A Monad-based Modeling and Verification Toolbox with Application to Security Protocols*

Christoph Sprenger and David Basin

Department of Computer Science, ETH Zurich, Switzerland
{sprenger,basin}@inf.ethz.ch

Abstract. We present an advanced modeling and verification toolbox for functional programs with state and exceptions. The toolbox integrates an extensible, monad-based, component model, a monad-based Hoare logic and weakest precondition calculus, and proof systems for temporal logic and bisimilarity. It is implemented in Isabelle/HOL using shallow embeddings and incorporates as much modeling and reasoning power as possible from Isabelle/HOL. We have validated the toolbox's usefulness in a substantial security protocol verification project.

1 Introduction

The choice of a specification formalism with supporting verification methods and tools is critical to the success of substantial verification projects. In order to obtain manageable proofs, models and their properties must be formulated at an appropriate level of abstraction and verification methods and tools must be available for reasoning at that abstraction level. In this paper, we present a comprehensive formalism providing such abstractions for the specification and verification of functional programs with non-pure features, namely state and exceptions. It consists of the following elements: (1) a modeling language capturing the computational structure of the systems under study, (2) a notion of component for structuring models, (3) a notion of component abstraction, including a proof method to establish an abstraction relation between two components, and (4) property specification languages and proof methods. We have implemented this toolbox in Isabelle/HOL [18]. Our aim is to provide a set of practically useful tools rather than to develop meta-theoretical studies about programming languages and logics. Therefore, we work with shallow embeddings and exploit Isabelle/HOL's expressive specification language (for 1–2) and powerful reasoning tools (for 3–4) as much as possible. We now discuss (1)–(4) in turn.

Ad 1. Modeling state and exception handling directly in HOL results in models that are cluttered with additional state parameters and error-handling branches. This situation calls for an additional layer of abstraction, which can be elegantly modeled using monads [15]. In our case, the monad of choice is a deterministic state-exception monad, which extends HOL with operations for sequential composition, state manipulation, and exception handling. All other control structures, such as pattern matching, if-then-else

* This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.

and function definitions, are borrowed directly from Isabelle/HOL. We would like to stress that most of our method and tools can be easily adapted to other monads.

Ad 2. We take a simple view of components as modules encapsulating a state with interface functions that manipulate this state. We represent the state in an object-oriented style as an extensible record and component composition as record extension and function composition. Using extensible records allows the extended component to automatically inherit properties proved for the base component.

Ad 3. We compare (abstract or refine) components using bisimilarity. Two components are bisimilar if all pairs of equally named interface functions are bisimilar. To show this, we have developed a compositional proof system to establish the bisimilarity of programs. Its judgments and rules resemble those of Hoare logic except that pre- and post-conditions are relations between pairs of states of two programs.

Ad 4. In order to state and prove properties about components, we have embedded several logics in Isabelle/HOL: a weakest pre-condition (WP) calculus, a Hoare logic, and linear-time temporal logic (LTL). All these logics rely on HOL's set theory as their underlying assertion language. The WP calculus is derived from Pitts' evaluation logic [20] and provides the basis for the definition of the Hoare logic. Both of these are tailored to the state-exception monad, whereas the temporal logic is interpreted over standard transition systems. The benefit of embedding these logics in Isabelle/HOL is that this enables the use of the corresponding, well-established proof methods. We reduce the verification of temporal properties to pre-/post-condition and assertional reasoning using standard proof rules [14]. The resulting Hoare triples are proved using a combination of Hoare logic and the underlying WP calculus. While Hoare logic provides the full flexibility of interactive proofs, we may unfold Hoare triples at any point during a proof and invoke the WP calculus, using Isabelle's efficient simplifier to automatically reduce the resulting weakest pre-conditions into simpler HOL assertions.

Our contribution is two-fold. First, we show how to leverage the rich modeling and reasoning infrastructure provided by a theorem prover like Isabelle/HOL for semantic embeddings of various modeling, specification, and verification concepts. This allows us to model, specify and reason at an adequate level of abstraction. We adapt and integrate well-known techniques with less standard and new ones to build a unique, comprehensive, and modular modeling and verification toolbox. Second, we provide a ready-to-use verification toolbox with a wide scope of application. This toolbox has proved its effectiveness in a substantial cryptographic protocol verification project [21]. For a rough idea of the size of the theories involved, our development spans more than 22k lines of Isabelle/HOL sources, organized into more than 50 theories. The toolbox accounts for about 20% of these figures. In particular, the combination of Hoare logic and the WP calculus offers a good degree of proof automation, which was crucial to the successful completion of the sizable verification effort involved. We use examples from our project [21] to illustrate the application of the concepts presented in this paper.

