Multi-perspective correctness of programs

Eduard Kamburjan[1,2] and Dilian Gurov[3]

[1] IT University of Copenhagen, Denmark
eduard.kamburjan@itu.dk
[2] University of Oslo, Norway
[3] KTH Royal Institute of Technology, Sweden
dilian@kth.se

Abstract. Traditionally, programs are formally specified and verified with respect to their *computational domain*, disregarding the domain in which they are to be applied. This, however, is inadequate for programs that simulate processes in a specific application domain, or programs that generate data that must conform to external, domain-specific specifications. Such programs need also to be correct with respect to their *application domain*. This work presents a Hoare Logic that manages two different perspectives on a program during a correctness proof: the computational view and the domain view. This enables us to specify the correctness of a program in terms of the domain *without* referring to the computational details, but at the same time to interpret failed proof attempts in the domain. For domain specification, we illustrate the use of description logics and base our approach on semantic lifting, an approach to interpret a program as a knowledge graph. We present a calculus that uses translations between both kinds of assertions, thus separating the concerns in specification, but enabling the use of description logic in verification.

1 Introduction

Programs are typically developed in the context of some domain and must truthfully represent the knowledge about this domain to be considered correct. When reasoning about a program, it is, therefore, interpreted as both a *computational structure* and as a *model* for the domain: At the very minimum, the implemented computational logic must correspond to the intended business logic, but in extreme cases, such as simulators, the domain is directly encoded in the program. Even if the program does not encode any business logic in its statements, it may generate output data that must be correct w.r.t. some external specification from the domain, for example for data exchange.

Deductive program verification [13] approaches employ logical reasoning to prove functional correctness, but currently rely on a singular specification language that focuses on the computational nature of programs: a program logic (PL), such as Hoare logic, that refers to statements and program elements (e.g., variables, expressions) within a first-order logic. Consequently, specification languages in deductive verification are disconnected from domain-focused logical approaches to knowledge representation, such as Description Logics (DL). DLs are an established tool to model domain knowledge with elaborate pragmatics in the form of, e.g., ontologies; yet, making use of them for program specification and verification remains unexplored.

However, a direct translation of a DL specification into the contracts of the program logic would still not enable us to interpret intermediate steps of proofs in the domain: We require a way to manage both views (computational view and domain view) and their relation throughout a proof attempt for functional correctness, such that (a) we can investigate an intermediate step from both perspectives, e.g., to extract information from failed proofs, and (b) use both DL and PL for functional specification of programs.

We investigate reasoning about the correctness of programs with specifications for both the *implementation* (i.e., the program specifics) and its connection to the application *domain*. Domain-specific specification, in the form of description logic assertions, enables domain experts to be involved in modeling and programming, by giving them a tool to express their constraints without exposing them to implementation details. We aim to retain as much of the DL pragmatics during verification as possible, while making use of their logical foundation to recover assertions about the program: Failed proof attempts should be interpreted and explained [7,30] in the domain. Similarly, keeping DL separate from program assertions enables the use of specialized solvers. Nonetheless, these assertions are used by a Hoare logic that operates only on the program state, and not on its interpretation in the domain.

Specification. To connect program state and description logics, we use ideas from *semantically lifted programs* [18]. The state of a semantically lifted program is *lifted* into the domain in the form of a *knowledge graph*. This graph can then be enriched with DL axioms to interpret the program state in terms of the domain.

At the core of our approach are *two-tier* specifications. A two-tier assertion $\left\{\begin{smallmatrix}\Delta\\ \Phi\end{smallmatrix}\right\}$, also written $\{\Delta;\ \Phi\}$, contains an assertion Φ about the program state, and an assertion Δ about the domain, which specifies the *lifted* state in terms of the domain. To connect the two assertions in the calculus, we lift not only the state, but also *the specifications*, in order to recover information for Φ from Δ.

Fig. 1 illustrates the relations between state specification, the lifted state specification and the domain specification containing the lifted state specification. It is critical that the domain specification is using only the notions and vocabulary of the enriched state, and is not describing the lifted program state directly – it is describing the lifted state *enriched with additional axioms*. Thus, the program logic must be able to infer possible program states from the domain specification.

Verification. Consider a program that models the assembly of a car. Its final state must be data that describes the assembled car. This data is specified using a domain ontology, which expresses concepts such as `Has4Wheels`(c). This concept states that car c has

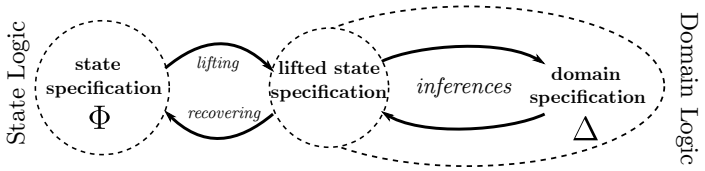

Fig. 1: Relation between domain and state specifications in their respective logics.

four wheels. Let us consider the following statement, that sets the variable `wheels` to the parameter `nrWhls`. In the domain, the specification expresses that after execution, the modelled car has four wheels. For the implementation, it states that the parameter `nrWhls` must be 4.

$$\left\{\begin{matrix} - \\ \texttt{nrWhls} \doteq 4 \end{matrix}\right\} \texttt{wheels} := \texttt{nrWhls} \left\{\begin{matrix} \texttt{Has4Wheels}(c) \\ - \end{matrix}\right\} \tag{1}$$

From the perspective of the domain experts, the precondition cannot be stated, since they do not know how the car c is modelled, and are not aware of the encoding of wheels as integers, or even the very *existence* of the variable `wheels`. Thus, both parts of the contract are stated from different perspectives and uphold the separation of concerns between domain and computation. But given a suitable specification lifting, we can transform the above two-tier triple into the following, and derive that to ensure the domain post-condition, the state must have set the variable `wheels` to 4. This is easily shown using a standard assignment rule.

$$\left\{\begin{matrix} - \\ \texttt{nrWhls} \doteq 4 \end{matrix}\right\} \texttt{wheels} := \texttt{nrWhls} \left\{\begin{matrix} \texttt{Has4Wheels}(c) \\ \texttt{wheels} \doteq 4 \end{matrix}\right\} \tag{2}$$

The connection between the two tiers enables us to investigate a failed proof attempt. Consider the specification $\{-;\ \texttt{wheels} \doteq 3\}$. It does not allow one to infer that the car is member of the concept `Has4Wheels`, even though the user expects it. But using the lifting we can use standard abductive and deductive reasoning in DL [21] to explain why the expected entailment cannot be derived: From the implementation specification $\texttt{wheels} \doteq 3$ we cannot deduce that $\texttt{Has4Wheels}(c)$. But when we try to abduce an explanation for the specification target $\texttt{Has4Wheels}(c)$, we arrive at the implementation specification $\texttt{wheels} \doteq 4$, which very clearly indicates where the program fails to output data according to the specification.

This enables us to tackle another bottleneck of deductive program verification – the notoriously difficult task to understand intermediate proof states [11], where specification is intermingled with proof and implementation specifics.

To enable this overall explanation approach, abductive reasoning in the domain logic, deductive reasoning in the program logic, and lifting must be compatible with each other. In the remainer of this work, we give a precise characterization when the logics and lifting can be composed in a sound way.

Contribution. Our main contributions are (1) a Hoare logic that manages two connected views on a program correctness proof, (2) a characterization when a domain logic can be plugged in into the two-tier logic, and (3) an instantiation for description logic. The remainder of this paper gives a precise description of the connection between state and domain specification required to set up a two-tier Hoare logic to enable such inferences.

2 Preliminaries

We give the basic definitions for the logic that we use to describe the states of the implemented program directly, as well as definitions for description logics for domain

specification. To simplify terminology, we refer to the former as *state logic* and to the later as *domain logic*. Both logics are based on semantics defined over values Val that include data values, which in our case will be the integers $\mathbb{Z}$ only, and names ∇, which correspond to nominals in description logic.

To ease the later connection between the two logics, we split function symbols into functions that result in a name, and data functions that result in a data value, and consider the set of program variables in the signature.

