POSITION PAPER:
Should We Begin
a Standardization Process
for Interface Logics?

Matt Kaufmann
J Strother Moore

Technical Report 72 January, 1992

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

1. Introduction

This paper is a companion paper to [1]. In that paper we argue against so-called *interface logics* for the verification of *systems*, and present an alternative approach that has already enjoyed considerable success. In this document we raise concerns about the use of interface logics even for the verification of software or hardware whose scale is smaller than what one might normally call ‘‘systems.’’

We believe that it important for us to raise these concerns. There has recently been some considerable discussion of ‘‘interface logics’’ in the verification community [2, 3]. Our impression is that there could be pressure forthcoming at some point to standardize one or more interface logics. We have chosen to air our concerns here at this time in order to head off such pressure.

Our basic position is that any standardization process is expensive, and given the limited manpower available for theorem proving and system verification, it is premature to consider standardization of interface logics. Moreover, we believe that it is likely to remain premature for awhile, if not indefinitely, for reasons we explain in this paper. More generally speaking, our position is that real progress in mechanically-assisted systems verification comes, and will continue to come, from the development and application of particular tools and logics for those tools.

Let us begin by considering what might be meant by an ‘‘interface logic’’. First we present a ‘‘narrow interpretation’’ that seems to make sense, and we issue a challenge that we argue should be met before further resources are spent on the topic. In the rest of this paper we consider a number of alleged benefits that would accrue from the presence of interface logics (which tend not to fall into the ‘‘narrow interpretation’’). In each case we raise concerns that make us dubious that interface logics will have any of these advantages.

1.1 A narrow interpretation

One possibility is that an ‘‘interface logic’’ is simply a logic that is, in essence, already shared by several existing systems, though minor syntactic details may vary. This ‘‘interface logic’’ should simply be a sort of isomorphic copy of all of these logics that is convenient for mechanical manipulation. That is, it is simply providing a common convenient syntax for the logics underlying those systems.

In this case we believe that an interface logic could be quite useful for allowing those theorem provers that already support that logic to behave as if they were integrated into a single system. That is, the advantage of such a logic could be that it allows for a uniform input and output language for the given systems. Thus with some potentially routine additional work, the systems could be literally integrated into a single proof assistant. And whether or not such additional integration were performed, builders of formula generators that target those systems could instead target the interface logic.

However, in order to be convinced of the utility of this ‘‘narrow interpretation’’ (where all the logics are essentially the same as the interface logic) we suggest consideration of the following action item.

> **CHALLENGE:** Connect several existing high quality mechanized reasoning systems, by defining a logic onto which the logics of those systems map isomorphically.

Notice that *we* do not propose to do such a task; in fact we have seen no need for interface logics in our own work. Rather, we suggest that proponents of this ‘‘narrow interpretation’’ of interface logics

demonstrate the idea's viability by providing an *example* based on connecting a nucleus of existing systems. Guttman's paper [3] is a step in this direction, though it does not provide several existing systems that isomorphically map precisely to one of the logics provided there.

We do not think it particularly useful to have community-wide discussion of such a task. Rather, we suggest that proponents of this ''narrow interpretation'' of interface logics carry out such a task however they choose, and present their work to the rest of the verification community.

1.2 Introduction to our general concerns

However, our impression is that the discussion of interface logics (especially in [2]) has gone beyond the ''narrow interpretation'' we discuss above. In fact, except for that interpretation we have not understood to our own satisfaction just how interface logics would be usefully applied to nontrivial verification problems. In the system verification realm we actually argue quite directly against interface logics in [1]. However, even outside that realm we have a number of concerns about interface logics.

Let us also acknowledge that standardization can in general be made difficult by waiting too long, since various divergent interests could by then be entrenched. On the other hand, theorem proving is a relatively young field and our intuition is that such entrenchment is not likely to increase substantially during the next few years. In fact, we believe that it is desirable to allow the field to grow in whatever direction it might naturally ''choose,'' without the potential for standardization to cut off fruitful lines of research.

This note consists of a number of sections, where each section considers a single potential benefit of interface logics. We welcome feedback, in particular on the following two questions:

- Have we omitted any important potential benefits of interface logics?
- Have we missed any considerations for those benefits that we *do* consider?

However, before moving to those sections, we set up some notation and attempt to clarify a few relevant points.

