10-2-2013

Formal Verification and Planning: An Evaluation

Rajesh Kalyanam
Electrical and Computer Engineering, Purdue University, rkalyana@purdue.edu

Tanji Hu
Electrical and Computer Engineering, Purdue University, hut@purdue.edu

Robert Givan
Electrical and Computer Engineering, Purdue University, givan@purdue.edu

Formal Verification and Planning: An Evaluation

Rajesh Kalyanam

Tanji Hu

Robert Givan

TR-ECE-12-12

October 2, 2013

School of Electrical and Computer Engineering

1285 Electrical Engineering Building

Purdue University

West Lafayette, IN 47907-1285

Formal Verification and Planning: An Evaluation

Rajesh Kalyanam and **Tanji Hu** and **Robert Givan**
Electrical and Computer Engineering, Purdue University, W. Lafayette, IN 47907
{rkalyana, hut, givan}@purdue.edu

Abstract

We explore the implications of improvements in hardware and knowledge representation for the application of automated reasoning systems to planning. We apply a novel automated-reasoning system with no planning-specific features to questions about the PDDL planning domains Blocksworld and (no airports) Logistics. Our system, with no human interaction and considering no specific problem instances, is able to verify all the key state invariants for both domains.

We propose organizing domain reasoning around (currently hand-written) recursive state-predicate–achievement macro-actions, such as a macro to achieve `clear`(b) in the Blocksworld. Leveraging (somewhat) limited human interaction, our system can completely characterize the effects of executing such recursive macros for each predicate in each domain. In addition, with substantial human interaction, our system can formally verify the solvability of arbitrary Blocksworld and Logistics problems, verifying a human-written generalized plan based on the macros. In each case, no specific problem instances are considered.

We loosely meter and qualitatively characterize the human interaction required for the above verifications in order to stimulate research to reduce this benchmark until it is zero. We propose (and where possible estimate the benchmark improvement for) plausible future approaches to reducing interaction, including eliminating the need for hand-definitions of the recursive predicate-achievement macros and generalized plans in these domains. Finally, we contrast our reasoning system favorably (for this task) with a widely used verification system, Coq.

1 Introduction

Early artificial intelligence researchers envisioned the dream of a question-answering system using logical entailment to interact with human users (McCarthy 1963). Systems using expressive representations and reasoning to derive plans were among the first explored to respond to the basic challenges of automated planning (Green 1969; Newell, Shaw, and Simon 1959; Fikes and Nilsson 1972). Decades were spent in attempts to design practically effective planners around the use of entailment together with logical representations of planning problems, e.g., (Reiter 1993; Knoblock, Tenenberg, and Yang 1991; Chapman 1987).

And yet for decades since that era, computer memory and time have become exponentially more available, year after year, making computer reasoning systems dramatically more effective. In addition, novel developments in representations and algorithms have enabled an increasing number of valuable applications involving reasoning systems (Bickford, Constable, and Rahli 2012; Klein et al. 2010; Strecker 2002; Liu and Moore 2004; Moore, Lynch, and Kaufmann 1998). In particular, in the last decade the interactive verification system Coq has been widely used to affirm the correctness of a range of programs (often identifying bugs along the way) (Leroy 2009; Ŝevčik et al. 2011).

In an interactive verification system, a formal proof of a claim from premises is constructed semi-automatically with the assistance of human interaction. The amount of human interaction needed to demonstrate a claim can be taken as a measure of how capable the automated system is on its own (McAllester and Givan 1993).

In contrast to the dream of leveraging entailment in expressive representations to find plans, heuristic search with mostly propositional representations has dominated AI planning research for decades, playing to the brute-force strengths of computer hardware. Search-based programs provide the best known domain-independent solutions to individual planning problems in most benchmarks. Broader goals, however, such as finding and verifying plans to solve entire domains ("generalized plans"), detecting domain invariants in expressive languages (like mathematical English), and recognizing many opportunities for problem decomposition have mostly eluded attack by heuristic search. Also, domain-independent heuristic search methods often fail to scale up to large problem instances as well as easily available domain-specific human solutions.

In this work we use interactive verification techniques to formally verify domain invariants and generalized-plan correctness in sample planning domains. The verified entailments include:

- (Blocks World) Nine invariants of the action dynamics, including:
  - no cycle of blocks can be created,
  - no block can have two blocks on it, or be on two blocks,
  - at most one block is held, with nothing on it, and
  - all blocks are exactly one of: held, on the table, or on a block.

- (Blocks World) For each state predicate, given a recursive macro to achieve that predicate: automatically verify the exact logical effects of the macro. For example, for the macro `make-clear`(b):
  - b and all blocks above b are clear, with the hand empty,
  - all blocks above b and any held block are now on the table,
  - all "on" relationships above b are no longer true, and
  - all other state facts are preserved.
- (Blocks World) Any goal state can be reached from any initial state, by checking a provided generalized plan.
- (Simplified Logistics) A similar set of verifications.
- (Branching-tower Blocksworld) Invariants are preserved.

These entailments are verified to hold independent of particular domain size, initial state, or trajectory of actions. Also, we discuss how macros and generalized plans similar to those provided can in many cases plausibly be automatically generated in future work.

It is not surprising that this machine verification is feasible, and in principle such a verification has been feasible for some time. Our result is significant because many of our verifications require only limited human interaction, and the remaining amounts of interaction can be (very loosely) benchmarked and targeted for elimination—here, we qualitatively characterize the remaining required interaction and propose approaches to eliminate much of it.

