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-Maximilians-University, Munich, Germany
gidon.ernst@lmu.de
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
{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 [28] 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], Frama-C [33], KeY [3], KIV [22], SecC [21], VerCors [6], VeriFast [31], Viper [37], Why3 [15] 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 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.

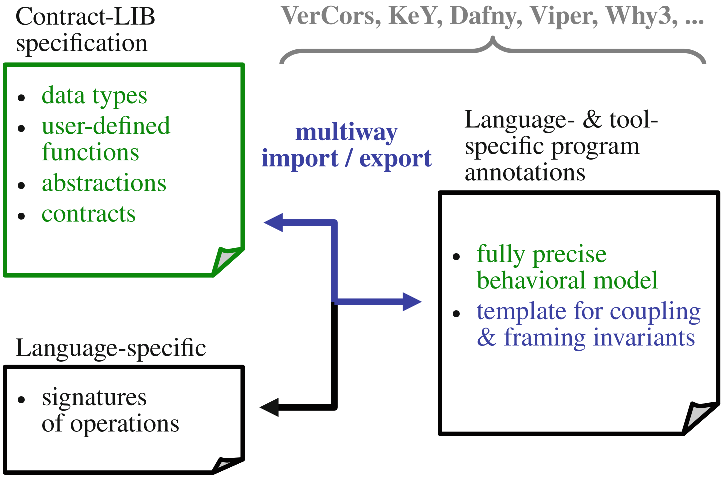

Fig. 1. Interoperability of Contract-LIB specifications. (Color figure online)

In an ideal world, verification tools would be *interoperable* in the sense that they can freely exchange specifications and other verification artifacts. In reality, 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).

Figure 1 illustrates sharing contracts between approaches and tools. At the top-left in green, language-agnostic Contract-LIB declarations describe behaviors and domain formalization of well-encapsulated modules (see Sect. 5). The language-specific structural information such as `interface` specifications in Java or header files in C are shown bottom-left in black. Together they enable exporting contracts to platform-specific specifications (on the right) for modular deductive verification with (methodology-dependent) coupling template and framing specifications. Conversely, platform-specific specifications can be imported as Contract-LIB declarations allowing Contract-LIB to act as an intermediate representation bridging verification paradigms.

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 formal software design scenarios that can benefit from an effective specification exchange technology (we come back to them in Sect. 7): tools can share formally verified standard libraries (like JDK or libc), heterogeneous projects with collaborating code bases written in different languages can be verified, and different verification techniques can join forces to verify parts serving different purposes in a larger project.

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 [11], BTOR2 [38], and the TPTP format [41] have had a huge impact and unlocked unforeseen applications.

First steps have already been made towards interoperability. For example, Armborst et al. [7] translate specifications and auxiliary annotations between JML, VerCors, and Krakatoa [24] for Java. In contrast, Klamroth et al. [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, regardless of the chosen methodology, specifications written in deductive tools typically encompass three different conceptual levels [45], 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 remarkably similar and stable across tools and hence can be a suitable target for an exchange format.

Therefore, Contract-LIB specifications only talk about abstractions that can be expressed by 2). Nevertheless, 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 and Structure: 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 (Sect. 2 and Sect. 3). We describe the aspects of the rationale behind its design that allows for easy adoption and integration into existing infrastructure. We discuss syntax, semantics, and showcase how different specification paradigms (Dynamic Frames, Separation Logic) can be connected to Contract-LIB (Sect. 3). We provide preliminary tool support in the form of a simple and stand-alone Java library (Sect. 4), which aims to be integrated with low effort with existing verification infrastructure. We demonstrate the applicability to a nontrivial example of a cache server implementation (Sect. 5). Finally, we discuss related efforts (Sect. 6) and give some outlook on potential use-cases for Contract-LIB (Sect. 7).

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

2 Contract-LIB Overview and Motivating Examples

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. In the terminology of Xu [45], Contract-LIB is a format that "formally and fully captures the requirements and nothing more" (point 2 in [45, Sec 1.]). In being based on this guiding principle, Contract-LIB can take on an opportunity for unifying those aspects in deductive specifications that —despite all the heterogeneity in methodology and language-specific aspects— are well-understood, stable, semantically unambiguous and widely supported in the better part of their functionality by most existing tools.