Definition 1 (Signatures). *A* state signature $\Sigma = \langle \mathtt{V}, \mathtt{F}, \mathtt{F}_d, \mathtt{P} \rangle$ *is a tuple of variable names* $\mathtt{V}$*, function symbols* $\mathtt{F}$*, data function symbols* $\mathtt{F}_d$*, and predicate symbols* $\mathtt{P}$*. A* domain signature $\Sigma_d = \langle \mathtt{N}, \mathtt{R}, \mathtt{T}, \mathtt{A} \rangle$ *is a tuple of nominals* $\mathtt{N}$*, abstract roles* $\mathtt{R}$*, concrete roles* $\mathtt{T}$*, and atomic concepts* $\mathtt{A}$*. We say that a (state or domain) signature* Σ *is a* sub-signature *of* Σ' *(*$\Sigma \subseteq \Sigma'$*) if all its components are subsets.*

In general, we refrain from treating arities formally and assume the usual syntactical checks to make formulas and interpretation respect the arity of symbols.

Definition 2 (Interpretations). *A* state interpretation $\mathcal{I}$ *over a state signature* $\Sigma = \langle \mathtt{V}, \mathtt{F}, \mathtt{F}_d, \mathtt{P} \rangle$ *is a map from: 1. function symbols* $f \in \mathtt{F}$ *to functions from values to values, 2. data function symbols* $f \in \mathtt{F}_d$ *to functions from values to integers, and 3. predicate symbols* $p \in \mathtt{P}$ *to functions from values to booleans.*

A domain interpretation $\mathcal{I}_d$ *over a domain signature* $\Sigma_d = \langle \mathtt{N}, \mathtt{R}, \mathtt{T}, \mathtt{A} \rangle$ *is a map from: 1. nominal symbols* $o \in \mathtt{N}$ *to names in* ∇*, 2. abstract role symbols* $R \in \mathtt{R}$ *to relations over names, 3. concrete role symbols* $T \in \mathtt{T}$ *to relations over names and* $\mathbb{Z}$*, and 4. atomic concepts symbols* $A \in \mathtt{A}$ *to subsets of* ∇*. The set of all state interpretations is denoted* $\mathbf{I}$*, while the set of all domain interpretations is denoted* $\mathbf{I}_d$*.*

Program variables are not interpreted by I, but are part of the program state. Numerical constants are treated as special 0-ary function symbols.

Definition 3 (States and State Logic). *Let* V *be the set of program variables. A* program state $\sigma : \mathsf{V} \to \mathsf{Val}$ *is a mapping from variables to values. Let* $\mathtt{S}$ *denote the set of all program states. Let* $\Sigma = \langle \mathtt{V}, \mathtt{F}, \mathtt{F}_d, \mathtt{P} \rangle$ *be a state signature. State formulas* Φ *are defined by the following grammar, where* v *ranges over* $\mathtt{V}$*,* p *over* $\mathtt{P}$*, and* f *over* $\mathtt{F} \cup \mathtt{F}_d$*. The set of all state formulas over* Σ *is denoted* $\Phi(\Sigma)$*.*

$$\Phi ::= \Phi \wedge \Phi \mid \neg\Phi \mid t \doteq t \mid p(\bar{t}) \qquad\qquad t ::= v \mid f(\bar{t})$$

The semantics of the state logic $\sigma, \mathcal{I} \models \Phi$ *is defined relative to a program state and a state interpretation, and is given in the technical report [17].*

We use the usual abbreviations such as $\vee$ and $\rightarrow$ and omit $\mathcal{I}$ in the satisfiability relation if it is understood. We define a simple description logic, $\mathcal{ALCO}(\mathsf{D})$, following mostly the semantics of Horrocks and Sattler [16] for $\mathcal{SHON}(\mathsf{D})$. We stress that our approach is not relying on any particular property of this logic (or any description logic), except for the presence of data types, but we consider description logics as the most suited formalism for domain specification in our framework.

Definition 4 (Description Logic). *Let $\Sigma_d = \langle \mathtt{N}, \mathtt{R}, \mathtt{T}, \mathtt{A} \rangle$ be a domain signature. The syntax of domain formulas δ is defined by the following grammar, where A ranges over $\mathtt{A}$, R over $\mathtt{R}$, T over $\mathtt{T}$, o over $\mathtt{N}$, and n over literals from $\mathbb{Z}$. The set of all domain formulas over Σ_d is denoted $\Delta(\Sigma_d)$. We use Δ to range over sets of domain formulas.*

$$\delta ::= C \sqsubseteq C \mid C(o) \mid R(o,o) \mid R(o,n)$$
$$C ::= \top \mid \bot \mid A \mid \neg C \mid C \sqcup C \mid C \sqcap C \mid \exists R.\, C \mid \forall R.\, C \mid \exists T.\, n \mid \forall T.\, n$$

The semantics $\mathcal{I}_d \models \delta$ is defined relative to a domain interpretation, and is given in the technical report [17]. We use the usual logic abbreviations, such as $\equiv$.

Given a formula Φ (resp. a domain formula Δ), we denote the signature containing just the symbols it uses by $\mathsf{sig}\ \Phi$ (resp. $\mathsf{sig}\ \Delta$). Semantic entailment is defined as usual: Given two formulas Δ, Δ', we say that Δ entails Δ' ($\Delta \models \Delta'$) if every interpretation that satisfies Δ, also satisfies Δ'. This naturally generalizes to sets of formulas. Given a set of domain formulas $\mathbf{K}$, we write $\Delta \models^{\mathbf{K}} \Delta'$ to denote that every interpretation that satisfies formula Δ and all elements of $\mathbf{K}$ also satisfies Δ'.

3 Motivating Example

Scenario. Consider a program that models the assembly of a small car, where a car is considered to be small if it has two doors and four wheels. This can be formalized in the domain logic using the following formula.

$$\mathtt{SmallCar} \equiv \mathtt{Has2Doors} \sqcap \mathtt{Has4Wheels} \sqcap \mathtt{Car}$$

Additionally, everything that has a body is a car, and everything that has a chassis has a body. For doors, wheels, and the body of the car, we can formulate the following, to express that everything that has 2 doors is part of the concept `Has2Doors`, and analogously for `HasFourDoors` and `HasBody`. We use the common pattern of *stubs* [22]: instead of modeling the number of doors using a `hasDoors` relation that maps to a number, we use a relation `doors` that maps to an individual that has some number associated with it using relation `hasValue`. As we see later, we can relate the stubs with variables in the programming language to connect the two formalisms.

$$\mathtt{HasChassis} \sqsubseteq \mathtt{HasBody} \sqsubseteq \mathtt{Car} \qquad \exists \mathtt{doors}.\exists \mathtt{hasValue}.2 \equiv \mathtt{Has2Doors}$$
$$\exists \mathtt{wheels}.\exists \mathtt{hasValue}.4 \equiv \mathtt{Has4Wheels} \qquad \exists \mathtt{body}.\mathtt{NonZero} \equiv \mathtt{HasBody}$$
$$\neg\exists \mathtt{hasValue}.0 \equiv \mathtt{NonZero}$$

The program itself is given in Fig. 2. The assembly is old-fashioned: it starts with a chassis, and has three substeps, namely adding the body by assigning a non-zero id, then adding the wheels (`addWheels`), and adding the doors. It is operating on a single car, which is modelled by the variable `bodyId` for the id of the body, where $\mathtt{bodyId} = 0$ models that no body is attached, the variable `doors` which models the number of doors on the body, and `wheels`, which models the number of wheels. The assembled car has a chassis, which is not explicit in the program.

