Formal Metatheory using Implicit Syntax, and an Application to Data Abstraction for Asynchronous Systems

Amy P. Felty[1], Douglas J. Howe[1], and Abhik Roychoudhury[2]

[1] Bell Labs, Murray Hill, NJ 07974, USA. {felty,howe}@bell-labs.com
[2] Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11790, USA. abhik@cs.sunysb.edu

Abstract. Abstraction is a useful tool in verification, often allowing the proof of correctness of a large and complex system to be reduced to showing the correctness of a much smaller simpler system. We use the Nuprl theorem prover to verify the correctness of a simple but commonly occurring abstraction. From the formal proof, we extract a program that succeeds when the abstraction method is applicable to the concrete input specification and in this case, computes the abstracted system specification. One of the main novelties of our work is our "implicit syntax" approach to formal metatheory of programming languages. Our proof relies entirely on semantic reasoning, and thus avoids the complications that often arise when formally reasoning about syntax. The semantic reasoning contains an implicit construction of the result using inductive predicates over semantic domains that express representability in a particular protocol language. This implicit construction is what allows the synthesis of a program that transforms a concrete specification to an abstract one via recursion on syntax.

1 Introduction

Theorem proving and model checking can be usefully combined by using a theorem prover to verify abstractions of protocols or system specifications. In particular, one can often use a model checker to verify some property of a protocol that has an infinite or intractably large state space, by first transforming the original or *concrete* protocol into a more *abstract* version for which model checking is feasible [13, 2]. A theorem prover can be used to check, for example, that the property (or some transformation of it) holds of the abstract protocol if and only if it holds of the original protocol. This can be done directly by formalizing the two versions of the protocol and proving the specific property of interest. This approach is taken in [7], for example, using the integration of a BDD based model checker as a decision procedure in PVS [11]. One can also, as in [10], provide general support for doing this kind of reasoning by formalizing a refinement calculus and methodology relating system specifications and abstractions; or as in [5], use a model checker with assumption commitment style reasoning on the abstract system and then use a theorem prover to discharge the assumptions in the concrete system.

Typically, when a system specification is represented in a theorem prover, a so called "shallow embedding" is used. In a shallow embedding of a programming or specification language in a theorem prover, programs and specifications are directly interpreted in the logic of the theorem prover. Thus, one formalizes only the semantics of the language. For example, the commands of an imperative programming language might be encoded as objects of type $com = state \to state$.

In contrast with shallow embedding is "deep embedding", where both the semantics *and* syntax of the embedded formalism are represented in the theorem prover. Using this approach, one might have a type *comsyn* consisting of abstract syntax trees of commands, and then a meaning function $M \in comsyn \to com$. Deep embeddings are considerably more difficult to reason about in theorem provers. In practice, shallow embeddings are used whenever possible, and deep embeddings are done only when one is interested in some property that cannot be expressed by referring to semantic objects alone. A comparison of these two methods is presented in [1], for example.

In this paper we show how to exploit the constructivity of the Nuprl theorem prover [4] to synthesize a particular verified-correct abstraction algorithm. We build a proof in Nuprl from which we can extract a program that takes a concrete specification as input, tests whether the abstraction method applies to it, and if so, returns the abstracted system specification.

One of the main novelties here is that we do *not* use a deep embedding. Our proof reasons only about semantics, yet we are able to synthesize a program that operates on syntax. Thus we reason only about the semantic aspects of the abstraction method, even though we are implicitly constructing the program that builds abstracted programs. The central idea is to define inductive predicates over semantic domains that express representability in a particular protocol language. We give a small illustrative example of the approach in Section 2.

It is not obvious that this notion of representability is adequate for non-trivial examples. Many concepts that are natural in reasoning about syntax cannot be directly expressed. For example, we cannot directly write down a function which takes a command and returns a list of all program variables occurring in it, since a command is just a function on states that is assumed to be representable, and we cannot in general determine the list of variables from this function.

As evidence for the practicality of our approach, we apply it to a simple but common *data abstraction* method. The correctness of the abstraction, as well as the representability of the abstract system specification, was proved in Nuprl. We used the program extracted from these proofs to obtain the abstraction of a simple communication protocol.

The only other paper we know of that uses the idea of representing syntax implicitly in type theory via an inductive predicate is [3], where it is proposed as a way of defining internal computational complexity measures. Nothing was implemented, and no proofs are given. Furthermore, the paper does not address the use of implicit syntax together with extraction to synthesize metaprograms.

We are aware of one other effort involving program extraction and model checking [12], in which the correctness proof of a model checker in the Coq proof checker yields, via extraction, an executable model checker which is then considered as a trusted decision procedure.

2 Example

To illustrate our approach, we consider a simple, rather artifical, example involving a trivial imperative programming language P where programs are sequences of assignments of variables to variables. We start with a semantic account of the language. Define $\mathit{state} = \mathit{string} \rightarrow \mathit{int}$, and define the type of (meanings of) commands to be $\mathit{com} = \mathit{state} \rightarrow \mathit{state}$. Assignment and command sequencing can now be defined semantically:

$$\begin{aligned} \mathit{assg} : \mathit{string} \rightarrow \mathit{string} \rightarrow \mathit{com} &= \lambda x.\, \lambda y.\, \lambda s.\, s[x \leftarrow s(y)] \\ \mathit{seq} : \mathit{com} \rightarrow \mathit{com} \rightarrow \mathit{com} &= \lambda c_1.\, \lambda c_2.\, \lambda s.\, c_2(c_1(s)) \end{aligned}$$

where $s[x \leftarrow l]$ is the state which maps x to l and all other variables y to $s(y)$. Write $x := y$ for $\mathit{assg}(x)(y)$ and $c_1; c_2$ for $\mathit{seq}(c_1)(c_2)$.