Notation. We use the notation IL to refer to an arbitrary interface logic, and PL to refer to an arbitrary theorem prover's logic (Prover Logic).

It is important to clarify the possible role of an ''interface logic.'' Such a logic could be used to connect *into* theorem provers, to connect *from* theorem provers, or both. Let us address these notions in turn. We refer to them respectively as the **INPUT** and **OUTPUT** roles for IL, i.e. its uses for interfacing *to* or *from* (the logic of) a theorem prover.

INPUT

The IL could be tied to the *input* of a prover's logic PL by writing a simple translator. Then formulas written in the IL could in principle be proved by that prover. That is, here we would presumably like to use the theorem prover (for reasons we'll discuss later in this note) to answer questions formulated in IL, i.e. questions of the following form.

> Does the IL sentence ϕ follow semantically from the set Γ of IL sentences?

As Josh Guttman has pointed out (personal communication), we must require that in this case, if Γ is any set of sentences of IL and ϕ is a sentence of IL, and if Γ' and ϕ' are their respective translations into PL, then

INPUT criterion:

$\Gamma' \models_{PL} \phi'$
implies
$\Gamma \models_{IL} \phi$

where here $\models_{PL}$ is the semantic consequence relation of PL (i.e. every model of the left side should be a model of the right side), and $\models_{IL}$ is the semantic consequence relation of IL. This requirement guarantees that if one translates a question of the form ''Does the IL sentence ϕ follow semantically from the set of IL sentences Γ'' into PL, and then applies a sound prover to obtain an answer of ''yes'' for the translation of this question into PL, then one can conclude that this question has an answer of ''yes'' as well.

Note that with a prover such as Nqthm (the Boyer-Moore theorem prover), the logic is sufficiently rich that the **INPUT** criterion may be difficult to satisfy. Of course, one might imagine extending the **INPUT** criterion so as to allow a fixed theory T_0 on the left side of ''$\models_{IL}$'' as follows:

Modified input criterion:

$\Gamma' \models_{PL} \phi'$
implies
$\Gamma \cup T_0 \models_{IL} \phi$

In the case of Nqthm, this says roughly that by adding the translation of the built-in Nqthm axioms and some kind of translation of its shell principle, principle of definition, and induction rule of inference, we regain the input criterion. However, we are concerned that the expression of T_0 could be rather awkward.

OUTPUT

Another possible use for an IL is for interfacing *from* a theorem prover. That is, suppose we have a translator from PL to IL. Again we have a requirement: this one ensures that when the prover has proved a theorem ψ of PL from a set of axioms Δ, then we can say something about IL, namely that the translation ψ^* of ψ into IL is a semantic consequence of the translation Δ^* of Δ into IL. That is, we require that

OUTPUT criterion:

$\Delta \models_{PL} \psi$
implies
$\Delta^* \models_{IL} \psi^*$

Let us summarize some potential benefits of interface logics that we will consider in the sections that follow. We have picked these up from various sources, in particular from discussion in the ''deftpi'' discussion via electronic mail [2].

- Allow different provers to be used on the same problem:
  - Different theorem proving systems could be compared;
  - Different tools could be brought to bear where they are most suited;
  - Builders of formula generators could defer their decision on which theorem prover(s) to target.
  - By using different tools on the same problem, added assurance could be gained that the purported theorems are indeed theorems.
- Allow incremental improvements in systems by allowing replacement of modules.

- Allow for clear presentation of what's been proved, in a uniform language, so that others can understand it.
- Assist with the definition of the semantics of new prover logics.

NOTE!!! We are not arguing against interface logics for those who wish to use them. We welcome proponents of the interface logic idea to create interface logics, use them successfully in applications, and to disseminate their results. We are simply explaining why we do not yet see the overall benefit of *standardizing* interface logics.

Acknowledgement. We thank Joshua Guttman of the Mitre Corporation for a number of interesting exchanges on the subject of this paper. We also thank Don Good for his comments on a draft of this paper.

2. Allow different provers to be used on the same problem (?)

Apparently some of the discussion about interface logics has suggested that they would allow different theorem provers to be used on the same problem. That is, we imagine that a problem would be formulated in an interface logic, and then translated into various prover logics. Thus, we expect that the interface logic should satisfy the ''**INPUT** criterion'' described in the preceding section. Some of the specific benefits that could presumably accrue are as follows.