Specification. Our aim is to specify that procedure `assembly` indeed assembles a small car. However, the domain expert has no knowledge about the computational encoding of the process, e.g., that the wheels are modelled as a global variable. The contract of procedure `assembly` is as follows. In the beginning we get the number of doors (which must be 2) and the id of the body (which must be non-null), and in the end it is a small car. The individual c is implicit in the program – in our example, the program assembles exactly one car, but this information is not relevant for the domain expert. As specifications, we use pairs $\left\{\begin{smallmatrix}\Delta\\ \Phi\end{smallmatrix}\right\}$ that express that domain formula Δ and state formula Φ must hold.

$$\left\{\begin{matrix}-\\ \texttt{nrDrs} \doteq 2 \wedge \texttt{bodyId} \neq 0\end{matrix}\right\}\texttt{assembly}()\left\{\begin{matrix}\texttt{SmallCar}(c)\\ -\end{matrix}\right\}$$

Let us now turn to the specification of `addWheels`. The domain specification explains what is expected from the view of the car assembly (the car already has a chassis), while the implementation specification ($\texttt{nrWhls} = 4$) specifies additional conditions *not visible in the domain* to ensure correctness. The former is specified by the domain expert, while the latter is added by the programmer.

$$\left\{\begin{matrix}-\\ \texttt{nrWhls} \doteq 4\end{matrix}\right\}\texttt{addWheels}(\texttt{nrWhls})\left\{\begin{matrix}\texttt{Has4Wheels}(c)\\ -\end{matrix}\right\}$$

The fact that the car has a chassis is not part of the specification — $\texttt{HasChassis}(c)$ is not the domain precondition. The reason is that the modelled car by default has a chassis. Thus, the fact $\texttt{HasChassis}(c)$ is part of the connection of the state with the background knowledge.

The post-condition is obvious - it states that afterwards the car being assembled is part of class `Has4Wheels`. Note that the implementation details are hidden from the domain experts – they do not know how c is modelled, whether it always has a chassis in the program, or whether this is explicit. They are, thus, not able to state the state precondition, as they are not aware of the encoding of wheels. Thus, the two parts of the contracts are stated from different perspectives and uphold the *separation of concerns* between domain and computation. Furthermore, we stress that the specification at the level of procedure contracts enables the participation of the domain expert in a more fine-grained specification, without being exposing too many technicalities, but requires that we must be able to switch between a domain and a state view in the middle of the analyzed statement.

```
1 var bodyId = 0; var wheels = 0; var doors = 0;
2 proc addWheels(nrWhls) begin wheels := nrWhls; end;
3 proc assembly(id, nrDrs) begin
4   bodyId := id; addWheels(4); doors := nrDrs;
5 end
```

Fig. 2: An assembly line program.

$\texttt{hasValue}(\texttt{wheelsVar}, 4)$ $\quad$ $\texttt{HasChassis}(c)$

$\texttt{wheels}(c, \texttt{wheelsVar})$ $\quad$ $\texttt{body}(c, \texttt{bodyVar})$ $\quad$ $\texttt{doors}(c, \texttt{doorsVar})$

Fig. 3: The first formula is (part of) the lifted state, the other connect lifted state and domain.

Verification. To verify that `addWheels` adheres to its specification, we have to show that its procedure body indeed transforms a car into one with four wheels, which is exactly Eq. 1 (p. 3). In a classical weakest precondition calculus, we would now substitute `wheels` by `nrWhls` in the post-condition – the post-condition obviously needs to be $\texttt{wheels} \doteq 4$. But in out setting we only have the domain specification. Instead of introducing redundancy in the specification, which would also break our separation between tasks for the domain expert and tasks for the programmer, we can retrieve a state post-condition as follows.

At its basis, we rely on semantic lifting, which generates a domain state from a program state. Let us consider the program state σ_0 with $\sigma_0(\texttt{wheels}) = 4$. Its lifting consists of axioms for the program state, information about the domain *and* additional formulas that connect the domain concepts with those describing the lifted program state. Those are given in Fig. 3. Note that the resulting knowledge graph has two parts: lifted program state, and domain knowledge. However, the domain specification is only concerned with the domain knowledge. The first part is generic for the program, e.g., the existence of variables – instead of designing a new lifting for every application, this *direct lifting* can be used as a basis to simplify modeling [18].

Still, we can deduce knowledge about the state: For example, if the car has four wheels (i.e., $\texttt{Has4Wheels}(c)$), then the corresponding variable must be set to 4 (i.e., $\texttt{hasValue}(\texttt{wheels}, 4)$). This information, in turn, can be interpreted in the program logic as $\texttt{wheels} \doteq 4$, in order to strengthen our specification into Eq. 2 (p. 3).

Using the rule for assignment, we can prove the correctness of `addWheels` w.r.t. to its specification. We must consider the relation of $\texttt{Has4Wheels}(c)$ and $\texttt{wheels} \doteq 4$ – as the program must establish both conditions, but only controls the state post-conditions, the state post-condition $\texttt{wheels} \doteq 4$ must imply the complete domain post-condition $\texttt{Has4Wheels}(c)$. Having established our example and illustrated the challenges therein. we now give a formal treatment of the underlying Hoare logic. We return to the assembly line, and show it correct, after presenting the calculus in Sec. 5.

Explanation. Given the above framework, it is easy to see that we can not only investigate two perspectives on pre- and post-conditions of procedures, but also explain proof state. For example, consider a slight variant of the program, where `addWheelsBug` contains a bug: it always assigns 3 to the `wheels` variable, but uses the same contract.

```
1 var bodyId = 0; var wheels = 0; var doors = 0;
2 proc addWheelsBug(nrWhls) begin wheels := 3;  end;
3 proc assembly(id, nrDrs) begin
4   bodyId := id; addWheelsBug(4); doors := nrDrs;
5 end
```

Applying the weakest-precondition calculus results in the following triple, which cannot be proven: the post-condition is not consistent and the state specification does not entail the domain specification.

$$\left\{\begin{array}{c} - \\ \texttt{nrWhls} \doteq 4 \end{array}\right\} \texttt{wheels} := 3 \left\{\begin{array}{c} \texttt{Has4Wheels}(c) \\ \texttt{wheels} \doteq 3 \end{array}\right\}$$

One can use *abduction* to retrieve a possible explanation of how $\texttt{Has4Wheels}(c)$ could be inferred, and obtain the axiom $\texttt{hasValue}(\texttt{wheelsVar}, 4)$ to see where the program went wrong – critically, it uses the domain to explain what the state should be, an explanation mechanism normally not available in program verification.

4 A Two-Tier Hoare Logic

Our task now is to ensure that the program indeed models the assembly of a small car at the domain level, not just through the name of its variables and procedures. Our approach is based on two-tier assertions: A two-tier assertion has two parts, or tiers, in different logics, that are connected through a lifting mechanism for translation. Each tier corresponds to a different perspective. To do so, we must first define how to interpret a state in the domain and define the semantic lifting of a state.

Definition 5 (State and Specification Lifting). *A* state lifting *is defined as a function* $\mu : \mathtt{S} \times \mathbf{I} \to \mathbf{I}^d$ *from program models to domain models. A* specification lifting $\hat{\mu} : \Phi(\Sigma) \to \Delta(\Sigma^d)$ *is a mapping from program formulas to domain formulas. We denote the signature of the images of* $\hat{\mu}$ *as its* kernel*, written* $\mathbf{ker}\ \hat{\mu} = \bigcup_{\Psi \in \Phi(\Sigma)} \mathsf{sig}\big(\hat{\mu}(\Psi)\big)$.

State and specification lifting must be compatible, such that if a state satisfies a state formula, then its lifting must satisfy the lifted formula. This is required for the soundness of lifting $\hat{\mu}$ – the state lifting μ is not used in the calculus we give later.

Definition 6 (Compatibility). *A pair* $(\mu, \hat{\mu})$ *is* compatible *w.r.t. a state interpretation* $\mathcal{I}$ *and a set of domain formulas* $\mathbf{K}$ *iff lifting state and formula preserves satisfaction:* $\forall \sigma \in \mathtt{S}.\, (\sigma, \mathcal{I} \models \Phi \Rightarrow \mu(\sigma) \models^{\mathbf{K}} \hat{\mu}(\Phi))$.

The domain logic is less expressive than the state logic, as its task is to specify in terms of the domain without exposing implementation details. Applying $\hat{\mu}$ allows one to interpret an intermediate state specification in the domain, for example to examine what this state is modelling. Similarly, the domain specification is not only part of the pre- and post-condition of the program, but also part of the pre- and post-condition of *procedures*. The lifting $\hat{\mu}$ is, thus, needed to add information to apply these contracts.