Consider the following fact about P: for every command c, if there is a variable x such that c only affects the value of x, then c is equivalent to a single assignment statement. This fact is false in general for members of com; to formalize it, we need to somehow formalize the syntax of P. We do this by inductively defining a representability predicate $R : \mathit{com} \rightarrow \mathit{prop}$ as follows.

$$\begin{aligned} R(c) \Leftrightarrow\ & \exists\, x, y \in \mathit{string}.\ c = (x := y) \\ & \vee\ \exists\, c_1, c_2 \in \mathit{com}.\ R(c_1)\ \&\ R(c_2)\ \&\ c = (c_1; c_2) \end{aligned}$$

Define $u(c)$, for $c \in \mathit{com}$, if there exists $x \in \mathit{string}$ such that for all $y \neq x \in \mathit{string}$ and all $s \in \mathit{state}$, $s(y) = c(s)(y)$. Our fact may now be formalized as follows.

$$\forall\, c \in \mathit{com}.\ R(c) \Rightarrow u(c) \Rightarrow \exists\, x, y \in \mathit{string}.\ c = (x := y)$$

Unfortunately, the obvious proof attempt, using induction on the definition of $R(c)$, fails, because when $c = (c_1; c_2)$, it is not necessarily true that $u(c)$ implies $u(c_1)$ and $u(c_2)$.

If we strengthen the assumption on the representation of c to require that u hold of all subcommands, then the obvious proof will work. We can state this property by modifying the definition of representability. In particular, define $R_q(c)$, for q a predicate on com, by replacing $R(c)$ by $R_q(c)$ in the definition of $R(c)$ and conjoining the right-hand side of the definition with $q(c)$. We can prove

$$\forall\, c \in \mathit{com}.\ R_u(c) \Rightarrow \exists\, x, y \in \mathit{string}.\ c = (x := y)$$

by a straightforward induction on $R_u(c)$. Since we are formalizing in a constructive logic, the proof will yield a program that takes a c and evidence for $R_u(c)$ and produces the x, y such that $c = (x := y)$.

In order to run this program on a particular c, we need a proof of $R_u(c)$. Since there is no general method in the type theory to go from a member of *com* to a representation, we assume we are given a proof of $R(c)$. The method for going from c to $R(c)$ can be implemented in Nuprl's metalanguage. The problem is now to go from $R(c)$ to $R_u(c)$. This is not always possible, so we choose, as a simple sufficient condition, to do this only for commands whose constituent assignments have a unique variable on the left hand side.

To deal with this kind of syntactic sufficient condition, we define a *possibility* operator on propositions, denoted ?. In particular, the formula $?A$ is defined to be $A \vee True$. Clearly, for any A, $?A$ holds because the right disjunct is provable. When we prove a theorem whose conclusion is $?A$, we take care to choose to prove the left disjunct (A) in situations where the sufficient condition is known to hold. Theorems of this form give a partial correctness result. The program that Nuprl extracts from the proof will either return a result of type A or the constant $axiom$ which is the proof of $True$. The fact that it does not return the trivial result $axiom$ when the sufficient condition holds is purely a metatheoretic one.

Returning to the example, we can prove a theorem $\forall\, c \in com.\ R(c) \Rightarrow ?R_u(c)$ by induction on the definition of $R(c)$. If we construct the right kind of proof, the extracted program will translate evidence for $R(c)$ into evidence for $R_u(c)$ in the case where c satisfies the sufficient condition given above.

3 Protocol Verification in Nuprl

For our data-abstraction case study, we use the environment for protocol verification that was built in the course of verifying the SCI cache coherence protocol [6]. Here, we briefly describe our shallow embedding of a Unity-like guarded command language in which protocols are expressed. We illustrate this language with our running example presented in Fig. 1. In this language, a *specification* or *program* is a list of guarded *actions*, each having a *guard* and a *body*, along with an initial condition on values of program variables. In general guards can be message receives or boolean conditions, and bodies can contain assignments, conditionals, and message sends. The example presents a protocol with three distinct processes—Sender, Channel (or Queue), and Receiver, denoted s, q, and r, respectively. Consider the two actions for the Receiver, marked r_1 and r_2. In r_1 the guard always holds and the body contains a message to the sender requesting data, where *request* is the *message type*. A message can also contain arguments as illustrated in r_2. Here, the guard indicates that this action can be executed if the first message in $buf[r]$ (r's message buffer) has type *rsend*. This message has one argument, *rdata*, containing the requested data. The message is removed from the queue (received) and the body is executed. The Sender and Channel processes both have a single parametrized action (s_d and q_m respectively). Action s_d is parametrized by the value of the data transmitted to the Channel. Action q_m is parametrized by the length of the channel. Thus, the above specification represents a collection of programs where the number of data

