensable as means to specification. However, the SMT-LIB standard includes functional arrays as the only compound data type. Although it is possible to represent sequences, sets, and maps directly in terms of arrays, this is not convenient at all as SMT-LIB arrays are unbounded, so that encoding cardinality and well-behaved equality is tricky. To that end, deductive tools with SMT backends tend to axiomatize in a way that it is tightly integrated with other aspects of the encoding.

Contract-LIB therefore takes the approach to fix the signatures of sorts and functions that are intended to be exchanged. We expect this list to converge over time during the adoption process, the current proposal is available online.[8] Here, the list of functions supported for the respective data types as well as their naming mirrors that of cvc5 [9][9] for `Set` and `Seq`, and those we propose for `Map` are similar. We emphasize that tools would still aim to translate these types and functions into their *front-end* specification language, so that they would still use whatever encoding is already in place. This includes concerns like partiality of operations (Sec. 4.5).

Having polymorphic container types like those discussed above as well as practical experience suggests that from a standpoint of expressiveness, first-class support for polymorphic definitions is highly desirable. Note, SMT-LIB only

[8] https://github.com/gernst/contract-lib/tree/main/src/test/contractlib/builtin
[9] https://cvc5.github.io/docs → Theory References

$\langle command \rangle$ ::= **(declare-fun (par (**$\langle symbol \rangle^+$**) ((**$\langle sort \rangle^*$**)** $\langle sort \rangle$**)))**
| **(define-fun (par (**$\langle symbol \rangle^+$**) ((**$\langle sorted\_var \rangle^*$**)** $\langle sort \rangle$**)**$\langle term \rangle$**))**
| **(assert (par (**$\langle symbol \rangle^+$**)** $\langle term \rangle$**))**
| ...

Fig. 8. Extensions for polymorphism in Contract-LIB.

supports schematic sorts, but has no built-in support for polymorphic definitions. However, some SMT-LIB like formats, such as TIP [18] have introduced notation that is fits well for our purposes, too. As shown in Fig. 8, we embrace the syntactic form `(par ...)`, already in use for **`declare-datatypes`** in SMT-LIB, to optionally appear in certain commands to bind type parameters, i.e., critical parts of commands are wrapped in a declaration of type parameters, simply given as a list **(**$\langle symbol \rangle^+$**)** of symbols.

4.3 Semantics of Contracts

We rely on the semantics of SMT-LIB as defined in [11, Sec 5.3]. Scripts are interpreted over Σ-structures $\mathbf{A}$. It assigns carrier sets $\sigma^{\mathbf{A}}$ to sorts σ, specifically, the carrier sorts of data type sorts are freely generated (the term model). Function symbols f are interpreted as total functions $f^{\mathbf{A}}$ over the carrier sorts of its signature. Open terms t with free variables are interpreted over a structure $\mathbf{A}$ and valuation v, which assigns to each sorted variable $x\colon \sigma$ a value $v(x) \in \sigma^{\mathbf{A}}$ of the respective carrier set. A pair $\mathcal{I} = (\mathbf{A}, v)$ is called an interpretation, which gives rise to the semantics of terms in the standard way, written $[\![t]\!]^I$. For a boolean term ϕ, we write $\mathcal{I} \models \phi$ if $[\![t]\!]^{\mathcal{I}} = \mathbf{true}$ where **true** is the semantic truth value (SMT-LIB is based on classical logic).

We introduce a new semantic domain of contracts $c \in \mathcal{C}$ that captures the interpretation of contracts as an indexed collection of binary relations between interpretations where suitable valuations range over variables representing all parameters. Specifically, $[\![c]\!]$ contains entries $(j, \mathcal{I}, \mathcal{I}')$ for calling the j-th pre-/postcondition successfully and entries $(j, \mathcal{I}, \bot)$ to signify that the j-th postcondition is not satisfied in $\mathcal{I}$.