2.1 High-Level Design and Ingredients

The approach is strongly inspired by Data Refinement [39] and is illustrated in Fig. 2. 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 depicted to be grouped into logical units that independently can be coupled to the respective mathematical abstractions, e.g. by the use of `ghost` state, abstraction predicates, data structure invariants, or similar ideas. The verification method and tool then ensures that such abstractions are compatible with the dynamic behavior of the implementation, typically in the form of a simulation proof.

To define the green part of Fig. 2 above the double line, Contract-LIB offers three ingredients that are typically required to formulate abstract but fully precise behavioral models:

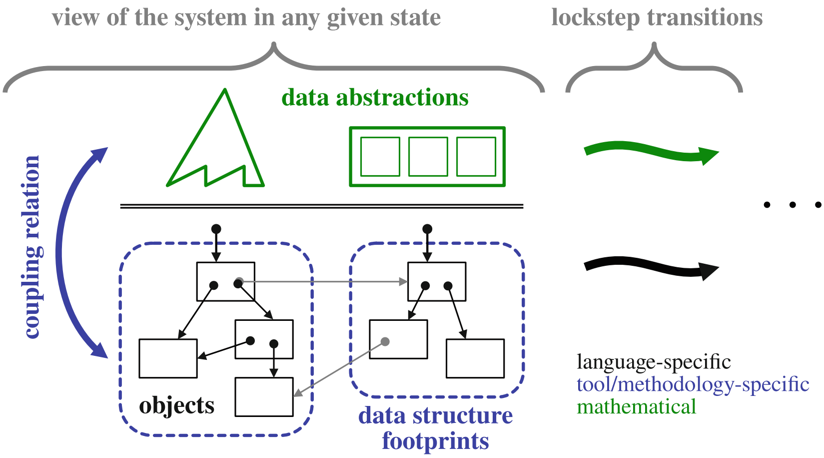

Fig. 2. Deductive verification of systems with respect to behavioral abstractions. Black: specific to the implementation language, green: shareable abstractions, blue: specific to the tool/methodology. (Color figure online)

1. An expressive *mathematical language* to model the application domain. Our proposal is based on SMT-LIB but standardizes commonly used mathematical types, finite sets, sequences, maps, together with widely-used functions.
2. A mechanism to declare *data abstractions* of the internal state of well-encapsulated modules. The key deliberation here is to allow mutation of state, possibly of multiple objects involved in a method call, but to carefully keep all data structures purely functional and thus avoid aliasing.
3. A mechanism to define *contracts of operations* or methods over these data abstractions, which relate inputs and outputs to possible state transitions.

Insofar, Contract-LIB aims for a point in the design space, which allows flexible contract definitions that cover many practical use-cases. However, for this first proposal, we deliberately avoid to incorporate more advanced and potentially language-specific features, including exceptions or mode sophisticated semantic models, such as a trace-based formulation of contracts. We leave such ideas for future work.

2.2 Example: Linked List Specifications in Different Languages

To give an intuition about the usage of Contract-LIB, Figs. 3 to 5 show how a method `add` of a linked list implementation can be specified in JML, Dafny, and VeriFast. The respective implementations discern a class `LinkedList` as the module boundary, which internally encapsulates the associated `Node` objects that represent the linked list in terms of their `value` and `next` attributes (as usual, not shown). Together, these objects constitute the lower part of Fig. 2 for any given linked list instance. Purpose of method `add` is to extend the list at the end by a single given element.

In JML (more precisely, the dialect used in KeY), the abstract state of the linked list is modeled with a ghost field `content` of sequence type. The state transition of the `add` method is characterized by a concatenation of the singleton sequence of the given value to the sequence from the pre-state (line 11). Apart from this functional specification, it is necessary to define the memory effect of the method as well. KeY's JML dialect supports Dynamic Frames [32], hence, this is done using a ghost field `footprint` of type `\locset` (a set of object-field pairs) to hold the memory footprint of the list which is also the set of memory locations the method is allowed to write (line 14). Finally, it is necessary to specify that the `add` method adds locations of newly-allocated objects to the footprint only, which is done with the so-called *swinging pivots* operator `\new_elems_fresh`.