However, the program itself is analyzed in terms of the state logic. For example, the effect of an assignment can be clearly expressed for the state specification, but not for the domain. Here, we require to *recover* information from the domain specification by applying the inverse $\hat{\mu}^{-1}$. Consider the state specification $\phi = \texttt{wheels} \doteq 4$ and the domain specification $\delta = \texttt{HasFourWheels}(c)$. The lifting $\hat{\mu}$ enables us, together with further inferences and assuming a fitting pair of liftings, to derive δ from ϕ, and the recovering mapping $\hat{\mu}^{-1}$ enables us to derive ϕ from δ. Before we connect state and

domain specification further, we give a direct lifting. The recovering mapping $\hat{\mu}^{-1}$ is only well-defined on the kernel of $\hat{\mu}$.

The characteristic formula χ_σ of a state σ is defined as $\bigwedge_{v \in \mathbf{dom}\sigma} v \doteq \sigma(v)$.

Definition 7 (Direct Lifting). *The specification lifting $\hat{\mu}_{\mathsf{direct}}$ is defined as follows, where $\mathtt{var}_v$ is a symbol identified by its index.*

$$\hat{\mu}_{\mathsf{direct}}(v \doteq n) = \{\mathsf{hasValue}(\mathtt{var}_v, n)\} \qquad \hat{\mu}_{\mathsf{direct}}(v \neq 0) = \{\mathtt{NonZero}(\mathtt{var}_v)\}$$
$$\hat{\mu}_{\mathsf{direct}}(\Phi_1 \wedge \Phi_2) = \hat{\mu}_{\mathsf{direct}}(\Phi_1) \cup \hat{\mu}_{\mathsf{direct}}(\Phi_2)$$

The state lifting is defined by: $\mu_{\mathsf{direct}} = \mathcal{I}$ such that $\mathcal{I} \models \hat{\mu}_{\mathsf{direct}}(\chi_\sigma)$.

The pair $(\mu_{\mathsf{direct}}, \hat{\mu}_{\mathsf{direct}})$ is compatible, and the example in Fig. 3 is an application of it with $\mathbf{K} = \{\mathtt{HasChassis}(c), \mathtt{wheels}(c, \mathtt{wheelsVar}), \dots\}$. The variables are also modelled as stubs – the formula $\mathtt{wheels}(c, \mathtt{wheelsVar})$ indeed expresses that the variable $\mathtt{wheelsVar}$ (i.e., $\mathtt{var}_{\mathtt{wheels}}$) is the stub that can be used in the domain to reason about the wheels of the car c. The kernel of $\hat{\mu}_{\mathsf{direct}}$ is as follows

$$\mathbf{ker}\ \hat{\mu}_{\mathsf{direct}} = \{\mathtt{hasValue}, \mathtt{NonZero}\} \cup \{\mathtt{var}_v \mid v \in \mathbf{dom}\ \sigma\} \cup \mathsf{sig}(\mathbf{K})$$

Note the explicit addition of `NonZero`, which enables us to lift (and recover from) more abstract specifications than characteristic formulas.

4.1 Assertions

Equipped with a formal definition of lifting, we now define specifications that have both a domain and a state component. We refer to such specifications as *two-tier assertions*.

Definition 8 (Two-Tier Assertion). *Let $(\mu, \hat{\mu})$ be a compatible set of liftings (w.r.t. some $\mathcal{I}$ and $\mathbf{K}$). Let Δ range over sets of domain formulas over Σ_d and Φ over state formulas over Σ. A* two-tier assertion *has the form*

$$\left\{\begin{matrix}\Delta \\ \Phi\end{matrix}\right\}$$

written $\{\Delta;\ \Phi\}$ for brevity, and has the following semantics

$$\sigma \models^{\mathbf{K}} \{\Delta;\ \Phi\} \quad \textit{iff} \quad \sigma, \mathcal{I} \models \Phi \textit{ and } \mu(\sigma), \hat{\mu}(\Phi) \models^{\mathbf{K}} \Delta$$

We say that a two-tier assertion is strongly consistent *if $\hat{\mu}(\Phi) \models^{\mathbf{K}} \Delta$.*

In a strongly consistent assertion, the domain is determined entirely by the state, which is exactly the condition we discussed above for post-conditions.

A contract is where domain specification can be used – we do not expect the domain expert to annotate intermediate specification in sequences of statements, but to interact with the developer on the level of procedures and other, abstracting language constructs.

$$\mathsf{cond}(P, R_1, R_2) = \left\{(\sigma,\sigma') \;\middle|\; (\sigma \in P \wedge (\sigma,\sigma') \in R_1) \vee (\sigma \notin P \wedge (\sigma,\sigma') \in R_2)\right\}$$

$$[\![\mathtt{v} := \mathsf{expr}]\!]_{\mathbf{C},\mathbf{K}} = \left\{(\sigma,\sigma') \mid \sigma' = \sigma[\mathtt{v} \mapsto \mathcal{A}[\![\mathsf{expr}]\!]]\right\}$$

$$[\![s_1\ ;\ s_2]\!]_{\mathbf{C},\mathbf{K}} = [\![s_1]\!]_{\mathbf{C},\mathbf{K}} \circ [\![s_2]\!]_{\mathbf{C},\mathbf{K}} \qquad [\![\mathbf{skip}]\!]_{\mathbf{C},\mathbf{K}} = \{(\sigma,\sigma) \mid \sigma \in \mathtt{S}\}$$

$$[\![\mathbf{if}\ (\mathsf{expr})\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}]\!]_{\mathbf{C},\mathbf{K}} = \mathsf{cond}(\mathcal{B}[\![\mathsf{expr}]\!], [\![s_1]\!]_{\mathbf{C},\mathbf{K}}, [\![s_2]\!]_{\mathbf{C},\mathbf{K}})$$

$$[\![\mathbf{while}\ (\mathsf{expr})\ \mathbf{do}\ s\ \mathbf{od}]\!]_{\mathbf{C},\mathbf{K}} = \mathsf{LFP}\ F_{\mathbf{C},\mathbf{K}}$$

$$\text{where } F_{\mathbf{C},\mathbf{K}}(R) = \mathsf{cond}(\mathcal{B}[\![\mathsf{expr}]\!], [\![s]\!]_{\mathbf{C},\mathbf{K}} \circ R, \mathsf{id}_{\mathtt{S}})$$

$$[\![\mathtt{p}(\mathsf{expr})]\!]_{\mathbf{C},\mathbf{K}} = \left\{(\sigma,\sigma') \mid \sigma \models_{\mathbf{K}} \mathtt{Pre}_{\mathbf{C}}(\mathtt{p}, \mathcal{A}[\![\mathsf{expr}]\!]_\sigma) \wedge \sigma' \models_{\mathbf{K}} \mathtt{Post}_{\mathbf{C}}(\mathtt{p}, \mathcal{A}[\![\mathsf{expr}]\!])\right\}$$

Fig. 4: Program semantics for statements s.

Definition 9 (Procedure Contract). *A* contract *for a procedure* $\mathtt{p}(\mathtt{v})$ *is a pair of two-tier assertions* $\left(\left\{\begin{smallmatrix}\Delta_1^{\mathtt{p}}\\ \Phi_1^{\mathtt{p}}\end{smallmatrix}\right\}, \left\{\begin{smallmatrix}\Delta_2^{\mathtt{p}}\\ \Phi_2^{\mathtt{p}}\end{smallmatrix}\right\}\right)$*, called the* precondition *and the* postcondition*, respectively. The set of all contracts in a program is denoted* $\mathbf{C}$*. Retrieving the precondition (resp. postcondition) of a procedure* p *with parameter* e *replacing its argument variable* v *is denoted:*

$$\mathtt{Pre}_{\mathbf{C}}(\mathtt{p},\mathtt{e}) = \left\{\begin{smallmatrix}\Delta_1^{\mathtt{p}}\\ \Phi_1^{\mathtt{p}}[\mathtt{v} \setminus \mathtt{e}]\end{smallmatrix}\right\} \qquad \mathtt{Post}_{\mathbf{C}}(\mathtt{p},\mathtt{e}) = \left\{\begin{smallmatrix}\Delta_2^{\mathtt{p}}\\ \Phi_2^{\mathtt{p}}[\mathtt{v} \setminus \mathtt{e}]\end{smallmatrix}\right\}$$

We can now introduce a simple, imperative programming language with procedure calls that operates on states. As we are not concerned with expressive power here, we limit expressions to a minimum, and procedures to only one parameter. The semantics of the language is relative to a set of contracts. This simplifies the later definition; a non-relative version is easily obtained by inlining. All variables are global, and we forbid recursive calls in preference of loops.