Important properties to note regarding this effort include: (1) our interactive verification system guarantees polynomial-time response at every interaction, and thus is not properly thought of as using "search," (2) the interaction language is very expressive — similar to mathematical English in expressiveness, and (3) the verification system is general-purpose and has no built-in knowledge of planning. The first two of these properties imply that the system cannot do anything resembling complete inference at each interaction step; however, the human/system pair is complete in that any entailed claim can be interactively proven.

We loosely measure the amount of human interaction needed to achieve such verification, thus providing a novel estimate of the "planning IQ" of an automated system. Our long-term goal is to eliminate the human interaction for those entailments that are obvious to humans, driving this estimate to zero, and field a system that automatically selects and verifies *obvious* domain invariants and generalized plans.

We conduct our interactive verifications in a recently developed general-purpose verification system, Ontic, release 12, with no planning-specific features.[1] Part of our contribution here is to demonstrate advantages of using the Ontic system for this task in comparison with the more well-known system Coq. We provide a direct comparison to Coq as part of this work.

The work reported here demonstrates that no domain-specific knowledge of planning domains is necessary for a reasoning system to verify many significant properties of benchmark planning domains with limited human assistance and no knowledge of planning apart from a mechanical translation of PDDL domains to logical representation, including some mechanically generated proofs about the translated PDDL.

Our verifications approach reasoning about planning domains in part by analyzing provided macros for achieving any single atomic formula. For example, we analyze a recursive macro `make-on`(s, a, b) for the Blocksworld that executes actions in state s to achieve a state with a on b. Part of our contribution here is to illustrate and discuss how thinking/reasoning about planning domains can be structured around mostly automated program analysis (and potentially generation) of such macros.

We summarize the representation language, proof language, and algorithmic features of the Ontic system in Section 2. We discuss the translation of PDDL (Planning Domain Description Language) domains into Ontic in Section 3. We then discuss benchmarking human interaction with Ontic, and evaluate Ontic's "planning IQ" on a series of planning-related verification problems in Section 4. We describe in detail the human interaction needed for some of the presented problems in Section 5, and discuss approaches to attaining independence from this interaction. We justify our choice to use Ontic by contrasting its benchmark performance with that of the widely used Coq verification system in Section 6. Finally, we compare our work with some recent work in generalized planning in Section 7.

2 The Ontic Verification System

We conduct our verifications in the Ontic verification system, release 12. For details on the original 1990 system, consult the cited book and papers on that system. (McAllester 1989; McAllester and Givan 1993; McAllester 1991) Full details of the Ontic release 12 system will be available in a forthcoming technical report. Here, we summarize key aspects of Ontic.

Representing Premises and Claims in Ontic

For accessibility and space reasons, when Ontic expressions are needed in this paper, we will present instead mathematical English translations. Full Ontic expressions for all verifications discussed herein are available upon request from the authors. Here, we discuss the expressive language features that make natural translations to and from mathematical English possible.

Class Expressions Formal claims made to the Ontic system are represented in the Ontic language. The central concept in the Ontic language is the *class expression*, an expression denoting a class of objects[2].

[1]Ontic is a descendant of a system by the same name created around 1990 but not used for published work at that time. The first version of Ontic was created by David McAllester (McAllester 1989; McAllester and Givan 1993). While very interesting, the 1990 system was generally too slow at that time to be effectively applied. The current Ontic is recently significantly adapted by the authors from that system but is based on the same principles.

[2]The most widely known class-based logics are *description logics* (Baader et al. 2003).

Classes of objects naturally generalize both logical terms (in Ontic, represented as classes containing exactly one object) as well as the familiar programming-languages notion of "type." Ontic class expressions can represent no values (the empty class of objects, such as "blocks that are on themselves"), single objects (such as logical terms or planning domain states or objects), or many values (such as the type "a tower of blocks in state s" representing all towers present in the state s). The primary purpose for generalizing terms to classes is to enrich the polynomial-time decidable fragment of the logic by enriching the expressiveness of the quantifier-free fragment (McAllester and Givan 1993); quantifier-free assertions about classes require quantifiers to express in standard predicate logic.

Single semantic values in Ontic are the logical domain objects, of course. The logical domain is closed under the formation of sets, sequences, and operators. In other words, any set of domain objects is itself another domain object, and likewise for sequences and operators[3][4].

Thus, Ontic class expressions can denote classes of set values, or classes of operator values, etc. For example, there is an Ontic class expression representing the type of all operators from states and blocks to blocks. The operator selecting the "clear block above block b in state s", i.e., the top block of the tower, given a state s and a block b, is then a single operator value of that type.

Ontic class expressions are written in a non-deterministic programming language (non-deterministic LISP), where the class represented is all possible non-deterministic values of the program expression. (Not all expressions are strictly executable, as the programming language is extended with declarative representational features.)

Ontic class expressions are formed compositionally from other class expressions and formulas recursively using operations with semantics such as union and intersection, filtering by a formula (keep only values making the formula true), and conditional choice between classes. A key case is the application of an operator to a class, collecting the image of the class members under that (possibly non-deterministic) operator. To illustrate the ability of this type of expression to construct interesting types concisely, consider the natural class "brothers of policemen" as applying the non-deterministic function "brothers-of" to the class of "policemen."