For an interpretation $\mathcal{I} = (\mathbf{A}, s)$, we will call valuation s an abstract state or more suggestively a *ghost state*, to highlight its conceptual role, and the intention of assigning formal meaning to contracts $\mathcal{C}$ is to characterize the abstract transition from states s at call time to states s' upon successful return of the operation, where structure $\mathbf{A}$ in $\mathcal{I}' = (\mathbf{A}, s')$ remains unchanged as expected.

In terms of its abstract syntax, a contract with n parameters and k pre-/postcondition pairs

$$c \in \mathcal{C} = (x_1\colon \mu_1\sigma_1, \ldots x_n\colon \mu_n\sigma_n;\ (\varphi_1, \psi_1), \ldots, (\varphi_n, \psi_k))$$

declares a list of parameters x_i, each annotated with a sort σ_i and its mode $\mu_i \in \{\mathbf{i}, \mathbf{o}, \mathbf{io}\}$. Pairs (φ_j, ψ_j) pairs of preconditions ϕ_j and postconditions ψ_j, which jointly describe the possible behaviors.

A contract is well-specified if variables with **o** do not occur in any φ_j and not inside **old**(_) in any ψ_j, as we prefer to not fix an interpretation that may be counterintuitive to some users.

The semantics of such a contract is intended to be as follows:

$$\begin{aligned}[\![c]\!] = \quad & \left\{ \left(j, (\mathbf{A}, s), \bot\right) \mid s \not\models \varphi_j \right\} \\ & \cup \left\{ \left(j, (\mathbf{A}, s), (\mathbf{A}, s')\right) \mid s \models \varphi_j \text{ and } s, s' \models \psi_j \text{ and } s \equiv_{\mathbf{i}} s' \right\}\end{aligned}$$

where $s \equiv_{\mathbf{i}} s'$ denotes that $s(x) = s'(x)$ for all $x_i \colon \mathbf{i}\,\sigma_i$ and $j = 1, \ldots, k$. Note the outcome $\bot$ when the precondition is violated, which is distinct from absence of tuples when the postcondition is `false`.

Relational semantics $s, s' \models \psi_j$ evaluates each occurrence of a variable inside **old**(_) with respect to s and otherwise with respect to s'. This mirrors the only sensible choice that input parameters must be interpreted with respect to the state at call time. Dafny disallows local assignments to method parameters to avoid the need of imposing this check.

4.4 Integration with Dynamic Frames and Separation Logic

For this discussion, we assume a simple semantic model with a global heap $h \colon H$ where $H = R \rightarrow\!\!\!\!\!\rightarrow O$ is the type of heaps modeled as a partial maps from references R to objects O.

An abstraction $\alpha^{\mathcal{I}}(r, t, h)$ with $\alpha \subseteq R \times \sigma \times H$ of an object residing at $r \in \mathrm{dom}(h)$ is a predicate that describes when the attributes of $o = h(r)$ and its "dependent" objects are together abstracted to the value $[\![t]\!]^I$ denoted by $t \colon \sigma$ in the current logical state s of an interpretation $\mathcal{I} = (\mathbf{A}, s)$.

In the Dynamic Frames approach, a single global heap h is reflected into the logic as explicit variables so that users may express well-formedness conditions or other properties over it. In addition, each object o with $o = h(r)$ is associated with a Dynamic Frame $f(o)$ which defines the set of dependent objects as $\{r' \mid r' \in f(o)\}$. This frame f may be represented either as an additional attribute of o, that is updated explicitly and constrained via an invariant (as in Sec. 3), or it is denoted by a potentially recursive logic function.

In Separation Logic, predicates $\alpha^{\mathcal{I}}(r, t, h)$ are *local* in the sense that the part of the heap described by h coincides with the memory locations required to determine the predicate's value. This intuitively means that (for precise predicates), knowledge that $\alpha^{\mathcal{I}}(r, t, h)$ holds implies that $\mathrm{dom}(h) = f(o)$, albeit function f is never made explicit.