Program P:
Initial Condition: $sent = -1$
Actions: $[r_1, r_2, s_0, \ldots, s_{n-1}, q_m]$ for some $n > 0, m \geq 0$

(r_1) $true \longrightarrow s!request$
(r_2) $buf[r]?rsend(rdata) \longrightarrow rcvd := rdata$

(s_d) $buf[s]?request \longrightarrow sent := d; q!qsend(sent, 0)$

(q_m) $buf[q]?qsend(data, i) \longrightarrow$
$\quad$ **if** $i = m$ **then** $r!rsend(data)$ **else** $q!qsend(data, i+1)$ **fi**

Property ψ:
Data delivery: $\forall y.\ G(sent = y \Rightarrow F(rcvd = y))$
Order preservation: $\forall y_1.\ \forall y_2.\ G((sent = y_1 \wedge F(sent = y_2)) \Rightarrow F(rcvd = y_1 \wedge F(rcvd = y_2)))$

Fig. 1. Running Example

values and the length of the queue are bounded by an arbitrary finite number. We prove universally quantified linear time temporal logic properties (such as the data delivery property in Fig. 1) of the example protocol, by performing data abstraction of the protocol.

Nuprl is a goal-directed interactive theorem prover in the style of LCF. It implements a constructive type theory with a rich set of constructors. Because of the constructivity, programs can be extracted from proofs. Logic is encoded via the propositions-as-types principle, whereby a proposition is identified with the type of data that provides evidence for the proposition's truth. The version of Nuprl we use [8] also supports classical reasoning, which can be used in any part of a proof that does not affect the extracted program. Formal mathematics in Nuprl is organized in a single library, which is broken into files simulating a theory structure. Library objects can be definitions, display forms, theorems, comments or objects containing ML code. Definitions define new operators in terms of existing Nuprl terms and previously defined operators. Display forms provide notations for defined and primitive operators. Theorems have tree structured proofs, possibly incomplete. Each node has a sequent, and represents an inference step. The step is justified either by a primitive rule, or by a *tactic*. Tactics provide automation to help with goal-directed search.

Our embedding of the semantics of state transition systems in Nuprl is fairly straightforward. We define a state as a pair where the first component is the usual mapping from identifiers to values (integers). The second component is a *history* variable that records the sequence of messages that have been sent and received during the entire execution. This history variable is important for reasoning about data that passes via messages. Messages have two components.

Message types such as *rsend* are encoded as integers as the first component of a message. The second component is a list of integers that encodes the message's arguments.

Expressions and commands are defined as functions on state. Define *exp* to be $state \rightarrow Z$, and, as in Section 2, define *com* to be $state \rightarrow state$. The commands and expressions used in Figure 1 are defined as functions over these types, and Nuprl's display forms are used to give their applications conventional notations. We use a dot notation for the value of a command or expression in a state, such as $e \cdot s$ for (es).

A program is defined as a pair containing a list of commands and an initial condition which is a predicate on state (of type $state \rightarrow P_1$ where P_1 is the type of Nuprl propositions). The initial condition must at least require that the history start out empty. In our model, a command is enabled if it changes the state when applied. Thus commands whose guards are true but do not change the state are considered disabled. A trace is defined in the usual way as a function of type $N \rightarrow state$. A predicate *trace_of* encodes the restriction that for any n, there is an action such that when applied to state n results in state $n+1$. Temporal operators such as G (always) and F (eventually) are defined as predicates on traces (of type $trace \rightarrow P_1$) using a fairly direct encoding of the definitions in [9]. We then define the notion of a property ψ being valid of a program P in the usual way as $P \models \psi$ iff $\forall tr : trace.trace\_of(P, tr) \rightarrow \psi(tr)$. In [6], the automation that we developed for the verification of protocols in Nuprl was discussed in detail. In the present work, we draw mainly on the machinery for rewriting, which draws on a large body of equality theorems for protocols. We will see some examples in Sect. 7.

4 Overview of Data Abstraction

In this section, we give an overview of the form of data abstraction used in our case study. Suppose we are given a program P and a property ψ of traces of the program, and we want to verify whether $P \models \psi$, *i.e.* whether all traces of P satisfy ψ. Suppose P contains a variable v that can take on an arbitrarily large number of data-values. We may be able to perform "data-value abstraction" on v to create an abstract program P' and an abstract property ψ' such that $P' \models \psi' \Leftrightarrow P \models \psi$ and such that v takes on values from a smaller set during execution of P'.

We first discuss how to compute an abstract program from a concrete program, and then discuss some sufficient conditions, that can be checked statically, under which this abstraction is safe.

In our example program in Fig. 1, the data that we are particularly interested in and whose values we want to abstract is the value that gets assigned to the identifier *sent*. The flow of this value through the program execution is important for proving both the data delivery property and the order preservation property mentioned in the figure. We formalize this flow as a set of identifiers that are affected by the value of *sent*. We must also consider communication via message buffers. To take this into account, we define a *message reference* to be a pair $\langle T, i\rangle$ where T is a message type, and i is natural number denoting a position in the list of arguments to a message. A *data reference* of a program is either a program variable or a message reference. For example, the value of *sent* gets passed via the message reference $\langle qsend, 0\rangle$.