- Different theorem proving systems could be compared.
- Different tools could be brought to bear where they are most suited.
- Builders of formula generators could defer their decision on which theorem prover(s) to target.
- By using different tools on the same problem, added assurance could be attained that the purported theorems are indeed theorems.

Let us address each of these items in turn.

2.1 Different theorem proving systems could be compared (?)

Certainly it is interesting and potentially valuable to be able to evaluate relative effectiveness of different theorem proving systems, especially if one can assess which classes of problems are most suitable to the respective provers. However, our experience is that comparison of different theorem proving systems is an expensive and difficult enterprise. Quoting from one such comparison [4]:

> Perhaps though it would be responsible of us to point out that it took much more effort to write this paper than to carry out the proofs.

Some potential pitfalls in comparing theorem provers are as follows. Most of these come essentially from [4].

- Counting characters or tokens from the final input file isn't necessarily a useful metric:
  - It fails to take into account the amount of work that went into fruitless paths that were eventually abandoned. (One could include such input into the total, but that seems excessively tied to the user's style; some users are very deliberate, others explore at the keyboard.)

  - It fails to take into account the extent to which the editor was used to generate text automatically or to provide useful guidance. For example, the Nqthm proof from [4] used two lemmas that were created with the emacs editor from existing text, using very few keystrokes; yet these consisted of 116 tokens.
  - It isn't always clear just what constitutes a ''token.''
- Replay time is a dangerous metric. For example, imagine a theorem prover that provides considerable heuristic support for proofs. Such a prover is likely to encourage the development of prover input that takes longer to replay than would corresponding input developed for a ''proof-checker.'' For, a less heuristic proof assistant might require more input but execute relatively quickly because it does not employ search strategies.
- It is not clear how to ''count'' tactics in a tactic-based prover.
- It is difficult to compare the naturality or the ease of use of different systems.
- It can be quite misleading to count the number of lemmas, since it is often just a matter of style whether ''proof steps'' are formulated as lemmas or as prover commands or hints.
- The degree of success in using a theorem prover depends largely on the user, and it is difficult to see how to control for that variable. Even the same user can have skills that are a better fit for one prover or another, or experience that serves her well for one prover but not for another.
- The presence of translators from intermediate languages to different provers does not address the issue of soundness of their implementations, or availability of the software, or the presence of good user manuals.
- The translation of IL to some particular PL will necessarily complicate the problems more for some PL's than others. Embeddings of one logic in another rarely preserve proof complexity.

We wonder if having an IL would really facilitate comparisons anyhow. Perhaps they would just put an added burden on those who wish to formalize problems in styles that are most natural for the provers in question.

Having said all this, we cannot resist bringing up what we feel is an obvious question: given the limited resources of the community, do we really expect that there will be much use of different tools on the same problem? If so, wouldn't we be content with their use on *different formalizations* of the same problem?

2.2 Different tools could be brought to bear where they are most suited (?)

Do we imagine people actually using more than one tool in practice? Our own experience with translation and formula generation is such that we have been happy to translate directly into the prover's internal form.

One scenario that we can imagine (as can others, apparently) is that one is building a formula generator in order to reason about programs in some imperative programming language, but has no particular theorem prover in mind at that time. Wouldn't such a person benefit from being able to target to an interface logic, in order to defer the decision of which prover to use, and perhaps avoid the need to build his own prover? We believe that the answer to this question is ''yes'', in principle. Such an approach also would be likely to allow different provers to be used, thus making the output of the formula generator available for use by

more than one theorem proving group.[1]

It's not clear to us that it is so efficient to have one's formula generator target an interface logic, rather than the logic of a particular theorem prover. Theorem provers today are rather idiosyncratic tools, which can be quite sensitive to the form of the ''verification conditions.'' Here we refer not only to minor syntactic issues, such as which boolean functions to take as primitives, but to more serious issues such as whether to use natural number or integer arithmetic, whether to define list indexing recursively or by direct axioms, whether or not to skolemize the quantifiers away, and so on.

For a concrete example, consider the Fortran VCG work described in [5]. There, the loop invariants were formulated as rewrite rules. Thus, knowledge of the target prover was important. The presence of an intervening interface logic would probably have complicated the task of formulating those loop invariants appropriately.