More advanced constructions to form class expressions include (dependently) typed recursive and non-recursive operator definitions, where the types are themselves arbitrary Ontic class expressions, as well as inverse and transitive closure applied to operators. Base case class expressions are the object constants and variables, denoting singleton classes.

The full language of class expressions allows the natural representation of nearly any class of objects that can be specified naturally in mathematical English. The ability to compose operator applications, conditionals, and recursion enables the definition of complex non-deterministic functions similar to LISP programming. We use this ability to create and analyze planning macros and generalized plans.

Throughout this paper, we will use mathematical English in discussing Ontic rather than giving a formal presentation of the Ontic language here, for reasons of space and to allow reader focus on the planning-related issues.

Formulas The primary Ontic atomic formula is the binary "is" formula, testing the subset relation between the classes. An "is" formula makes a typing or sub-typing assertion that one class is contained within another, generalizing the predicate logic equality test to classes. If both classes in the assertion are singleton (single-valued), then the "is" typing test reduces to the standard predicate-logic equality test. This enrichment of predicate logic to typing is vital to Ontic's performance.

Other atomic formulas are provided to assert that a class is non-empty, or that a class is deterministic (has at most one value). Boolean operators combining atomic formulas are provided, as well as universal and existential quantification (for convenience, as quantification is already representable using the atomic formulas together with the rich class expression language.)

Interactive Verifications in Ontic

Ontic supports the interactive development of verifications by providing a sound polynomial-time procedure for checking specific entailment claims. In one view, this *base reasoning procedure* models the human notion of what is "obviously entailed" (McAllester 1991). The procedure is sound but not complete; i.e., a possible response is "I don't know."

Ontic provides a very simple "Socratic proof system" (McAllester and Givan 1993) that enables a human user to prove any entailed claim by verifying a sequence of entailment claims with the base reasoner. The core of the Socratic proof system is that previously checked claims join the premise set automatically in future verifications, so that Ontic can verify a claim in two steps that cannot be verified in one step. The Socratic proof system also enables the user to specify case analyses, combining previously checked entailments, as well as universal generalization.

In this work, we will meter the length of a interactive proof by counting the number of base-reasoner Socratic proof steps needed to verify the final claim of the proof. Many of the verifications claimed in this paper require only one base-reasoner invocation, and thus no human interaction beyond posing the question.

While formally our Socratic proof system is a sequent system, with one sequent rule devoted to introducing base-reasoner-checked claims, Ontic provides a user interface for converting natural-style mathematical proofs into sequent proofs automatically. As a result, human-written Ontic proofs are exceptionally readable and similar in style to the informal proofs found in mathematics textbooks. See Figure 5 for a brief example. Informal proofs in textbooks expect the reader to "leap" between statements (entailment

[3]Ontic representations and inference are carefully designed to avoid Russell's paradox, instead providing objects only when the axioms of ZF set theory imply their existence

[4]Operators are typed (using Ontic class expressions as types), and the typings can be dependent (with the type of a later argument depending on the value of an earlier argument).

claims), filling in detail; Ontic proofs require the base reasoner to make similar leaps.

The base reasoner is implemented by a set of forward-chaining inference rules embodying the central basic properties of the syntactic constructs of the language. (For example, the rules implement the transitivity of subset, the simplification of conditionals when the test is known, the relationship between union and subset, etc.) These rules are used in a restricted forward-chaining procedure to answer entailment queries. The key restriction is that rules may fire only when no new expressions are introduced (all expressions appear in the premises or proposed conclusion, i.e., in the "query"); this restriction ensures a polynomial worst-case runtime (Givan and McAllester 1992).

This basic forward-chaining reasoning with the given expressions is enriched by multiple carefully-controlled mechanisms for limited introduction of new expressions into the forward-chaining reasoning. The most important of these is a novel matching-driven method for automatic quantifier instantiation, *subsumption matching*, that selects universal facts for instantiation when the resulting instance will provide a new type for an expression in the current query.

This instantiation method allows Ontic expressions in the query to be "type-checked" effectively during the polynomial-time basic reasoning, even though the "types" involved are general Ontic class expressions, so that the type language is undecidable[5].

Full source and executable for Ontic, release 12, is available upon request from the authors.

3 Translation from PDDL into Ontic

Automatic translation of PDDL domain descriptions into the Ontic language is generally straightforward. It is important, however, to select a translation that exploits the class-based reasoning available in Ontic. Rather than represent state information using predicates, the translation we use represents state information using Ontic operators and classes.

We mechanically translate single argument PDDL state predicates such as `ontable`(b) into non-deterministic Ontic operators from states to blocks. So, for instance, we have an Ontic class expression `(ontable-in` s`)` denoting the blocks that are on the table in state s.

Binary PDDL state predicates such as `on`(a, b) are also mechanically translated to non-deterministic operators. This construction is somewhat less familiar: given a block as input, along with the state, the operator yields a class collecting all related blocks under the binary predicate. So, the class `((on-in` s`)` a`)` denotes the class of all blocks on the block a in the state s. The PDDL translation does not include any information about `on` being functional—this is one of the invariants our system verifies upon request.

PDDL abstract action definitions are mechanically translated to Ontic operators mapping values for the action parameters to concrete actions. The action preconditions and effects of concrete actions are mechanically axiomatized following the PDDL action definition so that the class expression `(apply-action (pickup` b`)` s`)` denotes the state that results from picking up block b in state s.