In Dafny (Fig. 4), the specification is nearly identical, apart from the syntactic differences, and the representation invariant is encoded as a predicate that appears explicitly in pre- and postcondition, whereas in JML this is achieved via the invariant declaration in line 7. Again, a freshness condition for the footprint is required (line 15).

The specification in VeriFast (Fig. 5) contains more interesting differences. Instead of storing the abstract state in ghost variables, it is described by parameters of two predicates `ls` and `inv`. The contract of method `add` is defined by how the `content` parameter of predicate `inv` changes: The question mark in line 15 binds `content` as a universally quantified variable that can be referred to in the postcondition. Appending an element to the abstract sequence is described by the recursive function `snoc`, this functionality is supported out-of-the-box in JML and Dafny. The writable memory footprint needs not be explicitly specified since such memory layout is built-in into the Separation Logic of VeriFast.

In these small examples, we can already see the required ingredients for Contract-LIB outlined in Sect. 2.1. We explain how these are translated and represented into the Contract-LIB model shown in Fig. 6:

1. *Expressive mathematical language.* The Contract-LIB represents the sequence `content` of type **`\seq`** in JML, **`seq<int>`** in Dafny, and **`list<int>`** in VeriFast, in terms of a parametric sort as **`(Seq int)`**, cf. line 4. In the contract definition, described below, we make use of functions on sequences, like concatenation `seq.++` and `seq.unit`. The larger example in Sect. 5 furthermore makes use of standard SMT-LIB definitions to model the application domain.
2. *Data abstractions.* In Contract-LIB, the command **`declare-abstractions`** is available to define those data structures that represent abstractions of stateful interfaces, as can be seen in Fig. 6. This command is semantically analogous to the command **`declare-datatypes`** of SMT-LIB, which defines a family of potentially mutually recursive data types, but the more specific command signals the purpose of the induced sort `LinkedList` as an abstraction of class `LinkedList`. For our example, it defines a single constructor, also called `LinkedList`, that has the sequence as its parameter (lines 1–4). The identifier `LinkedList.content` represents the respective selector which extracts that parameter from a value of the type (cf. lines 16 and 18). Note that while the

```
class LinkedList {
 /*@ nullable @*/ Node head;

 //@ ghost \seq content;
 //@ ghost \locset footprint;

 //@ invariant content == ...;

 /*@ normal_behaviour
   @ ensures
      content == \old(content) + \seq(v);
   @ ensures
      \new_elems_fresh(footprint);
   @ assignable footprint;
   @*/
 void add(int v) { ... }
 ...
}
```

Fig. 3. Linked lists in JML, ...

```
class LinkedList {
 var head: Node?

 ghost var content: seq<int>
 ghost var footprint: set<object>

 ghost predicate Valid() { ... }

 method add(v: int)
  modifies footprint
  requires Valid()
  ensures  Valid()
  ensures
    content == old(content) + [v] &&
    fresh(footprint - old(footprint))
 { ... }
 ...
}
```

Fig. 4. in Dafny, ...

```
/*@
predicate ls(Node hd, list<int> xs)
  = ...;

predicate inv(LinkedList l, list<int> xs)
  = l.head |-> ?hd &*& ls(hd, xs);

fixpoint list<int>
  snoc(list<int> xs, int y) { ... } @*/

class LinkedList {
 Node head;

 void add(int v)
  //@ requires inv(this,?content);
  //@ ensures  inv(this,snoc(content,v));
 { ... }
 ...
}
```

Fig. 5. in VeriFast, ...

```
(declare-abstractions
  ((LinkedList 0))
  ((LinkedList ; abstract data
    ((LinkedList.content (Seq int))))))

(define-contract LinkedList.add
  ; parameters of the contract
  ; with input/output modes
  ((this (inout LinkedList))
   (v    (in    int)))

  ; list of possible behaviors,
  ; here consisting of a single
  ; precondition/postcondition pair
  ((true
    (= (LinkedList.content this)
      (seq.++
        (old (LinkedList.content this))
        (seq.unit v))))))
```

Fig. 6. ... and in Contract-LIB.

abstraction declared in this simple example just is a sequence of integers, it is possible to use parametric types as well (cf. Sect. 3.4).