From now on, we will use $\mathbf{d}$ to denote the set of all data references possibly affected by the values of the variable(s) being abstracted. In our example, we have

$$\mathbf{d} = \{sent, \langle qsend, 0\rangle, data, \langle rsend, 0\rangle, rdata, rcvd\}.$$

Clearly it is often possible to compute a suitable $\mathbf{d}$, but we have not done this in our case study and so we do not elaborate on it.

Our Nuprl development is paramterized with respect to the *abstraction function* φ, also called a *collapsing function*, that will map the values taken on by the data references in d to a small finite domain.

In our running example, in order to verify the data delivery property in Fig. 1 we will abstract the data values to a two valued abstract domain. For instance, we can use the functions φ_y, parameterized by the y of our data delivery property, defined by $\varphi_y(n) := (\mathit{if}\ \ n = y\ \ \mathit{then}\ \ y\ \ \mathit{else}\ \ -1)$. Using this function corresponds to tracking the delivery of the data-value y. The value -1 represents all other concrete values.

The abstraction function φ_y is parameterized by y, so we would need to generate a new abstract program for every y of interest. However, note that in our example, the only processes to assign to *sent* are the processes s_d, and that the possible assigned values are $\{0 \dots n-1\}$. The protocol is symmetric on these values: if we apply a permutation of theses values to the right-hand-sides of all assignments of constants to *sent*, then we get the same protocol. Because of this symmetry, checking the data-delivery property for an arbitrary y in $\{0 \dots n-1\}$ is the same as checking it for $y = 0$.

We can compute the abstract program P', given ϕ, as follows. First compute $\mathbf{d}$, and then, for all $u \in \mathbf{d}$, replace any constraint $u = n$ in the initial condition of P, where n is a constant, by $u = \varphi(n)$, and replace any assignment $u := expr$ (and any check $u = expr$) in any action of P by $u := \varphi(expr)$ ($u = \varphi(expr)$). If we know that data values are only passed around and not manipulated in $\mathbf{d}$ (and if we know that the property ψ we want to verify satisfies certain properties) then we are guaranteed that our data-value abstraction preserves enough information to verify property ψ.

We make this abstraction method more precise as follows. Suppose that: $D = \{0 \dots d-1\}$, $m \in D$, $\mathbf{d}$ is a set of data references, $\phi \in \mathit{int} \to D$, P and P' are programs, and ψ is a predicate on traces. We first describe how to lift φ (the abstraction function) to states and traces. In particular, we define a function on states, denoted $\gamma^{\varphi}_{\mathbf{d}}$, where φ is the function to be lifted and $\mathbf{d}$ is a set of data references. The collapsed state $\gamma^{\varphi}_{\mathbf{d}}(s)$ is obtained from state s by mapping the value of each program variable x in $\mathbf{d}$ to $\varphi(x.s)$ and applying φ to each value t such that there is a message reference $\langle T, i\rangle$ in $\mathbf{d}$ and t is the i^{th} argument to a message of type T in the history component of s. We will often just write γ when

φ and $\mathbf{d}$ are obvious from context. Traces being infinite sequences of states, we define $(\gamma^{\varphi}_{\mathbf{d}}(tr)).i = \gamma^{\varphi}_{\mathbf{d}}(tr.i)$ where $tr.i$ denotes the i^{th} state in the trace tr, for any natural number i. Note that we overload $\gamma^{\varphi}_{\mathbf{d}}$. Let $trace(P)$ denote the set of all valid traces of program P.

Conditions for P' to be a correct abstraction of P are as follows.

1. $\forall tr.(tr \in trace(P)) \Leftrightarrow (\gamma^{\varphi}_{\mathbf{d}}(tr) \in trace(P'))$
2. $\forall tr.\ \gamma^{\varphi}_{\mathbf{d}}(tr) \models \psi(m) \Leftrightarrow tr \models \psi(m)$
3. For all permutations f of D, $\forall tr.(tr \in trace(P)) \Leftrightarrow (\gamma^{f}_{\mathbf{d}}(tr) \in trace(P))$
4. For all permutations f of D, and for all $m \in D$, $\forall tr.\ tr \models \psi(m) \Leftrightarrow \gamma^{f}_{\mathbf{d}}(tr) \models \psi(f(m))$

If the above conditions hold true then $P' \models \psi(m)$ iff $P \models \forall y \in D.\ \psi(y)$.

Our Nuprl proof captures sufficient conditions for (1) and (3) to hold. The extracted program will check that these conditions hold for a given P (P' will be a function of P). We have not formalized the syntax of temporal logic, so our program does not check any sufficient conditions for ψ. In particular, conditions (2) and (4) are proved by hand on a case-by-case basis.

A generic sufficient condition for condition (1) is that the control flow of program P is completely independent of the values of the data references in $\mathbf{d}$. For example, there can be no conditional branching on the value of variables in $\mathbf{d}$. Additionally, the initial condition of the program and the guards of the program actions must be independent of the values of the data references in $\mathbf{d}$. As we will see in Sect. 7, this property can be defined formally as a static check on programs.

A sufficient condition for (3) is similar to the one for (1), except that we additionally require that in any action a containing an assignment $x := n$, where $n \in D$ and x is in $\mathbf{d}$, all assignments in a of constants to members of $\mathbf{d}$ have n as the right-hand side, and, furthermore, for every other $k \in D$, there is another action a' such that a' is the result of replacing in a each assignment of the form $z := n$ by $z := k$.

