A Hoare Logic for Domain Specification (Extended Abstract)

Eduard Kamburjan[1], Dilian Gurov[2]

[1]University of Oslo, Oslo, Norway
[2]KTH Royal Institute of Technology, Stockholm, Sweden

Abstract
Programs must be correct with respect to their application domain. Yet, the program specification and verification approaches so far only consider correctness in terms of computations. In this work, we present a two-tier Hoare Logic that integrates assertions for both implementation and domain. For domain specification, we use description logics and semantic lifting, a recently proposed 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. This work is an extended abstract of [1].

Keywords
Hoare Logic, Program Verification, Domain Specification

Motivation Programs must respect constraints coming from their application domain, and thus, their correctness hast to rely on an encoding of domain knowledge. At the very minimum, application logic must correspond to business logic, but in extreme cases, such as simulators or applications in model-based engineering, the domain is directly encoded in the program. Description logics (DL) are an established tool to model domain knowledge with elaborate pragmatics. While specification is a long-standing challenge for deductive verification [2, 3, 4], integration of description logics, or related technologies, such as the semantic web stack, into deductive verification of mainstream programming languages has not been explored.

In this work, we investigate reasoning about the correctness of programs with specification for both the *implementation* (i.e., the program specifics) and its connection to the application *domain*, while managing these two perspectives throughout the proof. 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 knowledge representation techniques and 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, for example, explained [5, 6] 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.

DL 2024: 37th International Workshop on Description Logics, June 18–21, 2024, Bergen, Norway
eduard@ifi.uio.no (E. Kamburjan); dilian@kth.se (D. Gurov)
0000-0002-0996-2543 (E. Kamburjan); 0000-0002-0074-8786 (D. Gurov)

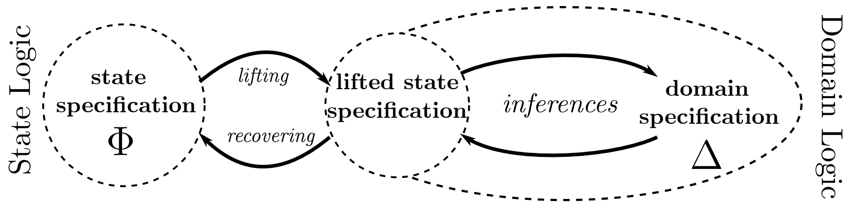

Figure 1: Relation between domain and state specifications in their respective logics.

Two-Tier Assertions. To connect program state and description logics, we use ideas from *semantically lifted programs* [7]. The state of a semantically lifted program is *lifted* into the domain in the form of a DL model, which 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 $\left\{ {\Delta \atop \Phi} \right\}$ 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. 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. We return to this further below, but first illustrate the use of two-tier assertions in a Hoare logic [8], which reasons about program executions.

Example. 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. $\texttt{SmallCar} \equiv \texttt{HasTwoDoors} \sqcap \texttt{HasFourWheels} \sqcap \texttt{Car}$.

Additionally, we know that 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 formulas to express that everything that has 2 doors is part of the concept `HasTwoDoors`, and analogously for `HasFourDoors` and `HasBody`. We use the common pattern of *stubs* [9]: 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`. We can use variables in the programming language to connect the two formalisms, but using them as (domain) stubs.

$$\texttt{HasBody} \sqsubseteq \texttt{HasChassis} \sqsubseteq \texttt{Car} \qquad \exists\texttt{doors}.\exists\texttt{hasValue}.2 \equiv \texttt{HasTwoDoors}$$
$$\exists\texttt{wheels}.\exists\texttt{hasValue}.4 \equiv \texttt{HasFourWheels} \qquad \exists\texttt{body}.\texttt{NonZero} \equiv \texttt{HasBody}$$
$$\neg\exists\texttt{hasValue}.0 \equiv \texttt{NonZero}$$

The program is given below. 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 $\texttt{bodyId} = 0$ 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 considered 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.

```
var bodyId = 0; var wheels = 0; var doors = 0;
proc addWheels(nrWheels) begin  wheels := nrWheels; end;
proc assembly(id, nrDoors) begin
  bodyId := id; addWheels(4); doors := nrDoors;
end
```

Let us examine the specification of addWheels in more detail. Intuitively, a Hoare triple $\left\{\begin{smallmatrix}\phi_1\\\psi_1\end{smallmatrix}\right\}\mathtt{s}\left\{\begin{smallmatrix}\phi_2\\\psi_2\end{smallmatrix}\right\}$ expresses that if in the prestate of a program the specification $\left\{\begin{smallmatrix}\phi_1\\\psi_1\end{smallmatrix}\right\}$, and the program terminated, then in the post-state $\left\{\begin{smallmatrix}\phi_2\\\psi_2\end{smallmatrix}\right\}$ holds. The domain specification explains what is expected from the view of the car assembly (the car already has a chassis), while the implementation specification ($\mathtt{nrWheels} = 4$) 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.

$$\left\{\begin{matrix} - \\ \mathtt{nrWheels} \doteq 4\end{matrix}\right\}\mathtt{addWheels}(\mathtt{nrWheels})\left\{\begin{matrix}\mathtt{HasFourWheels}(c)\\ -\end{matrix}\right\} \tag{1}$$

The post-condition is obvious - it states that afterwards the car being assembled is part of class HasFourWheels. Its domain precondition states that the car has a chassis before. 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 domain expert a more fine-grained specification, without being exposed to 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.