1. 9 statespace invariants are preserved by 4 Blocksworld actions

a.	At most one block is held.	**b.**	Nothing is on the held block.
c.	At most one block is on a block.	**d.**	At most one block is under a block.
e.	No block on the table is held.	**f.**	No block on a block is held.
g.	No block on the table is on a block.	**h.**	No block is transitively on itself.
i.	Every block is either on the table, on a block, or held.		

2. Exact effects of `make-clear`(s, b) in Blocksworld.
   - **a.** Executing the macro results in a single reachable state.
   - **b.** The type "blocks on the table" adds any held block and all blocks above b.
   - **c.** The type "clear blocks" grows, adding b and all blocks above b in s.
   - **d.** The type "held blocks" becomes empty.
   - **e.** For each block d, the type "blocks on d" is unchanged, except when d is the block b or a block above b , when it becomes empty.
3. Exact effects of `make-held`(s, b) in Blocksworld.
   - **a.** Executing the macro results in a single reachable state.
   - **b-e.** Four claims characterizing the state predicates, similar to `make-clear`.
4. Exact effects, `make-ontable`(s, b), Blocksworld (5 claims).
5. Exact effects of `make-on`(s, a, b) in Blocksworld.
   - **a.** Executing the macro results in a single reachable state.
   - **b.** The type "held blocks" becomes empty.
   - **c.** For each d, the type `on`(d) becomes empty if d is a, above a, or above b; the type becomes a if $d = b$; the type is otherwise unchanged.
   - **d.** "Blocks on table" adds any held block, and all above a or b, but excludes a.
   - **e.** The type "clear blocks" adds a, and the blocks above a or b, but excludes b .
6. Two provided generalized plans are fully correct.
   - **a.** A plan to reach any goal state without ever picking up correct blocks.
   - **b.** A plan to place all blocks on the table.
7. 8 statespace invariants hold in branching tower Blocksworld.

Figure 1: Verifications conducted for Blocksworld

The frame problem is handled by choosing the form of these mechanically-generated (from the PDDL) action-definition axioms carefully. Our action-definition axioms state, for each predicate, an equation between the class of objects defining the predicate before the action and the new class of objects after the action. For instance, for `ontable` and `putdown`, the axiom used states that *the blocks on the table after executing a* `putdown` *action in state s are exactly the blocks on the table in state s with the first argument of the action added*:

```
(forall ((s (a-state))
         (a (a-putdown-action)))
  (= (ontable-in (apply-action a s))
     (either (ontable-in s) (first-arg a))))
```

and the axiom for `on` and `unstack` states that *the blocks on a given block d after executing an action* `unstack`(b, c) *in state s are the blocks on d in s with b removed, if d and c are equal, and are exactly the blocks on d in s, otherwise*:

```
(forall ((s (a-state))
         (a (a-unstack-action))
         (d (a-block-of s)))
  (= ((on-in (apply-action a s)) d)
     (if (= d (second-arg a))
         (except ((on-in s) d) (first-arg a))
       ((on-in s) d))))
```

In addition to axioms for each predicate and each action, capturing the frame and add/delete effects of each

[5] Please note that we are not claiming to solve an undecidable typing problem in polynomial time. Rather, the type checking is incomplete. However, many interesting typing facts within the rich class language can be automatically discovered by the polynomial-time basic reasoner (Givan 1997).

8. 5 statespace invariants are preserved by 3 Logistics actions
   - **a.** No package is `in` two trucks. **b.** No object is `at` two locations.
   - **c.** Each truck is `at` a location. **d.** No package is `in` a truck and `at` a location.
   - **e.** Each package is either `in` a truck or `at` a location.
9. Exact effects of $\texttt{make-at}_{\texttt{truck}}(s, t, l)$ in Logistics.
   - **a.** Executing the macro results in a single reachable state.
   - **b.** For each location l', the type $\texttt{at}(l')$ is unchanged, except $\texttt{at}(l)$ gains t and $\texttt{at}(\texttt{at}^{-1}(t))$ loses t.
   - **c.** For each truck t', the type $\texttt{in}(t')$ is unchanged.
   - **d.** For each object o, $\texttt{location}(o)$ is unchanged, except $\texttt{location}(t) = l$.
10. Effects of $\texttt{make-at}_{\texttt{pkg}}(s, p, l)$ in Logistics.
    - **a-d.** Like the other macros, except that which truck is used is left unspecified.
11. Exact effects, $\texttt{make-in}(s, p, t)$, Logistics (4 claims).
12. A provided generalized plan is correct.

Figure 2: Verifications done for Simplified Logistics (trucks only)

action, the translation from PDDL instructs Ontic to perform some simple domain-independent verifications regarding the action definitions just created. These brief verifications serve to ensure that Ontic sees the basic domain-independent properties of the above-described axioms and definitions. Also included are a few basic planning definitions such as "reachable-state". Complete Ontic translations of the studied domains are available upon request from the authors.

The PDDL translation is the only source of planning-specific information used in our system apart from the metered human interaction, as Ontic is otherwise a general-purpose system designed for arbitrary mathematics, not planning.

4 Planning-related Verifications in Ontic

In Figures 1 and 2, we present in English the entailments we have verified with the Ontic system. We provide numeric benchmarks on the number of human interaction steps (Socratic questions/steps contributed by the human in the Socratic proof) needed to achieve each verification in Table 1. The numeric benchmark does not count the invocation at the top level, so if a verification is fully automatic, the benchmark is zero.