A key requirement is that abstractions $\alpha^{\mathcal{I}}(r, t, h)$ are *well-framed*: They are allowed to depend only on those memory locations of the dependent objects. In Separation Logic, this feature is built-in, in fact it is what gives Separation Logic its ease of reasoning: Any change of the heap that is legal according to the proof rules will either directly concern $\alpha^{\mathcal{I}}(r, t, h)$ or leave it implicitly intact.

In the Dynamic Frames, the situation is a bit more complicated, and we require that abstractions are stable under heap modifications, $\alpha^{\mathcal{I}}(r,t,h) \implies \alpha^{\mathcal{I}'}(r,t,h')$ for all heaps $h \equiv_{f(h(r))} h'$ that coincide on the locations denoted by the objects frame and all interpretations $\mathcal{I} \equiv_{\text{free}(t)} \mathcal{I}'$ that coincide on the valuation of t's free variables.

Well-framing of abstraction predicates relates to the notion of well-encapsulation discussed earlier: Suppose we have a method $m(r_1, \ldots, r_k)$ with k reference parameters, then the relation to Contract-LIB contracts in terms of some abstraction predicates $\alpha_1, \ldots, \alpha_k$ of the respective reference types to mathematical counterparts only makes sense if there is no mutual aliasing between the footprints of `inout` and `out` parameters with any type of parameter, including the `in` parameters. In Dafny, for example, this is in parts realized by making explicit the dependency via `reads` annotation on invariants like `Valid`, but in practice, additional constraints may need to be specified manually.

While enforcing well-encapsulation is in fact specific to the method and tool, it may be reasonable in practice to rely on unspecified and even synthetic predicates α that are solely introduced for the sake of tracking ownership of the references passed in and out of an otherwise opaque interface.

4.5 Partiality of Functions

So far, we have treated functions at the logic level as total. This fits well with the semantic underpinnings of SMT-LIB, but the specification languages of some tools impose constraints on what constitutes well-formed terms and formulas that may occur in contracts.

As an example, Dafny will check all subterms in specifications to use partially defined operations only within their domain. For instance, writing $\forall k.a[k] = 0$ for a sequence, map, or array a is ill-defined because key k may not be in the domain of the index operator. Instead, one has to write $\forall k.k \in \text{dom}(a) \implies a[k] = 0$ or similar, and boolean operators are asymmetric insofar as earlier operands can constraint the well-formedness of later ones. In contrast, in SMT-LIB operators are left unspecified outside their domains.

The rationale behind such checks is not so much to avoid logical inconsistencies but to give feedback to the user. This is particularly helpful in auto-active tools that do not expose or offer ways to interact with the state of the back-end prover, which otherwise makes it really hard to debug when the definitions of operators fail to apply for out-of-domain arguments.

For Contract-LIB we decided not to impose similar requirements upfront and instead leave it up to the respective tool integration to address these. This means that a Contract-LIB script may correspond to an ill-formed specification in Dafny whereas the same script might be accepted by another tool that would then just have a hard time establishing the desired correctness guarantees. It is not out of question that in the future we will rectify this design choice based on experience, however, until then we make it an informal requirement that any tool integration

must not make assumptions about particular properties of operators outside their natural domains (e.g., not define $x/0 = 0$ for convenience).[10]

A third option is to detect and possibly synthesize additional preconditions that capture the missing checks for well-formedness, which is a desirable feature of utility tools provided for dealing with Contract-LIB scripts, but we leave this to future work.

5 Pragmatics of Tool Integration

When proposing a language such as Contract-LIB, which is intended to be integrated and used by the developers of various verification tools, it is important to make good tools available early on. Therefore, we provide the following toolchain which is written in Java and can be found as open source on GitHub[11]:

- SMT-LIB files that declare the signatures of the standardized set of types and functions for reference, as described in Sec. 4.2.
- An ANTLR4 grammar for Contract-LIB based on the SMT-LIB grammar (version 2.6). It supports all commands **`declare-*`** and **`define-*`** as well as **`assert`** from SMT-LIB, and additionally the new commands **`declare-abstractions`** and **`define-contract`**, as described in **??**.
- Interfaces of factories to create abstract syntax trees (AST) nodes for Contract-LIB (found in the package `org.contractlib.factory`). The key design point of this interface is that it is fully generic in the types of sorts, functions, and contracts, so that Java-based tools can directly instantiate their own internal representation with litte overhead. Importantly, the interface provides scoping information already, so that name resolution becomes straight-forward.
- A parser based on the ANTLR grammar which has to be instantiated with a given factory implementation and which produces AST (found in the package `org.contractlib.ast`).
- There is also a very rudimentary example AST implementation (for exmaple, it does not enforce typing constraints) and a corresponding factory implementation to showcase how to make use of this library.

The library is small and has no dependencies apart from ANTLR4 to encourage its adoption. Developers who want to import Contract-LIB into their Java-based tools have two choices to make use of this library: One can either rely on the factory interface to directly instantiate the internal representation. This step may already involve full name resolution and type checking, giving a very direct way to parse Contract-LIB scripts. Alternatively, one can rely on or adapt the given AST implementation, and convert that to the tool's internal data structures by traversing the fully-built AST. At the moment of writing the paper, we are working on an implementation to convert Contract-LIB into the data structures used in the KeY prover (basically an AST of JML).

[10] This is actually a sensible and logically sound choice taken up by Isabelle and other interactive proof systems.

[11] https://github.com/gernst/contract-lib

However, supporting the syntax of Contract-LIB is not sufficient for end-to-end integration between tools. Depending on the use-case, as discussed in Sec. 2, the parts of specifications and interface signatures that are exported/imported may vary, with some already being given or some others being generated, including proof scaffolding. As a consequence, there are further concerns that need to be considered in relation to a particular tool and a particular programming language. We think that collecting a catalogue of best-practices would be one way to document the experience of working with the format.

For example, Naming conventions for relating the identifier used in Contract-LIB's command `define-contract` to the source code could be useful. For example composing names from class names and method names, as we have done in the examples, seems like a reasonable choice. Alternatively, when importing Contract-LIB into one's tool, the user may be asked to define the mapping. Similarly, Language-specific naming conventions, such as having a first parameter `this` and a (possibly) last parameter `result` to capture the single receiver and single return value in object-oriented languages like Java may need to be be taken into account.

6 Related Work

Contract-LIB is by no means the first nor only formal language for the formulation of operation summaries or contracts. However, it has distinguishing features that set it aside from other formalisms and make it the ideal choice as exchange format of contracts for well-encapsulated modules.

Recently, the Model eXchange Interlingua MoXI [39] has been introduced as a standard interchange format for symbolic model checking challenges. While it is like Contract-LIB based on the semantic foundations of SMT-LIB, its focus is on state-based transition systems with temporal properties and trace-based semantics. As such it has a single internal state which does not capture well the use case of an unrestricted number encapsulated entities which is needed for modular/object-oriented verification. Hence, MoXI seems a suitable formalism that can import Contract-LIB specifications, but tool exchange for design-by-contract specification should be at higher-level and more tailored to the needs.

Avestan [42] is another proposed declarative modeling language based on the semantic foundations of SMT-LIB. Its syntax is based on Alloy [31] with the goal to add logical expressiveness not present in SMT-LIB (like transitive closure) and to be more readable. These goals are complementary to those of CONTRACT-LIB that stays within the logical bounds of SMT-LIB and extends its syntax. Again, it can be a suitable format that can import Contract-LIB specifications.

Z [43], Event-B [2], B [1], ASM [16] and related methods are state-based formalisms that would also allow the specification of contracts However, they are significantly more sophisticated, come with rich set-theoretic formal language and lack a simple machine-readable textual representation or even a commonly accepted set of features.

TPTP [40] is the machine-readable exchange format the automated theorem prover community with an extensive set of benchmark examples. TPTP would have been alternative to SMT-LIB to base Contract-LIB on, but as most deductive tools use SMT solvers (and only partly ATP solvers) as back-ends nowadays and as the support for ADTs is naturally stronger with SMT-LIB the latter was the more natural choice.