5 Defining Representability in Nuprl

In this section, we define representability of commands which, as mentioned, allows us to reason semantically about data abstraction, while implicitly constructing a program that operates on syntax. In the interests of compactness, in this section, as well as later sections, we will usually use a more mathematical style of presentation instead of giving exactly what would would appear in the theorem prover. The differences in presentation are minor notational ones.

To talk about the representability of commands, we also need to define the representability of expressions. In both cases, we parameterize by a state invariant, since ultimately we will only want a program and its representation to be equivalent on certain states. In our case, we only need to consider states collapsed by γ. We define equality up to invariant I of functions on states (such as expressions and commands), written as $=_I$, as equality of values on all states satisfying I.

Representability of expressions, denoted $R_I(e)$ or $R[I](e)$, is inductively defined below. We omit the types of bound variables when they are clear from context. $R_I(e)$ is true iff

$$\begin{aligned}&e =_I \mathit{false} \lor e =_I \mathit{true} \lor [\exists n : \mathcal{Z}.e =_I n] \lor [\exists x : id.e =_I x]\\ &\lor [\exists b, e_1, e_2.(R_I(b) \land R_I(e_1) \land R_I(e_2)) \land e =_I (\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2)]\\ &\lor [\exists e_1, e_2.R_I(e_1) \land R_I(e_2) \land (e =_I (e_1 + e_2) \lor e =_I (e_1 - e_2) \lor e =_I (e_1 = e_2)\\ &\qquad\qquad\qquad\qquad \lor e =_I (e_1 \lor e_2) \lor e =_I (e_1 \land e_2))]\\ &\lor [\exists e'.R_I(e') \land e =_I \neg e']\end{aligned}$$

We use several abbreviations here. For example, n in the equality $e =_I n$ denotes $\lambda s.n$ and x in the equality $e =_I x$ denotes $\lambda s.(x.s)$. Note that we overload the operators in binary expressions. For example $\land$ also denotes the conjunction of Nuprl.

Representability of commands is parameterized by an invariant, as above, and also by a predicate on commands. Intuitively, $R_{I,Q}(c)$ (also denoted $R[I,Q](c)$) means that c is representable, up to I, in such a way that for each subcommand c', Q is true and c' preserves I. The exact right-hand side in Nuprl of the definition of $R_{I,Q}(c)$ (denoted `rcom[I,Q]` in Nuprl) is the following

```
(c =[I] skip
∨ (∃x:id. ∃e:zexp. rexp[I] e ∧| c =[I] x:=e)
∨ (∃c1,c2:com. (rcom[I,Q] c1 ∧ rcom[I,Q] c2) ∧| c =[I] (c1 ;c2))
∨ (∃b:zexp ∃c1,c2:com
     (rexp[I] b ∧ rcom[I,Q] c1 ∧ rcom[I,Q] c2)
     ∧| c =[I] (if b then c1 else c2))
∨ (∃b:zexp. ∃c':com. (rexp[I] b ∧ rcom[I,Q] c') ∧| c =[I] b --> c')
∨ (∃p:PId ∃d:zexp ∃M:Z. ∃as:zexp List.
     (rexp[I] d ∧ ∀(rexp[I];as)) ∧| c =[I] d!M(as))
∨ (∃p:PId. ∃c':com. ∃M:Z. ∃as:id List.
     rcom[I,Q] c' ∧| c =[I] p?M(as) --> c'))
∧| (Q c ∧ (∀s:state. I s ⇒ I (c s)))
```

The occurrence of $\forall$ applied to two arguments has the meaning that the property (the first argument) holds of every element of the list (the second argument). The operator $\land|$ is an alternate definition of conjunction in Nuprl which roughly makes the right hand side computationally insignificant, so that an extracted program producing a witness for the conjunction will only produce witnessing information for the left hand side. Using such alternate definitions can dramatically improve the computational efficiency of extracted programs.

For representability of programs, in addition to commands, we must represent the initial condition predicate. We choose to represent it as a command that only sets variable values. The initial states are those that result from running this command on a state with an empty history. We overload R again and use $R_{I,Q}(P)$ and $R[I,Q](P)$ to denote representability of programs. A program is representable if the initial state command is representable and each of the actions are representable. We omit its precise definition.

6 Main Results of the Nuprl Formalization

In this section, we discuss the culminating theorems of our formal proof development in Nuprl and illustrate how the program we extract from the formal proofs computes a data-abstracted version of a concrete program as long as the concrete program satisfies the condition that the control flow is independent of the data-values. We first give some additional definitions.

Instead of stating control/data independence explicitly as a requirement on programs, we will prove the theorems in such a way that the extracted program is a partial function that will succeed if the condition is satisfied and will fail otherwise. To do so, we use the possibility operator defined in Section 2.