Definition 10. *The* syntax *of our programming language is defined by the following grammar. Let* v *range over variables,* n *over literals, and* p *over procedure names.*

$$\begin{aligned}
\mathsf{prog} &::= \overline{\mathbf{var}\ \mathtt{v}\ =\ \mathsf{expr}\ ;}\ \overline{\mathsf{proc}} \qquad \mathsf{proc} ::= \mathtt{p}(\mathtt{v})\ \mathbf{begin}\ s\ \mathbf{end} && \textit{procedures}\\
\mathsf{expr} &::= n \mid \mathtt{v} && \textit{expressions}\\
s &::= \mathtt{v}\ :=\ \mathsf{expr};\ \mid s\ ;\ s \mid \mathbf{if}\ (\mathsf{expr})\ \mathbf{then}\ s\ \mathbf{else}\ s\ \mathbf{fi}\\
&\quad \mid \mathbf{while}\ (\mathsf{expr})\ \mathbf{do}\ s\ \mathbf{od} \mid \mathtt{p}(\mathsf{expr}) \mid \mathbf{skip} && \textit{statements}
\end{aligned}$$

The semantics *of our language is defined relative to a contract set* $\mathbf{C}$ *and a set of formulas* $\mathbf{K}$*, as a binary relation on states, i.e., as a* denotational semantics $[\![s]\!] \subseteq \mathtt{S} \times \mathtt{S}$*, shown in Fig. 4, where we use* $\mathsf{LFP}F$ *to denote the least fixed-point of a function* F*.*

Our semantics is *procedure-modular*, i.e., we define the semantics of a procedure call as the semantics of the contract of the called procedure. This is why our semantics is based on binary relations on states rather than on partial functions. The evaluation functions $\mathcal{A}[\![\cdot]\!]$ and $\mathcal{B}[\![\cdot]\!]$ of arithmetic and boolean expressions are standard and omitted for brevity. For an extended treatment of the used kind of denotational semantics we refer to the standard texts [31,28], and for details about the treatment of contracts to [12].

4.2 Hoare Triples

A two-tier assertion specifies a state, while two-tier Hoare triples relate the initial and final states of a program execution: If the precondition holds in the initial state, and the program terminates, then the postcondition holds in the final state. In our case, the pre- and post-conditions are lifted assertions.

Definition 11 (Two-Tier Hoare Triple). *A* two-tier Hoare triple *with respect to a compatible mapping* $(\mu, \hat{\mu})$ *has the following form*

$$\{\Delta_1;\ \Phi_1\}s\{\Delta_2;\ \Phi_2\}$$

with the expected semantics (given sets of formulas $\mathbf{K}$ *and contracts* $\mathbf{C}$*)*

$$\models_{\mathbf{C},\mathbf{K}} \begin{Bmatrix}\Delta_1\\ \Phi_1\end{Bmatrix} s \begin{Bmatrix}\Delta_2\\ \Phi_2\end{Bmatrix} \quad \textit{iff.} \quad \forall(\sigma,\sigma') \in [\![s]\!]_{\mathbf{C},\mathbf{K}} \cdot \left(\sigma \models_{\mathbf{K}} \begin{Bmatrix}\Delta_1\\ \Phi_1\end{Bmatrix} \to \sigma' \models_{\mathbf{K}} \begin{Bmatrix}\Delta_2\\ \Phi_2\end{Bmatrix}\right)$$

Let us now turn to the recovering mapping $\hat{\mu}^{-1}$. This faces the challenge of "delifting" arbitrarily formulas, while $\hat{\mu}$ must merely lift a limited set of expressions, on which we can easily enforce a normal form. Recovering could only operate on the limited signature $\mathbf{ker}\ \hat{\mu}$. For this reason, we must be able to infer formulas from a domain specification that are within this limited signature. While in some cases we may be able to deduce them, in the general case we may have to rely on *abduction*.

Fortunately, abduction is feasible in our setup – we have a clear notion of abductibles through the signature, and we only require formulas about individuals, not arbitrary formulas. This is exactly the well-explored setting of ABox abduction with abductibles [21]. We abstract from the exact mechanism to generate the inversible kernel and merely assume some function that realizes it.

Definition 12 (Kernel-Generator). *Let* $\hat{\mu}$ *be a specification lifting. A* kernel-generator *is a function* $\alpha_{\mathbf{K}} : 2^{\Delta} \to 2^{\Delta}$ *that, given a set of domain formulas* Δ*, generates another set of domain formulas* $\alpha_{\mathbf{K}}(\Delta)$ *such that* $\mathsf{sig}\big(\alpha_{\mathbf{K}}(\Delta)\big) \subseteq \mathbf{ker}\ \hat{\mu}$ *and* $\alpha_{\mathbf{K}}(\Delta) \models_{\mathbf{K}} \Delta$*.*

A kernel-generator can either perform abduction or deduction. In the case of abduction, most formulas are essentially implications that have $\mathbf{ker}\ \hat{\mu}$ as the consequent and the rest of the signature in the antecedent (e.g., *"if the program variable has this value, then the domain individual belongs to this concept"*). Abduction is not sound, but may still be useful to generate a condition for program correctness. One can precisely specify abduction with the usual conditions on $\alpha_{\mathbf{K}}$ [27]. In case of deduction, the kernel-generator infers the only possible values for the variables.

Given a state specification, we can lift it using $\hat{\mu}$ to a set of formulas with signature $\mathbf{ker}\ \hat{\mu}$. From there, we can deduce further formulas about the domain using description logic reasoning. To compare a given set of domain formulas, we use the kernel generator α to get formulas with signature $\mathbf{ker}\ \hat{\mu}$, from which we can inverse the lifting $\hat{\mu}^{-1}$. Implication on assertions is lifted as expected.

Definition 13. *One two-tier assertion implies another if the following holds.*

$$\{\Delta;\ \Phi\} \to_{\mathbf{K}} \{\Delta';\ \Phi'\} \quad \textit{iff.} \quad \forall\sigma.\,(\sigma \models_{\mathbf{K}} \{\Delta;\ \Phi\} \to \sigma \models_{\mathbf{K}} \{\Delta';\ \Phi'\})$$

Formally, we can now express the relations between $\hat{\mu}$, α and $\hat{\mu}^{-1}$ with the following lemma, on which our calculus will heavily rely.

Lemma 1. *The following three implications and equivalences hold.*

1. *Generating the kernel of a domain specification implies the input:*

$$\{\Delta, \alpha_{\mathbf{K}}(\Delta);\ \Phi\} \to_{\mathbf{K}} \{\Delta;\ \Phi\}$$

2. *Adding lifted specification preserves satisfiability:* $\{\Delta;\ \Phi\} \leftrightarrow_{\mathbf{K}} \{\Delta, \hat{\mu}(\Phi);\ \Phi\}$.
3. *Adding recovered specification preserves satisfiability:*

$$\left\{\begin{matrix}\Delta, \Delta' \\ \Phi\end{matrix}\right\} \leftrightarrow_{\mathbf{K}} \left\{\begin{matrix}\Delta, \Delta' \\ \Phi \wedge \hat{\mu}^{-1}(\Delta')\end{matrix}\right\} \text{ where } \mathsf{sig}(\Delta') \subseteq \mathbf{ker}\ \hat{\mu}$$

5 A Calculus for the Two-Tier Hoare Logic

The calculus combines the concepts introduced previously by integrating two systems of rules: The first implements a weakest-precondition calculus on the implementation-specification for each statement, except for procedure calls. These rules erase domain information, as every change in the implementation can potentially affect any formula in the lifted specification. The second system of rules implements the kernel-generation, lifting and recovering that enables us to restore this information, or to add information to the implementation from the domain before it is erased. Verification is compositional, i.e., local to a single procedure and relative to the context: the contracts and background knowledge. Our judgement is, thus, verifying a lifted Hoare triple in a fixed context.