3. *Contracts of operations.* Contracts are pre-/postcondition pairs that relate the input parameters and return values to a transition between a pair of states. To describe such contracts, the command **`define-contract`** is available in Contract-LIB (starting at line 6 in the example). A contract declares its name, here `LinkedList.add`, as well as its signature. In order to represent methods which affect the abstractions of multiple objects at the same time, all parameters including the receiver `this` are listed. Here, `this` is denoted to be of type `LinkedList`, defined by **`declare-abstraction`**, and parameter `v` is of integer type.
   As Contract-LIB is based on a purely functional view of data (i.e., there are no objects that can be modified implicitly via some heap), all parameters are annotated with an *input/output mode*. Here, `this` is declared to be `inout`, which means that its value may change and with it the abstract view on the underlying implementation type.
   In addition, the contract specifies a set of possible behaviors, here a single one, comprised of a respective precondition, here `true`, and a postcondition. Here, the postcondition is simply a translation of the equations from Figs. 3 to 5 that relate the value of `content` in the pre-state (marked with `old`) to the post-state.

3 Technical Realization of Contract-LIB

We formally define the syntax of Contract-LIB as well as its semantics on which integration with established methodologies rests. We then discuss some aspects related to theory definitions and the question of partiality of functions.

3.1 Syntax

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`**.

The grammar for extensions specific to Contract-LIB is shown in Fig. 7. For ingredient 2 *data abstractions* in Sect. 2, we introduce a new command, **`declare-abstractions`**, which has exactly the same structure and semantics as **`declare-datatypes`**, i.e., it defines a (freely-generated/inductive) algebraic data type, but which additionally signals that this data type serves as the abstraction of a stateful encapsulated module. This information can be picked up by tools that implement various transformations as envisioned in Fig. 1.

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

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.

For ingredient 3 *contract definitions* in Sect. 2 we introduce another new command **`define-contract`**, which captures possible state transitions on abstractions introduced via the command **`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 a **`define-fun`** command, however, instead of pairing names just with types as in `(x Int)` we additionally annotate an input/output mode as shown already in Fig. 6. For example, `(this (inout` $\langle type \rangle$`))` encodes that the abstraction of the object referenced by `this` may change during a method call and which reflects a corresponding **`modifies`** clause in Dafny or **`assignable`** clause in JML.

Each entry in the list of $\langle contract \rangle$s occurring inside a given **`define-contract`** is a pair of `(`ϕ_j ψ_j`)` of terms of SMT-LIB type `Bool` that specifies some potential behavior of the respective operation: If some precondition ϕ_j holds, then the operation is guaranteed validate the corresponding postcondition ψ_j.

3.2 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 $\mathbf{true}$ 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

$$c \in \mathcal{C} = (x_1 \colon \mu_1 \sigma_1, \dots x_n \colon \mu_n \sigma_n;\ (\varphi_1, \psi_1), \dots, (\varphi_n, \psi_k)) \tag{1}$$

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

A contract is well-specified if variables with **out** do not occur in any φ_j and not inside $\mathbf{old}(\_)$ in any ψ_j, as we prefer to not fix an interpretation that may be counter-intuitive to some users.

The semantics $[\![c]\!]$ of such a contract is intended to contain pairs of interpretations, so that $(\mathcal{I}, \mathcal{I}') \in [\![c]\!]$ represents a *successful* application of the contract within its precondition in state represented by $\mathcal{I}$ transitioning to state represented by $\mathcal{I}'$ upon returning. In addition, we have to account for the possibility that preconditions are violated, which is represented as $(\mathcal{I}, \bot) \in [\![c]\!]$. Introducing separate $\bot$-elements helps to distinguish this situation from the absence of successor states, e.g., when the postcondition is false or the method diverges (see e.g. [39]).

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

where $s \equiv_{\mathbf{in}} s'$ denotes that $s(x) = s'(x)$ for all $x_i \colon \mathbf{in}\, \sigma_i$ to encode that contracts are never allowed to change input variables and $j = 1, \dots, k$ ranges over all pre-/postcondition pairs of the contract. The relational semantics $s, s' \models \psi_j$ evaluates each occurrence of a variable inside $\mathbf{old}(\_)$ with respect to s and otherwise with respect to s'. We emphasize that these definitions are standard.