CASL [8] is an algebraic specification language that emphasizes on the specification of the semantics of (algebraic) data types which is not our goal. The shared semantics of data types is left a bit more implicit, analogous to that of SMT-LIB theories, and we do not wish to impose any particular axiomatization. Instead, tools are at liberty to translate types like sets/maps/sequences into their native format with potential additional nuances like partiality (cf. Sec. 4.5) and with particular strategies for proof automation.

Most of these languages are also rather complex, with sophisticated notation and expressive foundations. More importantly, they give a *primary* interpretation, whereas here, we allow tools to be flexible in certain ways, as long as the validity of the contract is retained (e.g. partiality).

There are a number of intermediate verification programming languages like Boogie [10], Why3 (or even possibly Dafny) that could serve as a common language for the community. However, experience shows that while these languages do have most features needed for the common ground they are already committed to particular methodological design decisions that make them unsuitable as basis for the common independent exchange language.

There have been some efforts towards translating translating specifications directly from one specification/verification framework to another or to base verification on a common low-level language. While this serves a similar goal, Contract-LIB wants to allow exchange between different formalisms without that one needs to know the idiosyncrasies at both ends of the translation. The Specification Translator [7] translates annotations of Java programs between VerCors, Krakatoa, and JML-based tools via an intermediate representation. The Karlsruhe Java Verification suite [34] serves the similar goal to integrate various Java verification tools but uses JML as the common exchange language. Similarly, Frama-C [33] is a framework for verification of C programs where different verification paradigms share a common ACSL specification. *b2w* [4] implements a translation of proof obligations from Boogie to Why3 and bridges the gap on a rather technical level.

There are ongoing efforts in the related area of proof assistants:[12] Project EuroProofNet "aims at boosting the interoperability and usability of proof systems", which is a much broader and more ambitious goal than ours, in parts because the languages of interactive proof assistants are much richer from a mathematical standpoint than SMT-LIB (e.g., including higher-order and dependent types) and because they consider translation of proofs as well [25,41].

[12] https://europroofnet.github.io/

Another interesting concurrent activity is the aim for a standardized contract language for Rust.[13] It involves practitioners and academics alike, which means that not only language-specific aspects are relevant but also contracts have a much wider range of purposes than here. While we aim to share our developments with this community, Rust's ownership system seems to imply that contracts relate three instead of two states, namely the state when a particular function call returns but also that when all borrows are returned, which in turn does not immediately fit into the semantic model of well-encapsulation discussed in Sec. 4.

7 Conclusion and Outlook

This paper introduced Contract-LIB, a standardized interchange format designed to enable the interoperability between deductive program verification tools and paradigms. It focuses on a concept of abstract contracts for stateful interfaces that verification tools can import and export. By leveraging the established SMT-LIB framework, Contract-LIB provides a concise, tool-agnostic representation of contract specifications that are a requirement of an efficient collaboration across different tools and programming languages. We outlined the syntactic and semantic core concepts of Contract-LIB that add data abstraction and operation contract definition mechanisms to SMT-LIB.

In future work, we will extend the prototypical reference implementation towards selected different verification tools covering different programming languages and will elaborate how well-encapsulation can be formulated across specification paradigms. We need to identify and collect the relevant data type and operation definitions that allow us to conduct larger case studies to show the utility and efficacy of the approach.

Acknowledgements. This work is based on a series of community discussions, started at Schloss Dagstuhl (Seminar 22451 on the Principles of Contract Languages, 2023) and refined more recently at the Lorentz Center (Seminar on Contract Languages, 2024). It is partially funded by the German Research Foundation (DFG) – 443187992.