Definition 14 (Calculus). *Let* $\mathbf{C}$ *be the set of contracts for a given program, and* $\mathbf{K}$ *a set of formulas. A* judgement *of the calculus has the following form.*

$$\mathbf{C}, \mathbf{K} \vdash \{\Delta_1;\ \Phi_1\} s \{\Delta_2;\ \Phi_2\}$$

We say that the judgement is valid*, if for every state, in every terminating run where all procedures adhere to their respective contract, the Hoare triple holds.*

$$\mathbf{C}, \mathbf{K} \vdash \left\{\begin{matrix}\Delta_1 \\ \Phi_1\end{matrix}\right\} s \left\{\begin{matrix}\Delta_2 \\ \Phi_2\end{matrix}\right\} \quad \textit{iff} \quad \models_{\mathbf{C},\mathbf{K}} \left\{\begin{matrix}\Delta_1 \\ \Phi_1\end{matrix}\right\} s \left\{\begin{matrix}\Delta_2 \\ \Phi_2\end{matrix}\right\}$$

Let $P_1, \ldots, P_n$ *and* C *be judgements. A* rule *has the following form.*

$$\frac{P_1 \ldots P_n}{C}$$

The rule is sound*, if validity of premises* $P_1, \ldots, P_n$ *implies validity of the conclusion* C*.*

To connect the two specifications, we require a set of rules to modify the lifted assertions that serve as pre- and postconditions, as well as to strengthen the precondition or weaken the postcondition. These rules, given in Fig. 5, are all given relative to some $\mathbf{K}, \alpha_{\mathbf{K}}, \mathcal{I}$ and a compatible pair $(\mu, \hat{\mu})$, and implement the connection between domain and computation specification. In detail, rules (**pre-lift**) and (**post-lift**) enable to lift the state

$$\textbf{(pre-lift)}\ \frac{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1,\hat{\mu}(\Phi_1)\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}} \qquad \textbf{(post-lift)}\ \frac{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2,\hat{\mu}(\Phi_2)\\ \Phi_2\end{smallmatrix}\right\}}{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}$$

$$\textbf{(pre-core)}\ \frac{\Delta_1 \models^{\mathbf{K}} \alpha_{\mathbf{K}}(\Delta_1) \quad \mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1,\alpha_{\mathbf{K}}(\Delta_1)\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}} \qquad \textbf{(post-core)}\ \frac{\Delta_2 \models^{\mathbf{K}} \alpha_{\mathbf{K}}(\Delta_2) \quad \mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2,\alpha_{\mathbf{K}}(\Delta_2)\\ \Phi_2\end{smallmatrix}\right\}}{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}$$

$$\textbf{(post-inv)}\ \frac{\mathsf{sig}(\Delta_2) \subseteq \ker \hat{\mu} \quad \mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta,\Delta_2\\ \Phi_2\wedge\hat{\mu}^{-1}(\Delta_2)\end{smallmatrix}\right\}}{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta,\Delta_2\\ \Phi_2\end{smallmatrix}\right\}} \qquad \textbf{(pre-inv)}\ \frac{\mathsf{sig}(\Delta_1) \subseteq \ker \hat{\mu} \quad \mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta,\Delta_1\\ \Phi_1\wedge\hat{\mu}^{-1}(\Delta_1)\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta,\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}$$

$$\textbf{(cons)}\ \frac{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1'\\ \Phi_1'\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2'\\ \Phi_2'\end{smallmatrix}\right\} \quad \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} \to_{\mathbf{K}} \left\{\begin{smallmatrix}\Delta_1'\\ \Phi_1'\end{smallmatrix}\right\} \quad \left\{\begin{smallmatrix}\Delta_2'\\ \Phi_2'\end{smallmatrix}\right\} \to_{\mathbf{K}} \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}{\mathbf{C},\mathbf{K}\vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}$$

Fig. 5: Rules for manipulating pre- and post-conditions.

specification. Rules (**pre-core**) and (**post-core**) abduct a core in the domain specification. We remind here that we define α so that the signature of its range indeed is the kernel. Rules (**pre-inv**) and (**post-inv**) apply the inverse lifting on the core. The consequence rule enables to strengthen, respectively weaken, the specification. We stress here that (**pre-core**) and (**post-core**) (and (**var**), see below) invoke a DL reasoner – keeping Φ and Δ separate enables us to do so, an approach which merely translates DL into first-order logic would require to pass the verification condition to a solver for less tractable logics.

The rules for statements are given in Fig. 6. Using the previously introduced rules we can easily derive more complex ones that operate on both levels.

Rule (**var**) is the assignment rule for variables. On the state level, it is exactly the rule from the original Hoare calculus, expressing the precondition as the syntactically updated post-condition. On the domain level, it expresses that any domain knowledge in the domain post-condition must be justified by the state post-condition. As the domain pre-condition, however, it erases all information as the assignment may have arbitrary effects on the domain. Note that we can erase the domain knowledge in practice – strong consistency does not imply equivalence. In detail, (**skip**) expresses that the `skip` statement has no effect on the state. Branching, handled by rule (**branch**), also erases the domain precondition, as it modifies the state precondition. Rule (**inv**) handles loops by unrolling. Lastly, rule (**contract**) just checks that the contract is adhered to, and (**seq**) is as expected completely analogous to the original rule. It is worth noting that rule (**contract**) uses domain specification only to syntactically match it with the Pre and Post predicates. Our main result is the soundness of the rules for lifted Hoare triples.

Theorem 1 (Soundness). *The rules in Fig. 5 and Fig. 6 are sound.*

Given the above rules, we can easily combine several operations to derive sound rules that operate in the domain as well. One simple way is to merely lift before and after the statement, such as in the following derived rule.

$$\textbf{(var)}\ \frac{\hat{\mu}(\Phi) \models^{\mathbf{K}} \Delta}{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\emptyset}_{\Phi[\mathtt{v}\setminus\mathtt{expr}]}\right\}\mathtt{v} := \mathtt{expr}\left\{{}^{\Delta}_{\Phi}\right\}} \qquad \textbf{(skip)}\ \frac{}{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\Delta}_{\Phi}\right\}\mathsf{skip}\left\{{}^{\Delta}_{\Phi}\right\}}$$

$$\textbf{(branch)}\ \frac{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\emptyset}_{\Phi\wedge\mathtt{expr}}\right\}s_1\left\{{}^{\Delta}_{\Phi}\right\} \qquad \mathbf{C},\mathbf{K} \vdash \left\{{}^{\emptyset}_{\Phi\wedge\neg\mathtt{expr}}\right\}s_2\left\{{}^{\Delta}_{\Phi}\right\}}{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\emptyset}_{\Phi}\right\}\mathbf{if}\ (\mathtt{expr})\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}\left\{{}^{\Delta}_{\Phi}\right\}}$$

$$\textbf{(loop)}\ \frac{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\Delta_1}_{\Phi_1}\right\}\mathbf{if}\ (\mathtt{expr})\ \mathbf{do}\ s;\ \mathbf{while}\ (\mathtt{expr})\ \mathbf{do}\ s\ \mathbf{od}\ \mathbf{else\ skip\ fi}\left\{{}^{\Delta_2}_{\Phi_2}\right\}}{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\Delta_1}_{\Phi_1}\right\}\mathbf{while}\ (\mathtt{expr})\ \mathbf{do}\ s\ \mathbf{od}\ \left\{{}^{\Delta_2}_{\Phi_2}\right\}}$$

$$\textbf{(contract)}\ \frac{}{\mathbf{C},\mathbf{K} \vdash \mathsf{Pre}(\mathbf{C},\mathtt{p},\mathtt{expr})\ \mathtt{p}(\mathtt{e})\ \mathsf{Post}(\mathbf{C},\mathtt{p},\mathtt{expr})}$$

$$\textbf{(seq)}\ \frac{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\Delta_1}_{\Phi_1}\right\}s_1\left\{{}^{\Delta_3}_{\Phi_3}\right\} \qquad \mathbf{C},\mathbf{K} \vdash \left\{{}^{\Delta_3}_{\Phi_3}\right\}s_2\left\{{}^{\Delta_2}_{\Phi_2}\right\}}{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\Delta_1}_{\Phi_1}\right\}s_1;\ s_2\left\{{}^{\Delta_2}_{\Phi_2}\right\}}$$