2 Background

Isabelle/HOL notation In Isabelle/HOL, $t :: T$ denotes a term t of type T. The expression $c\ x \equiv t$ defines the constant c with the parameter x as the term t. Definitions

constitute the principal mechanism for producing conservative extensions of HOL. Type variables are denoted by lowercase Greek letters. Given types α and β, $\alpha \Rightarrow \beta$ is the type of (total) functions from α to β, $\alpha \times \beta$ is the product type, and α *set* is the type of sets of elements of type α. The type *unit* contains a single element. There are several mechanisms to define new types. A datatype declaration introduces an inductive data type. For example, the polymorphic option type is defined by datatype α *option* = *None* | *Some* α. Functions of type $\alpha \Rightarrow \beta$ *option* are used to model partial functions from α to β. The declaration types $T_1 = T_2$ merely introduces a new name for the type T_2, possibly with parameters, as in types $\alpha \rightharpoonup \beta = \alpha \Rightarrow \beta$ *option*.

Application: Cryptographically Sound Protocol Verification In our security protocol verification project [21], we work with the protocol model proposed by Backes, Pfitzmann, and Waidner [2], henceforth called the *BPW model*. This model is substantially more complex than standard Dolev-Yao models (e.g., [19]), but has the advantage of being cryptographically sound, which means that properties proven of protocols hold with respect to standard cryptographic definitions of security (defined in probabilistic and complexity-theoretic terms). Roughly speaking, the BPW model can be viewed as a centralized cryptographic library component: it has an encapsulated state, where it tracks which principals know which messages, and a collection of interface functions for constructing, decomposing, and transmitting messages. Principals (and the attacker) use the interface functions when executing (respectively attacking) protocols. We model not only the BPW model, but also protocols defined on top of it, as components. We have constructed a second formalization of the BPW model, which abstracts the representation of protocol messages from DAGs to inductive terms, and we have proved its bisimulation equivalence with the first formalization. Finally, we have modeled the Needham-Schroeder-Lowe authentication protocol [12] on top of the second formalization and proved various security (secrecy and authentication) properties.

We refer the interested reader to [21] for more details about this application as well as for a detailed comparison with related work on security protocol verification.

3 Monads and Components

In this section, we describe a monad-based hierarchical component model. Components are formalized as an encapsulated state manipulated by a set of interface functions. Our formalization of components is extensible, whereby properties proved for a given component remain valid for all extensions.

3.1 State-exception monad

Monads are a concept from category theory introduced in programming language semantics by Moggi [15] to model different computational phenomena in a functional setting, including exceptions, state, non-determinism, and input/output. From a programming perspective, a monad is a type constructor with a unit and a binding operation enjoying unit and associativity properties. In our case, we are working with a state-exception monad with just a single type of exception:

```
datatype α result = Exception | Value α                    −− result type
types (α, σ) S = σ ⇒ α result × σ                          −− monad type

return :: α ⇒ (α, σ) S                                     −− monad unit
return a ≡ λs. (Value a, s)

bind :: (α, σ) S ⇒ (α ⇒ (β, σ) S) ⇒ (β, σ) S               −− monad bind
bind m k ≡ λs. let (a, t) = m s in
                 case a of Exception ⇒ (Exception, t) | Value x ⇒ k x t
```

A term of type $(\alpha,\ \sigma)\ S$ is called a *computation*. Computations act on a state of type σ and either result in an exception or a result value of type α. The monad unit, which we call return, embeds a value into a computation. Binding acts as sequential composition with value passing in the style of a let-binding. We write do $x \leftarrow m$; $k\ x$ for *bind* $m\ k$. The monad unit and associativity laws are easily proved in Isabelle/HOL:

```
lemma bind_left_unit: do x ← return a; k x = k a
lemma bind_right_unit: do x ← m; return x = m
lemma bind_assoc: do y ← (do x ← m; k x); h y = do x ← m; (do y ← k x; h y)
```