Having multiple pre-/postconditons is somewhat a convenience feature, we can join them up into a single precondition/postcondition pair. For a contract c given as in Eq. (1), the summarizing precondition is simply the conjunction of individual ones, whereas the summarizing postcondition must take care to guarantee the appropriate individual postcondition depending on the situation.

$$\phi \ \equiv \ \phi_1 \wedge \cdots \wedge \phi_k \qquad\qquad \psi \ \equiv \ (\phi_1 \implies \psi_1) \wedge \cdots \wedge (\phi_k \implies \psi_n)$$

The rationale to include this as a feature of Contract-LIB is to retain more full information when translating such multi-behavior contracts between tools.

3.3 Integration with Dynamic Frames and Separation Logic

For this discussion, we assume a simple standard semantic model with a global heap $h\colon H$ where $H = R \rightharpoonup O$ is the type of heaps modeled as partial maps from references R to object content O. For example, references R may be memory addresses and O are mathematical tuples or records that store the values of the object's attributes, but other choices are possible such as a fine-grained model in which R are pairs of addresses and attribute names.

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 Sect. 5), 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 $\mathcal{I} \equiv_{\mathrm{free}(t)} \mathcal{I}'$ 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 pred-

```
⟨command⟩ ::= (declare-fun (par (⟨symbol⟩+) ((⟨sort⟩*) ⟨sort⟩)))
            | (define-fun  (par (⟨symbol⟩+) ((⟨sorted_var⟩*) ⟨sort⟩)⟨term⟩) )
            | (assert      (par (⟨symbol⟩+) ⟨term⟩))
            | ...
```

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

icates α that are solely introduced for the sake of tracking ownership of the references passed in and out of an otherwise opaque interface.

3.4 Standardized Theories and Polymorphism

As we have seen in Sect. 5, 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 container types like 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 such container types 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.[1] Here, the list of functions supported for the respective data types as well as their naming mirrors that of cvc5 [9][2] 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 (Sect. 3.5).

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 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.

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

3.5 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 $\forall k. k \in \mathrm{dom}(a) \implies a[k] = 0$ 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[3]).

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.

4 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[4]:

- SMT-LIB files that declare the signatures of the standardized set of types and functions for reference, as described in Sect. 3.4.

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

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

- 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 Sect. 3.1.
- 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 example, 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).

However, supporting the syntax of Contract-LIB is not sufficient for end-to-end integration between tools. Depending on the use-case (cf. Sect. 7), 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 taken into account.

5 Modeling `Memcached` in Dafny and Contract-LIB

While Sect. 2 intends to demonstrate that Contract-LIB is flexible to accommodate a variety of common specification styles, here we aim to show that the format handles more complex case-studies well. The example is taken from the VerifyThis long-term challenge [23].[5] The specification is available at https://gist.github.com/gernst/eb0028af1961b6df4740b5ceca628cf9.

The system modeled here is `memcached`. It is an open-source 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. `Memcached` differs from a plain map-like interface insofar that entries have a lifetime after which the association expires and that entries can implicitly be evicted due to memory pressure.

5.1 Data Model of `memcached`

`Memcached` is accessed via a simple text-protocol as exemplified in Fig. 9. Here, an entry is stored under key `token` with flags `0`, a timeout of 30 s, and a length of 6 bytes, the subsequent 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.

In the specification language of Dafny [35], we model this protocol not in terms of its textual/binary representation[6] but instead formulate a higher-level abstraction using a combination of types like `Key`, `Flags`, and `Data`, which are left abstract, shown at the top of Fig. 10, and structured data types representing the content of protocol messages. `Entry` tracks the respective `data` and metadata comprised of `flags`, `expiry` timeout, and a version counter `unique`, which 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). `Result` is used to encode return values of operations which are further described below, where the `Values` constructor is used to encode both responses shown in Fig. 9, once with a singleton sequence and once with the empty sequence; remarking that there is an operation `gets`, too, which possibly returns multiple values. The model makes use of auxiliary definitions, too, such as predicate `live`, which 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.