Fig. 6: Rules for weakest precondition reasoning with lifted assertions.

$$\textbf{(lift-var)}\ \frac{}{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\hat{\mu}(\Phi[\mathtt{v}\setminus\mathtt{expr}])}_{\Phi[\mathtt{v}\setminus\mathtt{expr}]}\right\}\mathtt{v} := \mathtt{expr}\left\{{}^{\hat{\mu}(\Phi)}_{\Phi}\right\}}$$

While **(lift-var)** is sound, it does not transfer any information from the domain postcondition; the domain precondition is computed by a function of only the state precondition – it is, thus, not computing the weakest domain-precondition.

Using the mechanisms of the lifted core, we can give more precise versions of the rules for statements. Let $\mathsf{DPre}(\Delta,\Phi)$ be the domain knowledge constructed by abducting a lifted core from the domain postcondition ($\alpha(\Delta)$), delifting it into the state logic (via $\hat{\mu}^{-1}$), performing the substitution on the delifted core and the computation specification ($[\mathtt{v} \setminus e]$), and lifting the result back into the state logic ($\hat{\mu}$), thus realizing one full cycle of the information flow in Fig. 1: $\mathsf{DPre}(\Delta,\Phi) = \hat{\mu}\Big(\big(\Phi\wedge\hat{\mu}^{-1}(\alpha_{\mathbf{K}}(\Delta))\big)[\mathtt{v}\setminus\mathtt{expr}]\Big)$

However, for soundness it remains to show that the generated core is indeed implied by the domain specification. As discussed, this may not be the case if we use abduction for core generation. In this case, the open proof branches witness the abducted core and can be examined by the user. The following rule, for example, does so by integrating all steps to derive the domain precondition too.

$$\textbf{(total)}\ \frac{\Delta \models^{\mathbf{K}} \alpha_{\mathbf{K}}(\Delta) \qquad \hat{\mu}(\Phi) \models^{\mathbf{K}} \Delta}{\mathbf{C},\mathbf{K} \vdash \left\{{}^{\mathsf{DPre}(\Delta,\Phi)}_{\Phi\wedge\hat{\mu}^{-1}(\alpha_{\mathbf{K}}(\Delta))}\right\}\mathtt{v} := \mathtt{expr}\left\{{}^{\Delta}_{\Phi}\right\}}$$

Proposition 1. *Rule (**lift-var**) is sound. Rule (**total**) is sound.*

Completeness. Our system contains standard Hoare logic (when not using domain specification), which is complete up to the theory of terms [6]. Thus, for every domain logic strictly weaker than first-order logic, our system has the same property.

$$\dfrac{\dfrac{\dfrac{}{\texttt{hasValue}(\texttt{wheelsVar},4) \models^{\mathbf{K}} \texttt{HasFourWheels}(c), \texttt{hasValue}(\texttt{wheelsVar},4)}}{\mathbf{C},\mathbf{K} \vdash \{\substack{-\\ \texttt{nrWheels}\doteq 4}\}\texttt{wheels} := \texttt{nrWheels}\{\substack{\texttt{HasFourWheels}(c),\texttt{hasValue}(\texttt{wheelsVar},4)\\ \texttt{wheels}\doteq 4}\}}}{\mathbf{C},\mathbf{K} \vdash \{\substack{-\\ \texttt{nrWheels}\doteq 4}\}\texttt{wheels} := \texttt{nrWheels}\{\substack{\texttt{HasFourWheels}(c),\texttt{hasValue}(\texttt{wheelsVar},4)\\ -}\}}}{\mathbf{C},\mathbf{K} \vdash \{\substack{-\\ \texttt{nrWheels}\doteq 4}\}\texttt{wheels} := \texttt{nrWheels}\{\substack{\texttt{HasFourWheels}(c)\\ -}\}}$$

$$\begin{aligned}\mathbf{K} = \big\{&\texttt{SmallCar} \equiv \texttt{Has2Doors} \sqcap \texttt{Has4Wheels}, \quad \texttt{HasBody} \sqsubseteq \texttt{HasChassis} \sqsubseteq \texttt{Car}\\ &\exists \texttt{doors}.\exists \texttt{hasValue}.2 \equiv \texttt{Has2Doors}, \quad \exists \texttt{wheels}.\exists \texttt{hasValue}.4 \equiv \texttt{Has4Wheels}\\ &\exists \texttt{body}.\texttt{NonZero} \equiv \texttt{HasBody}, \neg\exists \texttt{hasValue}.0 \equiv \texttt{NonZero}, \quad \texttt{doors}(c, \texttt{doorsField})\\ &\texttt{HasChassis}(c), \texttt{wheels}(c, \texttt{wheelsVar}), \texttt{body}(c, \texttt{bodyVar})\big\}\end{aligned}$$

$$\texttt{Pre}_{\mathbf{C}}(\texttt{addWheels}, \texttt{nrWheels}) = \{-;\ \texttt{nrWheel} \doteq 4\}$$
$$\texttt{Post}_{\mathbf{C}}(\texttt{addWheels}, \texttt{nrWheels}) = \{\texttt{Has4Wheels}(c);\ -\}$$
$$\texttt{Pre}_{\mathbf{C}}(\texttt{assembly}) = \{-;\ \texttt{nrDoors} \doteq 4 \wedge \texttt{id} \neq 0\} \quad \texttt{Post}_{\mathbf{C}}(\texttt{assembly}) = \{\texttt{SmallCar}(c);\ -\}$$

Fig. 7: Proof of the running example

Example. We return to the assembly line, where we can now finally give a formal proof of our running example in Fig. 7. The domain knowledge $\mathbf{K}$ are the axioms from Sec. 3, given in Fig. 7 together with the contracts for all procedures. The proof of `addWheels`'s contract is below. The lifting is $\hat{\mu}(\texttt{wheel} \doteq n) = \{\texttt{HasValue}(\texttt{wheelsVar}, n)\}$.

Rule (**post-core**) is applied first and adds $\texttt{hasValue}(\texttt{wheelsVar}, 4)$ in the kernel generation through deduction – this follows from the *equivalence* axiom for `HasFourWheels`, as well as $\texttt{wheels}(c, \texttt{wheelsVar})$. The second applied rule is (**post-inv**), where this axiom is used to recover $\texttt{wheel} \doteq 4$. The third applied rule is (**var**), where we must show that the post-condition is strongly consistent. The proof for the contract of `assembly` is given in the technical. That proof uses a stronger contract for `addWheels` including framing, i.e., expresses which variables do not change, but bears no insights into the interplay of program and domain.

6 Related Work

While specification is a long-standing challenge for deductive verification [2,29,13], integration of description logics, or related technologies, such as the semantic web stack, into deductive verification of mainstream programming languages has not been explored. However, there are investigations on the direct integration of description logics into programming languages and subsequent model checking.

(Con)Golog [26,10] is an action programming language based on the situation calculus, designed to program agents that must access the current situation of their dynamic context. Golog has been connected with description logics to achieve decidable verification [33,32] of temporal logic properties, based on abstraction into a system where model checking is decidable [1]. To do so, the external world is modeled as a description logic model. In contrast, we target mainstream imperative programming

languages, where description logics are used only for specification. As we aim for better specification, questions of decidability are of lesser interest here. Knowledge and action bases [14] allow programs to access and manipulate a knowledge base using two abstraction operators, `ask` and `tell`. Again, verification of temporal properties based on model checking has been considered [14] with a focus on decidability [5,4], but no deductive system is given. Similarly, knowledge-based programs [9] use epistemic operators and have only been considered for analysis of simple temporal properties [20].