See Figure 3 for examples of the predicate-achievement macro definitions analyzed here. For each predicate-achievement macro, there is verified claim stating the effect on each state predicate as well as a claim of well-definedness (implying that all actions taken are applicable when taken, termination, etc.). This results in five verified claims for each macro in the Blocksworld and four for each macro in Logistics.

Our Blocksworld has four actions (`stack`, `unstack`, `pickup`, `putdown`) and four state predicates (`on`, `ontable`, `held`, and derived `clear`). Our (simplified) Logistics has three actions (`load`, `unload`, `drive`) and three state predicates ($\texttt{at}_{\texttt{truck}}$, $\texttt{at}_{\texttt{pkg}}$, `in`), and no airports or airplanes. Branching-tower Blocksworld, is the same as Blocksworld except that `clear` is not required for `stack` and the invariant "at most one block is on a block" is lost.

make-clear(s, b)— If the type `(holding-in` s`)` is occupied, then take `putdown` on its occupant. Then, if the type `((on-in` s`)` b`)` is occupied by block a, i.e., there is a block a on b in s, then recursively $\texttt{make-clear}(a)$, take `unstack` a, and take `putdown` a.

```
(define (make-clear (s (a-state)) (b (a-block-of s)))
  (if (there-exists (holding-in s))
      (make-clear (take (putdown (holding-in s)) s) b)
      (if (not (there-exists ((on-in s) b)))
          s
          (take (putdown ((on-in s) b))
            (take (unstack ((on-in s) b) b)
              (make-clear s ((on-in s) b)))))))
```

make-on(s, a, b)— Make b clear, then make a held, then stack a on b.

```
(define (make-on (s (a-state))
                 (a (a-block-of s))
                 (b (a-block-of s) such-that (not (= a b))))
  (take (stack a b) (make-held (make-clear s b) a)))
```

Figure 3: Example Predicate-achievement Macros in English. Ontic source is shown only for reference, not fully explained.

Some comments on the benchmark in Table 1 are in order. First, this measurement is necessarily vague. The number of interactions required will depend heavily on the expertise and effort of the human user. This issue is somewhat addressed by seeking a minimal number for each verification, over multiple users. Second, simply counting the interactions needed does not reflect the difficulty for the user in finding the necessary Socratic query to advance the proof. In our system, the natural proof style helps mitigate this issue—the necessary input will nearly always be a natural statement about the proof context. A user who could convince a mathematician can generally also convince our system. Also, in practice, we find that a low number of necessary interactions nearly always indicates an easier proof to construct, with the system providing excellent support. We note that the brevity of explanation needed by a human listener is a meaningful measure of his or her intelligence.

5 Discussion and Directions for Improvement

Our primary goal has been to report an evaluation of unspecialized, general-purpose verification as a planning tool; specifically, of Ontic as a planning tool. However, we have a secondary goal of describing the resulting verifications qualitatively and suggesting/evaluating potential directions of improvement for verification systems, with the eventual target of fully automating the verifications we evaluated. These directions of improvement include both possible general-purpose reasoning methods as well as directions for adding planning-specific features to reasoners. In this section, we discuss our verifications for this purpose.

We note that, in general, our verifications (available upon request) can be used by anyone to reveal which steps in a proof current systems are not finding automatically and to inspire general methods that will find these steps. Here, we discuss and estimate the benchmark improvements for the most readily indicated directions.

First, we observe that the greatest difficulty was encountered in verifications 2a, 5c, 5e, and 6. Here, we discuss the main difficulty in each of these.

2a The challenge in showing `make-clear` (see Figure 3) is well-defined is primarily that we must show a lemma that every block is under a clear block (so the base case is reached). Examination of the macro definition reveals that this lemma statement could be extracted automatically. The lemma itself follows from the lack of cycles (invariant 1h) and the finiteness of the set of blocks.

5c & 5e The analysis of `make-on` (again see Figure 3) requires composing the analysis of `make-clear` and that of `make-held`. This composition amounts to progressing state through a sequence of macro-actions, a much more difficult analysis than PDDL progression, and is discussed briefly below. Future work on separating frame and non-frame effects of macros automatically (very easy in PDDL actions, but harder for arbitrary macros) is needed here. In addition, a stronger library of general-purpose lemmas about transitive closures would reduce the key difficulty here, especially when reasoning about `clear`.

6 The verification of a Blocksworld generalized plan that does not move correct blocks requires all the types of human interaction discussed in this section and more, and remains clearly beyond full automation today. Our verification demonstrates that the easily accessible ideas for reducing interaction discussed below can directly reduce the interaction in this proof to 370 interaction steps. Further ideas are needed and can be motivated by examining these steps; by no means is the system optimized with regard to general techniques that may minimize many of the interactions involved. Fully automating this verification would be a very significant accomplishment standing on its own, for future work.

We observe that our system easily verifies all the key statespace invariants in the domains tested, with no human interaction, having eliminated the human interactions required by Coq as discussed in Section 6.

Although we have assumed here that the invariant statements are provided, note that the monotonicity of entailment implies that we can equally provide any superset of these invariants and a straightforward filtering process will narrow the set to those for which preservation under the actions can be verified. (Simply repeatedly drop those invariants for which preservation cannot be verified.) So a method that generates candidate invariants by pattern matching can replace the human input as long as it generates a superset of these invariants.