The Contract-LIB encoding of this part of the model is straight-forward and shown in Fig. 11. Abstract types are translated to **`declare-sort`** whereas Dafny's `datatype` declarations are represented as SMT-LIB `declare-datatype`(s). Recall that in SMT-LIB, sorts can be parametric, and that the number `0` attached to the sorts, resp. the empty parameter list of sort `Time` just declares that these sorts have no parameters. Predicate `live` is encoded in SMT-LIB here as a function

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

[6] https://github.com/memcached/memcached/blob/master/doc/protocol.txt

```
// 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. 9. 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 | ...

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

Fig. 10. Dafny: data model of `memcached`.

```
(declare-sort Key   0)
(declare-sort Flags 0)
(declare-sort Data  0)
(define-sort  Time () int)

(declare-datatypes
  ((Entry 0) (Result 0))
  (((Entry (data Data) (flags Flags) (unique nat) (expiry (Option Time))))
   ((Values (entries (Seq (Pair Key Entry))))
             (Stored) (Touched) (Deleted) ...)))

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

Fig. 11. Contract-LIB representation of the data model of the `memcached` protocol as an encoding of the Dafny model shown in Fig. 10.

definition that relies on pattern matching, a semantically equivalent alternative would be with a function declaration and two axioms.

5.2 `Memcached` as a Well-Encapsulated Stateful Module

Figure 12 shows the specification of the stateful interface of `memcached`, realized as **`trait`** `Code` in Dafny. It has two attributes, `entries` and `uniques` that are marked as **`ghost`** and are therefore not part of the implementation. Rather, they represent the abstract view of the internal state of the system and therefore give rise to a precise account of `Cache`'s dynamic behavior. Attribute `entries` 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.

```
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. 12. Dafny specification of the state of the data abstraction, expressed over the data types from Fig. 10 as ghost attributes of an interface specification realized as a **`trait`**. Moreover, the trait already provisions the connection to possible implementations.

Because **`trait`** `Code` represents a well-encapsulated *stateful* module, it is not encoded as plain sorts like those shown in Fig. 11, but using the new command **`declare-abstractions`**, specifically introduced in Contract-LIB for this purpose as shown in Fig. 13. The intended interpretation is that it defines a sort `Cache`

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

(declare-fun Valid ((this Cache)) Bool)

(assert (forall ((this Cache))
    (=> (Valid this)
        (forall ((key Key))
            (=> (map.contains key (Cache.entries this))
                ...))))))
```

Fig. 13. Representation of the Cache's state in Contract-LIB, as a translation of the two **ghost** attributes shown in Fig. 12. Predicate `Valid` can be partially specified already, insofar it entails properties that are expressible over the specification data types.

that is semantically equivalent to as if it was specified by **declare-datatypes**, but serves the more specific purpose to encode the abstraction of a module with the same name, here `Cache`. As selector names in SMT-LIB are global, we choose to prefix them with the name of the abstraction as in `Cache.entries`.

In addition to the state of the trait, Fig. 12 illustrates the plumbing with which one typically connects the **ghost** state towards the implementation in Dafny. It follows Dafny's methodology for specifying object-oriented programs based on the Dynamic Frames [32] methodology. In addition to the abstract representation of the state, the specification tracks the footprint which this abstraction depends on as an additional attribute (cf. Sect. 3.3). The coupling to the implementation data structures is then captured by the predicate `Valid`, which is framed explicitly using a **reads** clause and which takes the role of an invariant. If desired, we can choose to translate this predicate and some of the consequences it entails, too, as shown in Fig. 13, thanks to the expressiveness of SMT-LIB.

5.3 Modeling Dynamic Behaviors of `memcached` Operations

Of the many operations supported by `memcached`, Fig. 14 defines the contracts of the methods implementing protocol commands **get** and **set**, respectively, as introduced in Fig. 9. The contracts of these two operations have two parts: The first part, corresponding to tool-specific connection 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 on `footprint`.

```
trait Cache {
  // ...

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

    // Behavioral model, split into two possible cases
    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)

    // Connection to implementation
    modifies this, footprint
    requires Valid()
    ensures  Valid()
    ensures  old(footprint) <= footprint && fresh(footprint - old(footprint))

    // Behavioral model over the abstraction
    ensures  unique !in old(uniques) && uniques == old(uniques) + {unique}
    ensures  entries == old(entries)[key := Entry(data, flags, unique, expiry)]
    ensures  result == Stored
}
```

Fig. 14. Contract definitions of two operations of `memcached`.