The original work on semantically lifted programs [18] uses integrated queries to access the lifted state. A similar mechanism is used by the probabilistic, ontologized programs [8], which also give a model checker for temporal properties based on SPIN. For semantically lifted programs, a type system is given in [19], which uses description logic entailments to verify graph query containments that ensure safety of the language-integrated queries. Leinberger et al. [25] also give a type system, but base their system not on liftings and graph queries, but on a tight integration of the class systems and graph shapes [23], which are again reduced to description logic entailments [24].

Refinement, e.g., using abstract state machines [3], is an alternative, formal approach to correct software based on refining an abstract specification into correct code. Many of our results, such as the interpretation of proof steps and domain specification, can conceptually carry over, but we stress that our tiers are *different* views on the same state – the domain view is not more abstract, but a genuinely alternative interpretation.

7 Conclusion

This work provides a new approach to tackle the long-standing specification and explanation bottlenecks in verification using techniques developed for knowledge representation. By using description logics as a second tier for software specifications, we can retain their pragmatics without sacrificing the expressive power needed to specify computational functionality.

The conditions on the lifting and its integration into the calculus are our main result, and the used programming language is consequently kept minimal. Thus, questions of expressive power and complexity are left for future work. For an implementation of our approach, we require to extend a program logic of a realistic programming language, and provide a lifting for its specification language. We consider Java as the most promising candidate: semantic lifing has been investigated for the JVM [15] and a rich ecosystem of verification tools, program logics and specificatin languages is available.

A full implementation for Java, however, is beyond the scope of this work and would merely obfuscate the generalicity of the core concept introduced here: Managing several perspectives on a correctness proofs by integrating a domain logic as a special view.

Future Work. Beyond such investigations of complexity and implementation, it remains an open question whether it is possible to retain information in the domain specification without explicitly generating the kernel.

Acknowledgments This work was partially supported by the SM4RTENANCE EU project (grant nr. 101123490).

References

1. Baader, F., Zarrieß, B.: Verification of Golog programs over description logic actions. In: FroCos. Lecture Notes in Computer Science, vol. 8152, pp. 181–196. Springer (2013)
2. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification – specification is the new bottleneck. In: SSV. EPTCS, vol. 102, pp. 18–32 (2012)
3. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003), http://www.springer.com/computer/swe/book/978-3-540-00702-9
4. Calvanese, D., Ceylan, İ.İ., Montali, M., Santoso, A.: Verification of context-sensitive knowledge and action bases. In: JELIA. Lecture Notes in Computer Science, vol. 8761, pp. 514–528. Springer (2014)
5. Calvanese, D., Gianola, A., Mazzullo, A., Montali, M.: SMT safety verification of ontology-based processes. In: AAAI. pp. 6271–6279. AAAI Press (2023)
6. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. **7**(1), 70–90 (1978). https://doi.org/10.1137/0207005, https://doi.org/10.1137/0207005
7. Deng, X., Haarslev, V., Shiri, N.: A framework for explaining reasoning in description logics. In: ExaCt. AAAI Technical Report, vol. FS-05-04, pp. 55–61. AAAI Press (2005)
8. Dubslaff, C., Koopmann, P., Turhan, A.: Enhancing probabilistic model checking with ontologies. Formal Aspects Comput. **33**(6), 885–921 (2021)
9. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press (1995)
10. Giacomo, G.D., Lespérance, Y., Levesque, H.J.: ConGolog, a concurrent programming language based on the situation calculus. Artif. Intell. **121**(1-2), 109–169 (2000)
11. Grebing, S., Ulbrich, M.: Usability recommendations for user guidance in deductive program verification. In: 20 Years of KeY, Lecture Notes in Computer Science, vol. 12345, pp. 261–284. Springer (2020)
12. Gurov, D., Westman, J.: A Hoare Logic Contract Theory: An Exercise in Denotational Semantics, pp. 119–127. Springer International Publishing, Cham (2018)
13. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science, Lecture Notes in Computer Science, vol. 10000, pp. 345–373. Springer (2019)
14. Hariri, B.B., Calvanese, D., Montali, M., Giacomo, G.D., Masellis, R.D., Felli, P.: Description logic knowledge and action bases. J. Artif. Intell. Res. **46**, 651–686 (2013)
15. Haubner, A.W.: Inspecting Java Program States with Semantic Web Technologies. Master's thesis, Technische Universität Darmstadt, Darmstadt (2022). https://doi.org/10.26083/tuprints-00022143, the software developed as part of this thesis is available on GitHub. The Semantic Java Debugger: https://github.com/ahbnr/SemanticJavaDebugger The jdi2owl library: https://github.com/ahbnr/jdi2owl
16. Horrocks, I., Sattler, U.: Ontology reasoning in the SHOQ(D) description logic. In: IJCAI. pp. 199–204. Morgan Kaufmann (2001)
17. Kamburjan, E., Gurov, D.: A Hoare logic for domain specification (full version) (2024)
18. Kamburjan, E., Klungre, V.N., Schlatte, R., Johnsen, E.B., Giese, M.: Programming and debugging with semantically lifted states. In: ESWC. Lecture Notes in Computer Science, vol. 12731, pp. 126–142. Springer (2021)
19. Kamburjan, E., Kostylev, E.V.: Type checking semantically lifted programs via query containment under entailment regimes. In: Description Logics. CEUR Workshop Proceedings, vol. 2954. CEUR-WS.org (2021)

20. Knapp, A., Mühlberger, H., Reus, B.: Interpreting knowledge-based programs. In: ESOP. Lecture Notes in Computer Science, vol. 13990, pp. 253–280. Springer (2023)
21. Koopmann, P., Del-Pinto, W., Tourret, S., Schmidt, R.A.: Signature-based abduction for expressive description logics. In: KR. pp. 592–602 (2020)
22. Krisnadhi, A., Hitzler, P.: The stub metapattern. In: WOP@ISWC. Studies on the Semantic Web, vol. 32, pp. 39–45. IOS Press (2016)
23. Leinberger, M., Lämmel, R., Staab, S.: The essence of functional programming on semantic data. In: ESOP. Lecture Notes in Computer Science, vol. 10201, pp. 750–776. Springer (2017)
24. Leinberger, M., Seifer, P., Rienstra, T., Lämmel, R., Staab, S.: Deciding SHACL shape containment through description logics reasoning. In: ISWC (1). Lecture Notes in Computer Science, vol. 12506, pp. 366–383. Springer (2020)
25. Leinberger, M., Seifer, P., Schon, C., Lämmel, R., Staab, S.: Type checking program code using SHACL. In: ISWC (1). Lecture Notes in Computer Science, vol. 11778, pp. 399–417. Springer (2019)
26. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: A logic programming language for dynamic domains. J. Log. Program. **31**(1-3), 59–83 (1997)
27. Mayer, M.C., Pirri, F.: First order abduction via tableau and sequent calculi. Log. J. IGPL **1**(1), 99–117 (1993)
28. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Springer-Verlag, Berlin, Heidelberg (2007)
29. Rozier, K.Y.: Specification: The biggest bottleneck in formal methods and autonomy. In: VSTTE. Lecture Notes in Computer Science, vol. 9971, pp. 8–26 (2016)
30. Schlobach, S.: Explaining subsumption by optimal interpolation. In: JELIA. Lecture Notes in Computer Science, vol. 3229, pp. 413–425. Springer (2004)
31. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge, MA, USA (1993)
32. Zarrieß, B.: Verification of golog programs over description logic actions. Ph.D. thesis, Dresden University of Technology, Germany (2018)
33. Zarrieß, B., Claßen, J.: Verification of knowledge-based programs over description logic actions. In: IJCAI. AAAI Press (2015)