In addition to lifting φ to states and traces as in Section 4, we also lift it to commands and programs. For commands, we have $\gamma_{\mathbf{d}}^{\varphi}(c) = (\gamma_{\mathbf{d}}^{\varphi} \circ c)$. Thus, applying a collapsed command is the same as applying a command to a state and then collapsing the state. For a program $P \equiv \langle as, I\rangle$ (as is the list of commands, I denotes the initial condition), we have

$$\gamma_{\mathbf{d}}^{\varphi}\langle as, I\rangle = \langle map\ (\gamma_{\mathbf{d}}^{\varphi})\ as,\ \lambda s.\exists s'.(s = \gamma_{\mathbf{d}}^{\varphi}(s') \wedge I(s'))\rangle$$

where *map* is the usual mapping function on lists, and the first occurrence of $\gamma_{\mathbf{d}}^{\varphi}$ on the right hand side denotes the collapsing function for commands, while the second denotes the collapsing function for states. The function $\gamma_{\mathbf{d}}^{\varphi}$ on programs gives us a semantic notion of abstract program, which we call the *pseudo-abstract program*.

There are two main theorems. The first of these is

```
∀drs:dref List. ∀phi:(idempotent).
∀p:prog. ∀psi: { f:trace → P | respects(f;γ[drs;phi]) }.
  rprog p
  ⇒ (∀e:zexp. rexp e ⇒ rexp (phi o e))
  ⇒ ?(rprog[im(γ[drs;phi]),·] (γ[drs;phi] p)
        ∧| (p |= psi ⟺ γ[drs;phi] p |= psi))
```

This theorem says that given a list of data references, an idempotent collapsing function on integers, a representable program, and a temporal property satisfying a certain condition, then possibly the pseudo-abstract program is representable (up to states in the image of the abstraction function) and is equivalent with respect to the property `psi`. The hypothesis about `phi` is a technical detail that is explained later. We have defined specialized display forms for some operators in Nuprl, so, for example, `rprog[I,Q]` displays as just `rprog` in the case that both `I` and `Q` are **λ**`x. True`.

The second theorem is

```
∀drs:dref List. ∀d:N. ∀p:prog.
  rprog p
  ⇒ (∀psi: { f:Z → trace → P | perm_inv(d;drs;f) }
       ?( ↓( ∀y0:Nd. (∀y:Nd. p |= psi y) ⟺ p |= psi y0) ) )
```

This says that if `psi` is a function from integers to temporal properties satisfying a certain permutation invariance property, then possibly for all `y0` in the set $\{0, \ldots, d-1\}$, the program satisfies `psi` at all y iff it satisfies it at `y0`.

We apply our abstraction method to a particular program P (which we assume has been entered into Nuprl as a member of type `prog`) and to a particular function ψ from integers to temporal properties, by doing the following.

1. Prove a theorem that P is representable.
2. Prove that ψ satisfies the condition in the second theorem above, and that $\psi(0)$ satisfies the condition in the first theorem.
3. Run the extraction of the second theorem with arguments $\mathbf{d}$, some natural number d , P and the extracted program from the representability theorem. If the result is of the form $inl(.)$, then the property under the `?` holds and so it suffices to check the program satisfies $\psi(0)$; otherwise halt.
4. Run the extract from the first theorem on appropriate inputs. If the result is of the form $inl(x)$, then x will encode a representation of the abstracted program.

Part (1) has been automated. Part (2) is manual and corresponds to parts (2) and (4) of Section 4. In part (3), the list $\mathbf{d}$ must be entered manually. It would be straightforward to write an ML function to compute such a list given P, but we have not done this. We have implemented a procedure that takes the encoding produced by step (4) and makes it readable. We have proven theorems which condense some of the steps above, and particularize to a particular class of `phi`'s so as to eliminate the condition on `phi` in the first theorem, but we believe the above account is clearer. We have not bothered implementing uninteresting procedures to take ascii representations of protocols to Nuprl representations, nor to completely glue together the steps above.

We have applied our method to an instance of the program in Fig. 1. Let P_0 be the instance with the 3 data values $\{0, 1, 2\}$ and a queue of length 8. We choose the function φ_0, take $d = 3$, and take $\psi(y)$ to be $G(sent = y \Rightarrow F(rcvd = y))$. We also use the six-element set given earlier as as the set $\mathbf{d}$ of data references. Applying the steps above succeeds, and yields the result below. We use the following abbreviation where e is any expression: $F(e) :=$ (**if** $e = 0$ **then** 0 **else** -1).

Initial Condition:

$$\lambda s.\exists s'[(s'.2 = nil) \wedge s = (sent := -1; sent := F(sent); \\ data := F(data); rdata := F(rdata); rcvd := F(rcvd); skip)(s')]$$

(r_1) $true \longrightarrow s!request$
(r_2) $buf[r]?rsend(rdata) \longrightarrow rcvd := (F(rdata)$
(s_2) $buf[s]?request \longrightarrow sent := F(2); q!qsend(F(sent), 0)$
(s_1) $buf[s]?request \longrightarrow sent := F(1); q!qsend(F(sent), 0)$
(s_0) $buf[s]?request \longrightarrow sent := F(0); q!qsend(F(sent), 0)$
(q_8) $buf[q]?qsend(data, i) \longrightarrow$
if $i = 8$ **then** $r!rsend(F(data))$ **else** $i := i + 1; q!qsend(F(data), i)$ **fi**

Because the steps succeeded, it is guaranteed that checking $\forall y \in \{0 \ldots 2\}.\psi(y)$ holds for P_0 is equivalent to checking that $\psi(0)$ holds for the above program. Note that some trivial simplications are possible, for example reducing or eliminating some applications of F, and collapsing the identical actions s_1 and s_2. These simplications would be straightforward to implement, but we have not done so yet.