The second part of each contract 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.

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: The 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.

```
(define-contract Cache.get
    ((this (in   Cache) (key (in Key))))
    ((; first behavior: the key is exists and the entry is live
         (and (map.contains  (Cache.entries this) key)
              (live (map.get (Cache.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))
    ((; precondition omitted
      (and
        ; ...
        (= (Cache.entries this)
           (map.update (old Cache.entries this)
                key (Entry data flags unique expiry)))))))
```

Fig. 15. `Memcached` operations as contract definitions in Contract-LIB.

The corresponding Contract-LIB representation is shown in Fig. 15. We omit much of the translation of the actual formulas from Dafny to SMT-LIB, but we want to emphasize how mutable state, inputs, and outputs are represented as part of the signature. For example, `(key (in Key))` declares the `key` parameter to be an input argument.

In relation to the semantic model from Sect. 3.3, however, calling method `Set` induces a change to the underlying heap from h to h', where `old(entries)` refers to the attribute `entries` of the object residing in the old heap at h[`this`], which is different from h'[`this`] even if the value of reference `this` remains the same.

This transition is mirrored in Contract-LIB by declaring that parameter `(this (inout Cache))` is both read and updated by the corresponding call (for `Set`). However, while `this` in Dafny is a *reference*, in Contract-LIB, the abstraction is a *value* that has copy semantics and does not require any indirection via the heap.

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) [40] has been proposed 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 of 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 [43] is another proposed declarative modeling language based on the semantic foundations of SMT-LIB. Its syntax is based on Alloy [30] 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 [44], 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 [41] is the machine-readable exchange format of the automated theorem prover community with an extensive set of benchmark examples. TPTP would have been an alternative to SMT-LIB to base Contract-LIB on, but as most deductive tools use SMT solvers (and only partly ATP solvers) as backends 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. Sect. 3.5) 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 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 needing 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:[7] 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 [26,42].

Another interesting concurrent activity is the aim for a standardized contract language for Rust.[8] 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 Sect. 3.

7 Outlook: Use Cases for Contract-LIB

We identified a number of possible promising collaborative verification scenarios for which we see that the tool interoperability and method integration made possible by Contract-LIB will prove very beneficial.

Co-development of Larger Projects Composed of Modules: A first scenario for interoperability is using multiple deductive verification tools in the context of a larger project with multiple components in the same programming language. Given that a formal specification often touches different formal aspects, having a choice between different tools with complementary strengths is desirable. For

[7] https://europroofnet.github.io/
[8] https://hackmd.io/w8AS2N09R76aXDFK9ogHBQ?view

example, VerCors can be used to prove linearizability of a concurrent data structure and the resulting Contract-LIB contract be communicated to the OpenJML [19] verifier that scales better for lightweight properties to establish the client's sequential correctness.

Applications Comprised of Multiple Implementation Languages: Cross-language verification projects are challenging [25] unless tools already support multiple programming languages. In one scenario, native code (compiled from C) is invoked from a higher-level programming language (such as Java). Reasons for using native code include efficiency, required low-level functionalities, or depending on a library written in a lower-level language. Another prime use-case is the use of a more abstract programming language like Dafny to implement critical infrastructure. Dafny programs can be translated into multiple target programming languages, and there are project that use this infrastructure at scale. For critical applications, one may want to verify the interaction between the parts in different languages and Contract-LIB offers a way to mechanically translate the interface contracts between Dafny and the verifier used for the conventional language.

Shared Specifications of Standard Libraries. Shareable abstractions of standard libraries like Java's JDK or `libc` are an important step for the whole research field to make tools more "industry-ready". While small but intricate verification challenges usually do not require comprehensive library support, scaling to larger and more realistic code bases will require 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 still underdeveloped, with notable exceptions e.g. provided in JML[9] or in VeriFast[10]. Contract-LIB will provide the means to seamlessly exchange library specifications across tools for the same implementation language, even across specification paradigms. Making such specifications available can also tie deductive techniques closer to fully automated software model checkers that participate for example in SV-COMP [13].