2.3 Builders of formula generators could defer their decision on which theorem prover(s) to target (?)

This idea makes a certain amount of sense, assuming that interface logics are constructed with the **INPUT** criterion in mind. In fact, we argue in [1] that the truly important constraint on writers of formula generators is that they choose a well-defined logic, not necessarily a theorem prover, as their target.

However, we think the benefit of using an interface logic may be fairly minimal. If one builds a formula generator that targets a particular theorem prover, we suspect that only minimal effort would be required to retarget to a different prover, provided both provers' logics embed the interface logic that would otherwise have been used.

The drawbacks discussed at the end of the preceding subsection apply here as well.

2.4 By using different tools on the same problem, added assurance could be attained that the purported theorems are indeed theorems (?)

It is already possible to use different tools on the same problem. For example, NASA has recently been supporting work using at least three different theorem provers on the same problems.

This isn't to say that all three systems use logically equivalent *formalizations* of the problems, however. An interface logic satisfying the **INPUT** criterion (see above) might help there, assuming that the interface logic were strong enough to meet that criterion while inexpressive enough to be embedded into all three prover logics. That's a rather strong assumption, however, since at least one of the logics uses higher-order constructs to express its specifications.

---

[1] Are VCGs really used much any more anyhow? Indeed, some of us at Computational Logic, Inc. proposed a GVE to Nqthm hookup to NRL, but it wasn't funded. Given the lack of sufficient interest to hook up probably the most widely used verification condition generator to perhaps the most successfully used prover for system verification, then why should we believe that there would be interested in a ''generic'' version of such a hookup? It's not at all clear to us, offhand, that an interface logic would make such a hooking-up task significantly simpler.

3. Allow incremental improvements in systems by allowing replacement of modules (?)

Such as, which modules? Our experience suggests that it isn't as helpful as one might think to have stand-alone theorem prover modules. For one thing, provers aren't even written in the same computer language. For example, Nqthm and RRL are written in Common Lisp, HOL and Nuprl is written in ML, Larch is written in CLU, OTTER is written in C, and Oyster/Clam are written in Prolog.

Perhaps more significantly, it can be difficult to write stand-alone modules even for a single theorem prover. The linear arithmetic procedure in Nqthm is a good example of this phenomenon [6]: it was discovered that the particular decision procedure was much less important than the way it was tightly integrated into the rewriting process. Some recent preliminary work by Kaufmann in incorporating binary decision diagram (BDD) technology [7] into Nqthm seemed to significantly *slow down* the prover, in spite of its apparent superiority for tautologies to the prover's clausification method. The problem appears to be that we need to consider more carefully the interaction of Nqthm's clausification process with the BDD algorithm we used. The point here is that it seems naive to expect that stand-alone modules are a big win. Perhaps someone can point us to examples of modules in theorem provers that have been ripped out? Actually we have a couple of examples like that, but in neither case do we believe that our code could be made easily portable for different provers.

Before accepting an argument that interface logics could assist in the development of modules to be shared by different provers, we would like to hear of significant examples that one might imagine. We suspect that it would be at least as easy to draw the interface boundary at the prover boundaries rather than at the level of modules. We have already discussed the feasibility of such a use of interface logics in Section 2.

Even if such examples exist, we wonder whether the presence of an interface logic would simplify the task of integrating the module. We consider it likely that implementors would prefer to understand the algorithm in question and re-implement it for their own internal data structures, rather than use an interface provided by an IL.

4. Allow for clear presentation of what's been proved, in a uniform language, so that others can understand it (?)

Here, we imagine either that one ultimately presents his results in an interface logic (IL), either by translating an IL specification into his prover's logic (PL) and then working with the prover, or by translating the PL specification into an IL. In the former case, the **INPUT** criterion (see Section 1) must be met; in the latter case, the **OUTPUT** criterion must be met.

From this point of view, it should be the case that there are few interface logics around and they are quite simple. Otherwise, it is not clear that it is helpful for understanding specifications to translating results to (or from) an interface logic.

Consider however all the different features that some existing logics have which are *not* shared. Do we imagine a small number of logics, i.e. significantly smaller than the number of prover logics, that could accommodate such a collection of features?