References

1. Abrial, J.: The B-book - assigning programs to meanings. Cambridge University Press (1996). https://doi.org/10.1017/CBO9780511624162, https://doi.org/10.1017/CBO9780511624162
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book, LNCS, vol. 10001. Springer (2016)

[13] https://hackmd.io/w8AS2N09R76aXDFK9ogHBQ?view

4. Ameri, M., Furia, C.A.: Why just boogie? translating between intermediate verification languages. CoRR **abs/1601.00516** (2016), http://arxiv.org/abs/1601.00516
5. Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems. ACES-MB&WUCOR@ MoDELS **1508**, 19–26 (2015)
6. Armborst, L., Bos, P., van den Haak, L., Huisman, M., Rubbens, R., Ömer Şakar, , Tasche, P.: The vercors verifier: a progress report. In: Computer Aided Verification (CAV) 2024 (2024), to appear.
7. Armborst, L., Lathouwers, S., Huisman, M.: Joining forces! reusing contracts for deductive verifiers through automatic translation. In: International Conference on Integrated Formal Methods. pp. 153–171. Springer (2023)
8. Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL: the Common Algebraic Specification Language. Theoretical Computer Science **286**(2), 153–196 (2002). https://doi.org/https://doi.org/10.1016/S0304-3975(01)00368-1, https://www.sciencedirect.com/science/article/pii/S0304397501003681
9. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022, Proceedings, Part I. pp. 415–442. No. 13243 in LNCS, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
10. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17, https://doi.org/10.1007/11804192_17
11. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
12. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
13. Basin, D., Cremers, C., Dreier, J., Sasse, R.: Tamarin: verification of large-scale, real-world, cryptographic protocols. IEEE Security & Privacy **20**(3), 24–32 (2022)
14. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
15. Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hritcu, C., Ishtiaq, S., Kohlweiss, M., Leino, R., Lorch, J.R., et al.: Everest: Towards a verified, drop-in replacement of https. In: 2nd Summit on Advances in Programming Languages (2017)
16. Börger, E.: The ASM method for system design and analysis. A tutorial introduction. In: Gramlich, B. (ed.) Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3717, pp. 264–283. Springer (2005). https://doi.org/10.1007/11559306_15, https://doi.org/10.1007/11559306_15

17. Bornat, R.: Proving pointer programs in hoare logic. In: Backhouse, R.C., Oliveira, J.N. (eds.) Mathematics of Program Construction, 5th International Conference, MPC 2000, Ponte de Lima, Portugal, July 3-5, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1837, pp. 102–126. Springer (2000). https://doi.org/10.1007/10722010_8, https://doi.org/10.1007/10722010_8
18. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Tip: tons of inductive problems. In: International Conference on Intelligent Computer Mathematics. pp. 333–337. Springer (2015)
19. Cok, D.R.: Openjml: Software verification for java 7 using jml, openjdk, and eclipse. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014. EPTCS, vol. 149, pp. 79–92 (2014). https://doi.org/10.4204/EPTCS.149.8, https://doi.org/10.4204/EPTCS.149.8
20. Ernst, G., Huisman, M., Mostowski, W., Ulbrich, M.: Verifythis - verification competition with a human factor. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11429, pp. 176–195. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_12, https://doi.org/10.1007/978-3-030-17502-3_12
21. Ernst, G., Murray, T.: Seccsl: Security concurrent separation logic. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 208–230. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_13, https://doi.org/10.1007/978-3-030-25543-5_13
22. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: Kiv: overview and verifythis competition. International Journal on Software Tools for Technology Transfer **17**(6), 677 – 694 (2015). https://doi.org/10.1007/s10009-014-0308-3
23. Ernst, G., Weigl, A.: Verify this: Memcached—a practical long-term challenge for the integration of formal methods. In: International Conference on Integrated Formal Methods. pp. 82–89. Springer (2023)
24. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, Berlin, Germany. LNCS, vol. 4590, pp. 173–177. Springer (2007)
25. Gauthier, T., Kaliszyk, C.: Sharing hol4 and hol light proof knowledge. In: Logic for Programming, Artificial Intelligence, and Reasoning: 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings 20. pp. 372–386. Springer (2015)
26. Gogolla, M., Hamann, L.: Proving properties of operation contracts with test scenarios. In: Prevosto, V., Seceleanu, C. (eds.) Tests and Proofs. pp. 97–107. Springer Nature Switzerland, Cham (2023)
27. Hähnle, R., Huisman, M.: Deductive software verification: from pen-and-paper proofs to industrial tools. Computing and Software Science: State of the Art and Perspectives pp. 345–373 (2019)
28. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B.J., Wilhelm, R. (eds.) ESOP 86, European Symposium on Programming, Saarbrücken, Federal Republic of Germany, March 17-19, 1986, Proceedings. Lecture Notes in Computer Science, vol. 213, pp. 187–196. Springer (1986). https://doi.org/10.1007/3-540-16442-1_14, https://doi.org/10.1007/3-540-16442-1_14