We wrote a small Nuprl program, given below, to glue together the computational parts for this example. The evaluator for Nuprl programs is a basic call-by-need interpreter, and so is quite slow. We implemented a simple-minded general program optimizer before running the example program. The example terminated in about 5 seconds (on a 400MhZ PC).

Below is a closed Nuprl term whose evaluation produces the representation of the abstracted program.

```
let phi = phi_eg(0) in let psi = λy.psi_eg1(y) in
let p = sqr_inst1 in let p_rep = ext{sqr1_rep} in
let drs = sqr_drs1 in let phi_rep = ext{phi_eg_rep} 0 in let d = 3 in
if isl(ext{poss_data_indep} drs d p p_rep psi)
then let res = ext{abs_thm_2} drs phi p (psi 0) p_rep phi_rep in
     if isl(res) then outl(res) else "No" fi
else "No" fi
```

The expressions `ext{abs_thm_2}` and `ext{poss_data_indep}` name the respective programs extracted from the two theorems discussed in section 6. Recall that both of these programs produce a value in a disjoint union. The program above first uses `poss_data_indep` to test if the example program (bound to `p`, with representation bound to `p_rep`) satisfies the permutation invariance property expressed in the second theorem. If so (*i.e.* if the result is in the left part of the disjoint union), then it runs `abs_thm_2`, testing the result for success using `isl`. In the unsuccessful cases, `"No"` is returned.

We could have typechecked the above program to ensure that we put the pieces together properly, and that the result satisfies the desired properties. Doing this typechecking would have entailed proving the appropriate conditions on `psi`. We just proved these separately.

The output of this program is an explicit piece of data that completely specifies the required abstract program. However, it is rather hard to read, involving numerous injections into disjoint sums, and also junk such as parts of the program's semantics. To help with readability, we implemented a conventional kind of recursive data type in the type theory for representing terms and expressions, and extracted a function that translates to this second representation. From this latter representation, we obtained by inspection the form of the result given in section 6.

7 Some Details of the Nuprl Proofs

Most of the work in the proof is related to conditions (1) and (3) of Section 4 and is independent of the kind of temporal properties being checked. We give details only on the parts related to condition (1). Most of the work related to condition (3) is similar. The work related to condition (1) is divided into three main theorems. These theorems form the bulk of the proof of the first "top-level" theorem given Section 6. We discuss each theorem below, and describe a few example steps of their proofs. In what follows, we are taking the P' in condition (1) to be the pseudo-abstract program $\gamma(P)$.

For all three theorems, we assume that P is some program, $\mathbf{d}$ a set of data references, and φ an idempotent function on integers. The idempotence requirement is necessary to show that certain commands satisfy the homomorphism property discussed below.

The first theorem says that condition (1) holds assuming that for all commands c in P, $(\gamma \circ c) = (\gamma \circ (c \circ \gamma))$. We call this latter property the *homomorphism property* on commands, and denote it as $hom_\gamma(c)$. The reason that we consider this property is that it is a simple semantic sufficient condition for the control flow of P to be independent of the data references in $\mathbf{d}$.

Let ψ be a temporal property, *i.e.* a predicate on traces. Define $respects(\psi, F)$ iff $\forall tr.\psi(tr) \Leftrightarrow \psi(F(tr))$ holds. Our first theorem is the following.

Theorem 1. *Suppose* $R[True, hom_{\gamma_{\mathbf{d}}^{\varphi}}](P)$ *holds, and let* ψ *be a temporal property such that* $respects(\psi, \gamma_{\mathbf{d}}^{\varphi})$. *Then* $P \models \psi \Leftrightarrow \gamma_{\mathbf{d}}^{\varphi}(P) \models \psi$.

The second theorem embodies a check of a syntactic sufficient condition for the condition $R[True, hom_{\gamma_{\mathbf{d}}^{\varphi}}](P)$ of 1.

Theorem 2. *If* $R[True, True](P)$ *then* $?R[True, hom_{\gamma_{\mathbf{d}}^{\varphi}}](P)$.

We prove this theorem by induction on representability, considering a case $?R[True, hom_\gamma](c)$ for each type of command c, and then proving the property $R[True, hom_\gamma](c)$ for the commands where it can be seen to hold according to our sufficient condition. If we were using an explicit approach to syntax, this inductive argument would correspond to a proof by induction over syntax trees that (possibly) the homomorphism property holds of the meaning of a tree and all of its subtrees.

One of the base cases of the induction is when $c = (x := e)$. In this case, we do a case analysis on $x \in \mathbf{d}$. In the case $x \notin \mathbf{d}$, we use a lemma whose conclusion is $?(e = (e \circ \gamma))$, which says that possibly e's value is independent of γ. The lemma is proved by induction on the representability of e. In the case $x \in \mathbf{d}$, we do a case analysis on the representation of e. In the cases where e is a constant or a variable, we know that $R[True, hom_\gamma](x := e)$. In the other cases, we prove $?R[True, hom_\gamma](x := e)$ the trivial way, by introducing the right disjunct of the definition of ?. The hardest part of the proof of Theorem 2 is the cases for the commands for sending and receiving messages, where we have to make a correspondence between message data references and the argument list of the command.