The monad-specific operations are concerned with state transformations and exception handling. State extraction and update are handled by the functions *distill* and *xform*, respectively.

```
distill :: (σ ⇒ α) ⇒ (α, σ) S                              −− return transformed state
distill f ≡ λs. return (f s) s

xform :: (σ ⇒ σ) ⇒ (unit, σ) S                             −− set transformed state as new state
xform f ≡ do t ← distill f; λs. return () t
```

Exception handling is achieved by the functions throw and try_catch, which are duals of return and *bind* with respect to the type of result.

```
throw :: unit ⇒ (α, σ) S
throw x ≡ λs. (Exception, s)

try_catch :: (α, σ) S ⇒ (α, σ) S ⇒ (α, σ) S
try_catch k h ≡ λs. let (r, t) = k s in
                      case r of Exception ⇒ h t | Value b ⇒ (Value b, t)
```

We write try k catch h instead of try_catch $k\ h$ and we call h the exception handler. Exception handling can easily be generalized to handle multiple types of exceptions. In this case, the exception would be passed to the exception handler h.

We borrow control structures such as if-then-else and pattern matching (case) from the Isabelle/HOL meta-language. We do not need iteration and recursion in our application, but these can also be borrowed from Isabelle/HOL to a certain extent, which we discuss below. Summarizing, we use the state-exception monad to add a layer of abstraction to Isabelle/HOL that is appropriate to specify our models.

Generality of our approach Ideally, we would like to define monads and the monad laws abstractly and instantiate them to the concrete monads of interest. This could be

achieved by using axiomatic constructor classes in the style of Haskell, but Isabelle's type system only supports the less axiomatic *type* classes. The latter do not support type constructors and the axioms may only contain a single free type variable. Thus, we can only define concrete monads. There are also restrictions concerning the definition of recursive functions. Only terminating functions can be defined and the corresponding Isabelle/HOL packages are pushed to their limits when termination not only depends on the explicit function arguments, but also on the implicit state argument of the monad. One possible solution is to define and prove specialized fixed point theorems as in [11].

A different, more elaborate solution to both problems above is proposed by Huffman, Matthews, and White [6]. They formalize axiomatic constructor classes in Isabelle/HOLCF, an extension of Isabelle/HOL with domain theory. By a clever construction, they are able to represent HOLCF domains in a universal domain and reflect representable domains as values and type constructors as continuous functions. This allows them to define axiomatic constructor classes in terms of axiomatic type classes and apply them to realize an abstract definition of monads. Moreover, in the domain-theoretic context, fixed points can be used to define recursive functions given any continuous functional. However, for our toolbox, we have chosen to live with the restrictions imposed by the simpler, direct formalization of concrete monads in Isabelle/HOL.

3.2 Hierarchical component model

Our components consist of a state that is manipulated by a set of *interface functions*. For reasons that will become clear shortly, we model a state as a record and consider its individual fields to be the state variables. For example, record $point = x :: nat\ \ y :: nat$ defines a record type for points, of which the record $(\!|x{=}1,\ y{=}2|\!)$ is an element.

Example 1 (BPW model). We start by briefly describing the BPW model and refer the reader to [2] for full details. The BPW model constitutes a library of cryptographic operations, which keeps track of, and controls access to, the messages known by each party. The BPW model provides *local functions* for operating on messages and *send functions* for exchanging them between an arbitrary, but fixed, number N of honest users and the adversary (which we identify with the network). The state of our abstract formalization of the BPW model is defined as:

$$\mathsf{record}\ \delta\ mLibState = knowsM :: party \Rightarrow hnd \rightharpoonup \delta\ msg$$

There is a single state variable called *knowsM*, which defines for each party a partial function mapping handles to protocol messages. Here, the type δ of payload data is polymorphic, since it depends on the concrete protocol built on top of the BPW model.

The interface functions refer to messages indirectly using *handles*, which are a form of pointer required for reasons of cryptographic soundness. As an example of an interface function, we show here the function for generating fresh nonces.

$$gen\_nonceM :: party \Rightarrow (hnd,\ (\delta,\ \sigma)\ mLibState\_scheme)\ S$$
$$gen\_nonceM\ u \equiv \mathsf{do}\ n \leftarrow distill\ (\lambda s.\ freshTagM\ s);\ newmsgM\ u\ (mNonce\ n)$$

The function *freshTagM* returns a fresh tag, which has not been used so far, i.e. is not part of any known message. Then *newmsgM u* (*mNonce n*) creates a fresh handle h and updates the knowledge map *knowsM* for user u with the new mapping $h \mapsto mNonce\ n$.