The system can also verify many of the macro effects without human interaction (2b, 2c, 2d, 2e, 5b) and the rest require very limited human interaction (except 5c and 5e discussed above). We now focus on what clearly automatable interactions the macro analysis verifications revealed.

Many of the human interactions required by Ontic in our evaluation fall into categories that appear automatable by extending the reasoner in the following directions. Only the first item is general-purpose, with the others being planning specific:

Type checking 28 interactions directed Ontic to verify that an action a in an expression $\texttt{take}(a, s)$ is applicable in s. This expression is a special case of typed function application with expressive types (arbitrary class expressions), and we naturally envision a system that automatically attempts to show well-typedness of any application expression explicit in a proof. Ontic should, but currently does not, automatically attempt to prove that argument expressions for a function meet the declared type restriction of the function, noting that in the Ontic language this is an undecidable question. 58 interactions would by examination be eliminated by adding such a (not planning-specific) system.

PDDL Type invariance 16 interactions were required to make explicit the invariance of the type meanings (i.e., no new blocks appear). While our system can handle richer domains with block creation, we believe a planning-specific modification to the reasoner can inexpensively automatically eliminate all these interactions.

Precondition checking and regression assistance While type checking can automatically direct the system to check applicability of an action, a more planning-specific adaptation of the reasoner would be even more aggressive in this case and specifically logically regress this test into an attempt to show that each precondition is satisfied. In the case where a composition of actions or macros is present, the reasoner could be designed to continue this regression (on explicit expressions only, not as an exploration of statespace). Implementing this further change would appear to eliminate 20 further interactions.

Reachability 20 interactions directed Ontic to verify that macros and action applications return reachable states. A planning-specific reasoner could conceivably be designed to automatically try such verifications without direction

Verif. #	# Steps	Reduced Steps	English name
1	0	0	9 blocks invariants
2a	21	10	`make-clear` well-defined
2b-e	0	0	`make-clear` effects
3a	4	0	`make-held` well defined
3b-e	11	5	`make-held` effects
4	9	0	`make-ontable`, 5 claims
5a	11	0	`make-on` well defined
5b	0	0	effect on `held`
5c	24	24	effect on `on`
5d	9	6	effect on `ontable`
5e	28	21	effect on `clear`
6a	418	370	generalized plan for any goal
6b	14	14	gen. plan to put all blocks on table
7	0	0	8 branching-tower blocks invariants
8	0	0	5 logistics invariants
9a	2	0	$\texttt{make-at}_{\texttt{truck}}$ well defined
9b-d	1	0	effects of $\texttt{make-at}_{\texttt{truck}}$
10a	6	1	$\texttt{make-at}_{\texttt{pkg}}$ well defined
10b-d	8	8	effects of $\texttt{make-at}_{\texttt{pkg}}$
11a	9	4	`make-in` well defined
11b-d	3	3	effects of `make-in`
12	25	24	generalized plan

Table 1: Counts of Human Interactions for Each Verification Evaluated. The "verification number" refers to the line number in Figure 1 or Figure 2. The "reduced steps" column refers to the reduction achievable by eliminating interactions listed in Section 5.

when actions or macros are being applied.

We believe fully implementing the above ideas would substantially reduce the interaction needed in the evaluated verifications, resulting in the improved (not yet achieved) benchmark shown in the final numeric column of Table 1. One purpose of this evaluation has been to identify and motivate this further work.

Human-written macros and plans Our verifications require human-written predicate-achievement macros and generalized plans. However, note that any process for automatically constructing such macros and plans would be tremendously aided by the ability to automatically analyze candidate definitions.

We note that, for simple domains like Blocksworld, natural techniques can propose mutually recursive predicate achievement macros for analysis, automatically. For instance, for any action that adds a predicate, one can construct a predicate-achievement macro that seeks to achieve the preconditions of that action (using recursive calls to other predicate-achievement macros) and then takes that action. This is precisely the structure of most of our macros. Analysis can then be used to determine a successful ordering for the precondition achievement calls. A first step towards such a method is automating the verifications presented herein.

Similarly, analysis of the predicate achievement macros can suggest or imply orderings for achieving conjunctive goals, leading to a provably correct generalized plan. This is currently further out of reach, but the predicate-preservation properties of the macros proven in our analysis do imply directly that block towers must be built bottom-up and this could be the basis for a future method for automatically constructing the generalized plan.

6 Planning-related Verifications in Coq

There are a number of published reasoning systems that could be considered for performing similar verifications to those reported here, e.g., (Bickford, Constable, and Rahli 2012; Klein et al. 2010; Strecker 2002; Liu and Moore 2004; Moore, Lynch, and Kaufmann 1998), with varying amounts of human interaction required. Here we discuss brief experiments conducting planning verifications using the Coq system and compare the resulting verifications with our Ontic verifications.