Formal Specifications of Normative Documents. Request for Comments (RFCs) establish important internet standards for protocols like HTTP and SSL. There exist a number of protocol formalizations in the literature like Tamarin [12], but these rely on modeling languages of specialized tools and are rarely connected to implementations, with notable exceptions like Project Everest [14]. Exporting formal models as Contract-LIB specifications and sharing them 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 [23,29]

[9] https://github.com/OpenJML/Specs
[10] https://github.com/verifast/verifast/tree/master/bin

may benefit from a common interchange format. Competition organizers can use this format to specify verification tasks providing participants with a clear goal for their verified implementations. Conversely, the abstract specifications could be a basis for a machine-assisted analysis and comparison of the results, which today are judged by humans [20]. For the longer-running competitions, Contract-LIB enables tighter collaboration in the sense of the scenarios mentioned before [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 [27]. Possibly, there is some common kernel that can be translated losslessly between Contract-LIB and the recently proposed Model eXchange Interlingua (MoXI) [40]. 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.

The prototypical reference implementation currently supports only few target languages and features. In future work, we will extend it towards a diverse range of 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.

8 Conclusion

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.

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). We are grateful for the valuable feedback from the reviewers, which helped to improve the paper significantly. This work 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
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)
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., et al.: 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., et al.: CASL: the common algebraic specification language. Theoret. Comput. Sci. **286**(2), 153–196 (2002). https://doi.org/10.1016/S0304-3975(01)00368-1. https://www.sciencedirect.com/science/article/pii/S0304397501003681
9. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (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, Cham (2005). 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). www.SMT-LIB.org
12. Basin, D., Cremers, C., Dreier, J., Sasse, R.: Tamarin: verification of large-scale, real-world, cryptographic protocols. IEEE Secur. Privacy **20**(3), 24–32 (2022)
13. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). LNCS, vol. 14572, pp. 299–329. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57256-2_15
14. Bhargavan, K., et al.: Everest: towards a verified, drop-in replacement of https. In: 2nd Summit on Advances in Programming Languages (2017)
15. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64. Wrocław, Poland (August 2011)
16. Börger, E.: The ASM method for system design and analysis. A tutorial introduction. In: Gramlich, B. (ed.) FroCoS 2005. LNCS (LNAI), vol. 3717, pp. 264–283. Springer, Heidelberg (2005). https://doi.org/10.1007/11559306_15

17. Bornat, R.: Proving pointer programs in hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000). 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
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.) TACAS 2019. LNCS, vol. 11429, pp. 176–195. Springer, Cham (2019). 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.) CAV 2019. LNCS, vol. 11562, pp. 208–230. Springer, Cham (2019). 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. Int. J. Softw. Tools Technol. 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, Cham (2023). https://doi.org/10.1007/978-3-031-47705-8_5
24. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_21
25. Furia, C.A., Tiwari, A.: Challenges of multilingual program specification and analysis. In: ISoLA 2024. LNCS. Springer (2024)
26. Gauthier, T., Kaliszyk, C.: Sharing HOL4 and HOL light proof knowledge. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 372–386. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_26
27. 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, Cham (2023)
28. 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)
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. LNCS, vol. 12345, pp. 246–260. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64354-6_10
30. 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
31. 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., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617,

pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
32. Kassios, I.T.: The dynamic frames theory. Formal Aspects Comput. **23**(3), 267–288 (2011). https://doi.org/10.1007/S00165-010-0152-5
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
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
35. M. Leino, K.R.: Accessible software verification with dafny. IEEE Softw. **34**(6), 94–97 (2017). https://doi.org/10.1109/MS.2017.4121212
36. Meyer, B.: Applying "design by contract". IEEE Comput. **25**(10), 40–51 (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
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, Cham (2018)
39. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-oriented Proof Theories and their Comparison, Cambridge Tracts in Theoretical Computer Science, vol. 46. Cambridge University Press (1998)
40. Rozier, K.Y., et al.: Moxi: an intermediate language for symbolic model. In: SPIN (2024)
41. 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)
42. Thiré, F.: Interoperability between proof systems using the logical framework Dedukti. Ph.D. thesis, Université Paris-Saclay (2020)
43. 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
44. Woodcock, J.C.P., Davies, J.: Using Z - specification, refinement, and proof. Prentice Hall international series in computer science, Prentice Hall (1996)
45. 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, May 2024