- No law of the excluded middle (e.g. Nuprl/Oyster, Calculus of Constructions provers)
- Simple type theory with polymorphism (HOL, at least)
- Dependent products (NuPrl/Oyster, Calculus of Constructions)
- Recursive data types, built-in and definable by users (Nqthm)
- Partial functions without ‘‘bottom’’ (Lutins/IMPS)
- Partial functions with bottom (Clio)
- Mathematical induction of different flavors (RRL)
- *NO* true equality (Calculus of Constructions)
- *NO* quantification (Nqthm)

Perhaps it’s pretty obvious that our personal concern, as researchers with an interest in Nqthm, is that the Nqthm logic will not fit into any interface logic that is adopted as a standard. However, the constructive logics of some of today’s provers are probably even further from any standards that one might propose. We are concerned that the adoption of standard interface logics might unduly discourage the use of state-of-the-art theorem provers.

The challenge we put forth above in Subsection 1.1 could be weakened so that it only requires that the **OUTPUT** criterion (cf. Subsection 1.2) be met:

> **modified CHALLENGE:** Connect several existing high quality mechanized reasoning systems, by defining a logic into which the logics of those systems embed such that the **OUTPUT** criterion is satisfied.

To the extent that this weakened challenge is met, the concern we express in the present section would be alleviated.

5. Assist with the definition of the semantics of new prover logics (?)

The idea here is that if suitable interface logics exist, then perhaps it will be the case that given a new prover logic, one of these ILs will be rich enough in syntax so that the prover logic can be defined by translating its syntax into the IL syntax and inheriting the IL’s notion of semantic entailment. One can then presumably check that the prover implements axioms and rules of inference that respect these semantics.

This is appealing in principle, since the idea is that by defining the semantics of the ILs once and for all, we can avoid the need to do this for each new logic simply by translating that logic into an IL. However, defining semantics for logics isn’t generally difficult anyhow. In those cases where it is difficult, we wonder if well-understood interface logics would be rich enough to ‘‘absorb’’ the prover logic. An exception could be the treatment of partial functions by Lutins (see [3]) that could make it a useful interface logic for other logics with partial functions. However, note that in the informal discussions in [2], Roger Jones has explained that he is content so far with HOL as a target for the use of HOL to support Z, including its partial functions.

If such an approach to semantics were ever widely adopted, we wonder if interesting new logics couldn’t be discouraged simply because they didn’t fit well into the standard(s). The Calculus of Constructions is

probably an example of a logic (now, a family of logics) that is likely not to fall into such a framework.

6. Conclusion

A solution to the concern raised in the paragraph immediately above, and perhaps to all the concerns raised in this paper, is simply to let people propose and publish interface logics (e.g. see [3]) and demonstrate that they map to existing logics by implementing the mappings. If there is a natural demand for interface logics and they are found to be useful, then standardization can be considered. We hope that by the time standardization is considered, our understanding of the issues involved have reached sufficient maturity to guide us in making good decisions.

The concerns we have tried to put forward in this paper are twofold. First, we are not convinced that a mature level of understanding of issues in interface logics exists at this time. Second, we have not yet seen sufficient evidence that increased understanding would in fact suggest the desirability of standardized interface logics.

References

1. Donald I. Good, Matt Kaufmann, J Strother Moore, ‘‘The Role of Automated Reasoning in Integrated System Verification Environments’’, Technical Report 73, Computational Logic, Inc., January 1992.

2. ‘‘<Informal electronic discussion among members of the deftpi@itd.nrl.navy.mil mailing list>’’.

3. Joshua D. Guttman, ‘‘A Proposed Interface Logic for Verification Environments’’, Tech. report M91-19, The MITRE Corporation, March 1991.

4. Kaufmann, Matthew J. and David Basin, ‘‘The Boyer-Moore Prover and Nuprl: An Experimental Comparison’’, Proceedings of Workshop for Basic Research Action, Logical Frameworks, Antibes, France. Also published as CLI Technical Report 58

5. R. S. Boyer and J S. Moore, ‘‘A Verification Condition Generator for FORTRAN’’, in *The Correctness Problem in Computer Science,* R. S. Boyer and J S. Moore, eds., Academic Press, London, 1981.

6. Robert S. Boyer and J Strother Moore, ‘‘Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic’’, Tech. report ICSCA-CMP-44, University of Texas at Austin, 1985.

7. Randal E. Bryant, ‘‘Graph-Based Algorithms for Boolean Function Manipulation’’, *IEEE Transactions on Computers*, Vol. C-35, No. 8, August 1986, pp. 677--691.

Table of Contents