29. Huisman, M., Monti, R., Ulbrich, M., Weigl, A.: The verifythis collaborative long term challenge. In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.) 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 (Dec 2020). https://doi.org/10.1007/978-3-030-64354-6_10, https://doi.org/10.1007/978-3-030-64354-6_10
30. Huisman, M., Ravara, A. (eds.): Formal Techniques for Distributed Objects, Components, and Systems - 43rd IFIP WG 6.1 International Conference, FORTE 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings, Lecture Notes in Computer Science, vol. 13910. Springer (2023). https://doi.org/10.1007/978-3-031-35355-0, https://doi.org/10.1007/978-3-031-35355-0
31. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002). https://doi.org/10.1145/505145.505149, https://doi.org/10.1145/505145.505149
32. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4, https://doi.org/10.1007/978-3-642-20398-5_4
33. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/S00165-014-0326-7, https://doi.org/10.1007/s00165-014-0326-7
34. Klamroth, J., Lanzinger, F., Pfeifer, W., Ulbrich, M.: The karlsruhe java verification suite. In: Ahrendt, W., Beckert, B., Bubel, R., Johnsen, E.B. (eds.) The Logic of Software. A Tasting Menu of Formal Methods - Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 13360, pp. 290–312. Springer (2022). https://doi.org/10.1007/978-3-031-08166-8_14, https://doi.org/10.1007/978-3-031-08166-8_14
35. M. Leino, K.R.: Accessible software verification with dafny. IEEE Software **34**(6), 94–97 (2017). https://doi.org/10.1109/MS.2017.4121212
36. Meyer, B.: Applying "design by contract". IEEE Computer **25**(10), 40–51 (Oct 1992)
37. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Pretschner, A., Peled, D., Hutzelmann, T. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 50, pp. 104–125. IOS Press (2017). https://doi.org/10.3233/978-1-61499-810-5-104, https://doi.org/10.3233/978-1-61499-810-5-104
38. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2 , btormc and boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 587–595. Springer International Publishing, Cham (2018)
39. Rozier, K.Y., Dureja, R., Irfan, A., Johannsen, C., Nukala, K., Shankar, N., Tinelli, C., Vardi, M.Y.: Moxi: An intermediate language for symbolic model. In: SPIN (2024)
40. Sutcliffe, G.: Stepping Stones in the TPTP World. In: Benzmüller, C., Heule, M., Schmidt, R. (eds.) Proceedings of the 12th International Joint Conference on Automated Reasoning. p. To appear. Lecture Notes in Artificial Intelligence (2024)

41. Thiré, F.: Interoperability between proof systems using the logical framework Dedukti. Ph.D. thesis, Université Paris-Saclay (2020)
42. Vakili, A., Day, N.A.: Avestan: A declarative modeling language based on smt-lib. In: 2012 4th International Workshop on Modeling in Software Engineering (MISE). pp. 36–42 (2012). https://doi.org/10.1109/MISE.2012.6226012
43. Woodcock, J.C.P., Davies, J.: Using Z - specification, refinement, and proof. Prentice Hall international series in computer science, Prentice Hall (1996)
44. Xu, M.: Research Report: Not All Move Specifications Are Created Equal. In: Proceedings of the 2024 Workshop on Language-Theoretic Security (LangSec). San Francisco, CA (5 2024)