We limit our discussion here to comparing Coq and Ontic representational features and their effects on reasoning about planning, in particular on verifying Blocksworld invariants. Our experiments have shown a requirement for more human interaction in conducting the Coq verifications. [6]

We presented nine statespace invariants that are preserved by the four Blocksworld actions in our representation. There are thus 36 preservation theorems stating that a given action

[6]In principle, Coq and other systems can be adapted to conduct these verifications with similar results to ours with Ontic; one purpose of this paper is to stimulate research in the application of a variety of systems to planning domains for comparison, system development, and expected reduction over time of the required human input.

```
Definition det_holding (s : state) :=     Defining the invariant:
  forall b1 b2, holding b1 s ->               at most one block is
    holding b2 s -> b1 = b2.                being held in state s

Theorem pickup_det_holding :
  forall x s,
    osr s (take (pickup x) s)     Hypothesis: (take (pickup x) s)
                                     is one step reachable from s
    -> det_holding s              Hypothesis: holding at-most-one
                                                       block in s
    -> det_holding (take (pickup x) s).      Theorem: After taking
                                             (pickup x), will hold
                                                 at-most-one block
Proof.
  intros x s Hosr Hprev.             Names variables and hypotheses
  inversion Hosr.                Regress one step reachable hypothesis
                                     to action preconditions of pickup
                                         -->results in H2, used below
  unfold det_holding.             Expand the definition of det_holding
                                                   in the proof goal
  intros b1 b2 Hb1 Hb2.          Names for variables and hypotheses
                                     generated in the previous step
  apply pickup_holding in Hb1.             Apply the action dynamics
                                             to get b1=x ∨ holding b1
  apply pickup_holding in Hb2.             Apply the action dynamics
                                             to get b2=x ∨ holding b2
  unfold handempty in H2.          Expand the definition of handempty

  assert (not (holding b1 s)) by apply H2.   Instantiate H2 on b1

  assert (not (holding b2 s)) by apply H2.   Instantiate H2 on b2

  intuition.                      Figure out propositional tautologies
  congruence.               Prove b1 = b2 based upon b1 = x and b2 = x
Qed.
```

Figure 4: An example proof in Coq. The red lines (unfold, assert) are domain-specific human inputs. Green lines (inversion, apply) could easily be automated in a planning-specific system. Both count as interactions here for comparison to Ontic.

```
(suppose-there-is ((s (a-legal-state))    Invariants assumed in s
                   (a (an-applicable-action-in s)))
  (suppose (is a (pickup (a-block-of s)))
    (show (at-most-one (holding-in (take a s))))))
```

Figure 5: The Ontic proof corresponding to Figure 4.

preserves a given invariant. Each of these theorems is proven without human interaction by the Ontic system. None of these 36 theorems is accepted without interaction by the Coq system, even if we presume that common tactics such as “auto” and “simpl” are automatically tried.

The needed human interaction varies between the 36 verifications and requires human expertise to generate. Typically a small number of carefully human-specified theorem instantiations are required. Over the entire set of 36 invariant-preservation verifications, our Coq work required 237 human proof interactions, not counting stating the 36 theorems or any human invocations of the tactics intros, auto, simpl, intuition (when no subgoals are generated), and unfolding of definitions appearing in the theorem, which we assume can be easily automated. We also made sure to create tactics for common interaction sequences in order to count them as one interaction when invoked. We show one example Coq verification in Figure 4 and the corresponding Ontic proof in Figure 5.

We discuss dimensions along which Ontic provides what appear to be qualitative advantages over Coq for our planning verifications.

Taxonomic Representations Due to the class-based syntax used in Ontic, as discussed above, the quantifier-free fragment of the Ontic language is more expressive than that of Coq. Ontic can state formulas such as $\forall x \exists y \, \texttt{ontable}(y) \wedge \texttt{on}(x, y) \rightarrow \texttt{clear}(x)$ without quantifiers (as the formula `(is (on ontable) clear)` stating a subtyping relationship between the type "on a block on table" and the type "clear block"). This quantifier-free expressiveness reduces the number of theorem instantiations the system needs to find, and thus reduces the human interaction in assisting in finding such instantiations.

Type-based Automatic Quantifier Instantiation Ontic includes a general-purpose mechanism, *subsumption matching*, for automatically selecting instances of universal theorems when those instances will help provide new typings for expressions in the targeted theorem. This mechanism is much richer than simply instantiating theorems with conclusions that match pieces of the desired theorem—the effect of the "auto" tactic applied to a desired theorem (somewhat simplified). As a result, again, fewer human-selected instantiations are needed for the Ontic proofs in our verifications than the Coq proofs. 146 of the 237 human interactions needed in our verifications of the invariants in Coq are human-selected instantiations, where no human interactions were needed in Ontic.

Automated Case Analysis Ontic includes an automated case analysis mechanism that automatically introduces a variety of case splits when the target entailment is "obvious" to the basic reasoner on one side of the split. Because one side is checked in polynomial-time without further case analysis, this method has an overall linear cost in the number of potential case splits considered. This mechanism eliminates the need for human interaction to select case splits for the analysis. This issue did not limit Coq in the invariant proofs but can in other settings contribute significantly to human interaction requirements for Coq in comparison to Ontic.

Declarative vs Procedural Proof Style The use of Socratic proof in Ontic results in a style of interactive proof in which the human tries simpler and simpler claims about the proof context until a claim is found that the system can verify but did not already know, and then repeating this process in the presence of the new theorem, until the target theorem is accepted. This process is roughly analogous to the process a human teacher might use to teach by asking Socratic questions. The resulting Ontic proofs are declarative in nature and consequently quite natural for humans to read, understand, and learn to generate.