To obtain a program that computes abstracted specifications, we must show that the pseudo-abstract program $\gamma_{\mathbf{d}}^{\varphi}(P)$ is representable. This involves showing that its initial condition can be expressed as a property on states, and that each of the commands $(\gamma \circ c)$ of $\gamma(P)$ can be represented as a command in the guarded command language. The program representing $\gamma_{\mathbf{d}}^{\varphi}(P)$ need only be equivalent to $\gamma_{\mathbf{d}}^{\varphi}(P)$ on collapsed states, that is, states in the image of the function γ. We express this notion formally via a predicate on states, denoted im_γ, defined by $im_\gamma(s)$ iff $\exists s' : state.\ s = \gamma(s')$. We need the additional condition on φ that for any expression that is representable, $(\varphi \circ e)$ is also representable.

Theorem 3. *Suppose that for any expression e, $R[True, True](e)$ implies $R[True, True](\varphi \circ e)$. If $R[True, hom_{\gamma_{\mathbf{d}}^{\varphi}}](P)$, then $R[im_{\gamma_{\mathbf{d}}^{\varphi}}, True](\gamma_{\mathbf{d}}^{\varphi}(P))$.*

The proof is by induction on $R[True, hom_{\gamma_{\mathbf{d}}^{\varphi}}](P)$. For the case $c = (x := e)$, if $x \in \mathbf{d}$, then we use the fact that $(\gamma \circ c) = (\gamma \circ (c \circ \gamma))$ to show that $(\gamma \circ (x := e))$ is equivalent to $x := (\varphi \circ e)$. Because of the assumption on ϕ, we know that $x := (\varphi \circ e)$ is representable.

8 Conclusion

Using the example of data-value abstraction, we have verified the correctness of an abstraction method for specifications satisfying a particular sufficient condition on their syntax. We have exploited the constructivity of Nuprl to extract a program which can compute the abstract specification corresponding to any concrete specification satisfying the sufficient condition. Most importantly, we were able to do so using only semantic reasoning.

It is unlikely that the approach of dealing with syntax implicitly will always be preferable. This is not a problem, since it can easily coexist with the explicit approach. For example, we could define a conventional recursive type of abstract syntax trees, write a meaning function, and prove that for every representable program there is a tree whose meaning is the program.

Our work was complicated somewhat by our choice of protocol language and its formalization. In particular, since commands are functions on states, instead of relations, non-deterministic commands cannot be represented. With non-determinism, one can include a command that non-deterministically chooses one from an indexed set of commands — this would have been a more natural choice for the actions s_d, and would have obviated the need, in the sufficient condition for symmetry, for finding actions that are similar up to constants on right-hand sides of assignments. Another complication due to the protocol language is its lack of types. This language was developed inside Nuprl to support reasoning about particular protocols, and not for metareasoning about programs.

The precise form of inductive definition mechanism implemented in standard Nuprl [4] is not valid in the classical extension [8] used here. It is not too difficult to adapt it, but we have not done so yet and hence have simply axiomatized the two inductive definitions we needed.

We believe that our results can be extended to deal with temporal properties in the same way as programs. One difficulty is dealing with binding expressions such as universal quantification. It might be possible to deal with universal quantification by using a program variable in place of the quantified variable. We should also be able to extend the results to data-path abstraction, for example collapsing the queue in our example.

It should be possible to use our techniques in other theorem provers based on constructive type theory. Classical theorem provers could also formalize the same notion of representability, but it would likely be much less useful, since representability would not encode syntax, and theorems whose conclusion is an application of the possibility operator would be vacuous.

Acknowledgements. The authors would like to thank R. P. Kurshan for suggesting the example data-abstraction problem and for useful discussions. The third author would also like to thank Bell Labs for providing the opportunity to work on this problem through a summer internship.

References

1. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In *International Conference on Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.
2. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. 19th Ann. ACM Symp. on Principles of Prog. Lang.*, Jan. 1992.
3. R. L. Constable. A note on complexity measures for inductive classes in constructive type theory. *Information and Computation*, 143(2):137–153, 1998.
4. R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
5. J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. *Proceedings of the Seventh International Conference on Computer Aided Verification (CAV '95),* Vol. 939 of *Lecture Notes in Computer Science*, pages 54–69, 1995.
6. A. P. Felty, D. J. Howe, and F. A. Stomp. Protocol verification in Nuprl. In *Tenth International Conference on Computer Aided Verification*, pages 428–439. Springer-Verlag Lecture Notes in Computer Science, June 1998.
7. K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Proceedings of Formal Methods in Europe (FME)*, 1996.
8. D. J. Howe. Semantics foundations for embedding HOL in Nuprl. In *Algebraic Methodology and Software Technology*, pages 85–101. Springer-Verlag Lecture Notes in Computer Science, 1996.
9. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991.
10. O. Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technische Universität München, 1998.
11. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *Proceedings of the Seventh International Conference on Computer Aided Verification (CAV '95),* Vol. 939 of *Lecture Notes in Computer Science*, pages 84–97. Springer-Verlag, 1995.
12. C. Sprenger. A Verified Model Checker for the Modal μ-calculus in Coq. *Proceedings of TACAS98*, pages 167–182, 1998.
13. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th Ann. ACM Symp. on Principles of Prog. Lang.*, Jan. 1986.