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 expressed as

$$\left\{\begin{matrix} - \\ \mathtt{nrWheels} \doteq 4\end{matrix}\right\}\mathtt{wheels} := \mathtt{nrWheels}\left\{\begin{matrix}\mathtt{HasFourWheels}(c)\\ -\end{matrix}\right\}$$

In a classical Hoare calculus, based on weakest-preconditions [8], we would now substitute wheels by nrWheels in the post-condition – the post-condition obviously needs to be $\mathtt{wheels} \doteq 4$. 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:

$$\texttt{hasValue}(\texttt{wheelsVar}, 4) \qquad \texttt{HasChassis}(c) \qquad \texttt{wheels}(c, \texttt{wheelsVar})$$
$$\texttt{body}(c, \texttt{bodyVar}) \qquad \texttt{doors}(c, \texttt{doorsVar})$$

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 [7]. In our example, the first axiom is (part of) the lifted state, the last four axioms connect lifted program state and domain knowledge.

Still, we can deduce knowledge about the lifted state: If the car has four wheels ($\texttt{HasFourWheels}(c)$), then the corresponding variable must be set to four ($\texttt{hasValue}(\texttt{wheels}, 4)$). This information, in turn, can be interpreted in the program logic as $\texttt{wheels} \doteq 4$, in order to strengthen our specification into Eq. 1.

Using the rule for assignment, we can prove the correctness of addWheels w.r.t. to its specification. We must consider the relation of $\texttt{HasFourWheels}(c)$ 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 $\texttt{HasFourWheels}(c)$.

The following describes how the above is expressed in a proof in the calculus given in [1], where full semantics and proof system are given. The judgement $\mathbf{K} \vdash \left\{\begin{smallmatrix}\phi_1\\ \psi_1\end{smallmatrix}\right\}\texttt{s}\left\{\begin{smallmatrix}\phi_2\\ \psi_2\end{smallmatrix}\right\}$ expresses that the given Hoare triple is valid, if for all reasoning steps the set of axioms $\mathbf{K}$, which models domain knowledge, is added to the lifted state and specifications.

The first rule application is the above deduction that if a car has four wheels then the wheels variable must have the value four in the lifting . The second rule application then recovers this information in the state logic. The process of recovery can be deductive and abductive, and we refer to [1] for details. Finally, the standard (backwards) rule for assignment in Hoare logic can be used and performs the usual substitution. It remains only to show that the lifted post-state specification is consistent, as described above.

$$\begin{array}{c}
\hline
\texttt{hasValue}(\texttt{wheelsVar}, 4) \models^{\mathbf{K}} \texttt{HasFourWheels}(c), \texttt{hasValue}(\texttt{wheelsVar}, 4) \\
\hline
\mathbf{K} \vdash \left\{\begin{smallmatrix} - \\ \texttt{nrWheels}\doteq 4\end{smallmatrix}\right\}\texttt{wheels} := \texttt{nrWheels}\left\{\begin{smallmatrix}\texttt{HasFourWheels}(c),\texttt{hasValue}(\texttt{wheelsVar},4)\\ \texttt{wheels}\doteq 4\end{smallmatrix}\right\} \\
\hline
\mathbf{K} \vdash \left\{\begin{smallmatrix} - \\ \texttt{nrWheels}\doteq 4\end{smallmatrix}\right\}\texttt{wheels} := \texttt{nrWheels}\left\{\begin{smallmatrix}\texttt{HasFourWheels}(c),\texttt{hasValue}(\texttt{wheelsVar},4)\\ -\end{smallmatrix}\right\} \\
\hline
\mathbf{K} \vdash \left\{\begin{smallmatrix} - \\ \texttt{nrWheels}\doteq 4\end{smallmatrix}\right\}\texttt{wheels} := \texttt{nrWheels}\left\{\begin{smallmatrix}\texttt{HasFourWheels}(c)\\ -\end{smallmatrix}\right\}
\end{array}$$

Conclusion. Specification, especially when connected to external documents, not just computations, is a long standing problem in deductive program verification. This work presents a two-tier Hoare logic that integrates description logic specification over a semantically lifted program. At its heart, it introduces semantical lifting of specifications to leverage the pragmatics DL to program specification and program logic caluculi.

References

[1] E. Kamburjan, D. Gurov, A hoare logic for domain specification (full version), 2024. arXiv:2402.00452.

[2] C. Baumann, B. Beckert, H. Blasum, T. Bormer, Lessons learned from microkernel verification – specification is the new bottleneck, in: SSV, volume 102 of *EPTCS*, 2012, pp. 18–32.

[3] K. Y. Rozier, Specification: The biggest bottleneck in formal methods and autonomy, in: VSTTE, volume 9971 of *Lecture Notes in Computer Science*, 2016, pp. 8–26.

[4] R. Hähnle, M. Huisman, Deductive software verification: From pen-and-paper proofs to industrial tools, in: Computing and Software Science, volume 10000 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 345–373.

[5] X. Deng, V. Haarslev, N. Shiri, A framework for explaining reasoning in description logics, in: ExaCt, volume FS-05-04 of *AAAI Technical Report*, AAAI Press, 2005, pp. 55–61.

[6] S. Schlobach, Explaining subsumption by optimal interpolation, in: JELIA, volume 3229 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 413–425.

[7] E. Kamburjan, V. N. Klungre, R. Schlatte, E. B. Johnsen, M. Giese, Programming and debugging with semantically lifted states, in: ESWC, volume 12731 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 126–142.

[8] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (1969) 576–580.

[9] A. Krisnadhi, P. Hitzler, The stub metapattern, in: WOP@ISWC, volume 32 of *Studies on the Semantic Web*, IOS Press, 2016, pp. 39–45.