In contrast, the Coq interface expects the user to identify a "tactic" that can be used to reduce a current goal to subgoals. Thus the user is expected to identify a procedural approach to the proof, and can even write such procedural approaches by writing tactics. We believe this interface encourages the development of more and better tactics, by users and implementors, but does not lead to a focus on automatically deploying and combining tactics effectively in a domain-independent manner. As a result, Coq proofs often rely heavily on the human for selecting and combining tactics. Because of the procedural style of human-Coq interaction, the resulting proofs do not resemble human proofs and are readable only to Coq users as specifications of the procedures needed to conduct the verification [7].

The Socratic proof system of Ontic forces Ontic implementors to provide a single reasoning tool, the basic notion of "obvious entailment", i.e., the Ontic base reasoner, that is as strong as possible. This forces implementors to automate tradeoffs and interactions between strategies such as case analysis, Skolemization, and universal instantiation. This can be a drawback when considering the power of the human/machine combination, as often the human can handle the tradeoff in making tactic selections better than the current Ontic would. However, this structure does encourage the study of the reduction/elimination of human input.

7 Related Work

Here we compare briefly to the generalized planning work of (Srivastava et al. 2011). This very nice work provides a domain-independent approach to automatically constructing a verified generalized plan for a class of domains that includes both the Blocksworld and Logistics. The work exploits a *role*-based abstraction to represent a generalized plan and detect loop termination by decreasing role counts.

There are several respects in which this prior work is not directly comparable to our work. First, their method assumes domain invariants rather than verifying them. Second, the applicability of their method is unclear outside of $\mathcal{FC}^3$ domains, where the role-based abstraction can represent the choices needed. We show our results on the branching-tower Blocksworld statespace invariants, a non-$\mathcal{FC}^3$ domain, to briefly demonstrate how easily our approach moves outside that restriction. Third, we note that our class expressions provide a richer abstraction than roles and will support a wider range of analyses. Finally, the reliance on role-based abstraction leads to sensitivity to the introduction of irrelevant predicates, which can be difficult to detect. Our verifications reported here may also be affected, but less so; an irrelevant predicate can generally be expected to have no effect due to the monotonicity of entailment.

The Srivastava et al. methods should be considered for incorporation as planning-specific methods with a general-purpose reasoner to improve its planning IQ benchmark.

8 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0905372.

References

Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; and Patel-Schneider, P. 2003. *The description logic handbook:*

[7]A declarative account of the subgoals generated could be automatically extracted, but would not be an account of the choices made by the human user and would contain more detail than a human reader would expect.

Theory, implementation and applications. Cambridge University Press.

Bickford, M.; Constable, R.; and Rahli, V. 2012. The logic of events, a framework to reason about distributed systems. Technical report, arXiv: 1813: 28695, CIS, Cornell University.

Chapman, D. 1987. Planning for conjunctive goals. *Artificial intelligence* 32(3):333–377.

Fikes, R., and Nilsson, N. 1972. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3):189–208.

Givan, R., and McAllester, D. 1992. New results on local inference relations. In *Principles of Knowledge Representation and reasoning: Proceedings of the Third International Conference (KR92)*, 403–412.

Givan, R. 1997. Obvious properties of computer programs. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*.

Green, C. 1969. Theorem-proving by resolution as a basis for question-answering systems. *Machine Intelligence* 4:183–205.

Klein, G.; Andronick, J.; Elphinstone, K.; Heiser, G.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S. 2010. seL4: formal verification of an operating-system kernel. *Communications of the ACM* 53(6):107–115.

Knoblock, C. A.; Tenenberg, J. D.; and Yang, Q. 1991. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference on Artificial intelligence - Volume 2*, AAAI'91, 692–697. AAAI Press.

Leroy, X. 2009. Formal verification of a realistic compiler. *Communications of the ACM* 52(7):107–115.

Liu, H., and Moore, J. 2004. Java program verification via a JVM deep embedding in ACL2. *Theorem Proving in Higher Order Logics* 117–125.

McAllester, D., and Givan, R. 1993. Taxonomic syntax for first order inference. *Journal of the ACM* 40(2):246–283.

McAllester, D. 1989. *Ontic: A Knowledge Representation System for Mathematics*. MIT Press, Cambridge, MA.

McAllester, D. 1991. Observations on cognitive judgments. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 910–915. Morgan Kaufmann Publishers.

McCarthy, J. 1963. *Programs with common sense*. Defense Technical Information Center.

Moore, J. S.; Lynch, T.; and Kaufmann, M. 1998. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm. *IEEE Transactions on Computers* 47(9):913–926.

Newell, A.; Shaw, J.; and Simon, H. 1959. *Report on a general problem-solving program*. Rand Corporation.

Reiter, R. 1993. Proving properties of states in the situation calculus. *Artificial Intelligence* 64:337–351.

Ŝevčik, J.; Vafeiadis, V.; Zappa Nardelli, F.; Jagannathan, S.; and Sewell, P. 2011. Relaxed-memory concurrency and verified compilation. *ACM SIGPLAN Notices* 46(1):43–54.

Srivastava, S.; Immerman, N.; Zilberstein, S.; and Zhang, T. 2011. Directed search for generalized plans using classical planners. In *Proc. of the International Conference on Automated Planning and Scheduling, Freiburg, Germany*, 226–233.

Strecker, M. 2002. Formal verification of a Java compiler in Isabelle. *Automated Deduction—CADE-18* 135–163.