For hierarchical model construction and proofs we rely on record extensibility: record *cpoint* = *point* + c :: *color* extends points with a color field. Behind the scenes, the definition of a record type *r* creates a *record scheme* α *r_scheme*, which extends the declared type *r* with a polymorphic field *more* of type α. The type *r* is derived as *unit r_scheme*. Record extensibility is based on the instantiation of the record scheme parameter α with additional fields, which induces a subtyping relation between record schemes and their instantiations [16]. For example, *point* and *cpoint* are both subtypes of α *point_scheme* (but *cpoint* is not a subtype of *point*).

We exploit record subtyping to construct new components from existing ones. Given an existing component C with a state of type, say, *C_scheme*, we construct a new component D by extending the state of C with additional variables and defining a new set of interface functions. This construction has the advantage that any invariant I (in fact, any property) we have proved for component C still holds when C's interface functions operate on the extended state; since the latter is just an instance of *C_scheme*, there is no need for an explicit proof of this fact. This explains why the function *gen_nonce* in Example 1 operates on the record scheme (δ, σ) *mLibState_scheme* instead of the plain record δ *mLibState*. Moreover, if the component D only uses C's interface functions to manipulate C's part of the state (as sound engineering practice dictates), then a straightforward proof shows that the invariant I lifts to D.

Observe that record extension is linear and neither associative nor commutative. Therefore, this is a hierarchical method rather than a compositional one. Although this is adequate in our case, it may be too restrictive for other applications. Without using record subtyping, we would have to explicitly embed the state of each component into a global system state and prove that properties of individual components can be lifted to the composed system. The benefit of such additional work would be a compositional method, where properties of individual components can be proved separately.

Example 2 (Generic protocols). Security protocols are modeled on top of the BPW model. Each user A runs a protocol component P_A, which maintains its own local state and provides a user and a network handler, which respond to input from the user and the network, respectively. The local state of each user is modeled as an extension of the state of the BPW model with a new polymorphic field *loc* type *user* $\Rightarrow$ σ. The type variable σ will be instantiated by concrete protocols.

```
record (δ, σ) globState = δ mLibState +
  loc :: user ⇒ σ                    −− local state of each protocol machine
```

The protocol handlers invoke the local BPW functions to parse and construct protocol messages. The output of either handler may go to the user or the network (i.e., the adversary). A protocol component is a record type, whose fields are the two protocol handlers. A protocol is a function assigning a protocol component to each honest user. Concrete protocols inhabit instances of the type (ι, o, δ, σ) *protocol*, where ι and o are the types of user input and output.

```
datatype o proto_out = pToUser o | pToNet netmsg

record (ι, o, δ, σ) proto_component =
  proto_user_handler :: ι ⇒ (o proto_out, (δ, σ) globState) S
```

```
proto_net_handler :: user ⇒ hnd ⇒ (o proto_out, (δ, σ) globState) S

types (ι, o, δ, σ) protocol = user ⇒ (ι, o, δ, σ) proto_component
```

The complete protocol system provides (1) local BPW model functions used by the adversary to parse and construct messages and (2) system-level user and network handlers, which are parametrized by the protocol. These handlers connect the protocol-level handlers to the BPW-model send functions in order to exchange messages with the adversary. A message transfer corresponds to updating the recipient's knowledge map.

Example 3 (NSL protocol). We now give an instance of the above protocol type: the Needham-Schroeder-Lowe (NSL) protocol [12], a well-studied, three-step protocol for mutual authentication. In our setting, the local state of each protocol component consists of a variable *nonces*, which is a function mapping each user to a set of nonce handles that were created in protocol sessions with that user. Below, we show the protocol user handler in some detail, but we only sketch the protocol network handler.

```
record ustate = nonces :: user ⇒ hnd set

NeedhamSchroederLowe :: (user, user, user, ustate) protocol
NeedhamSchroederLowe A ≡ (|
  proto_user_handler = λB.                    −− construct the first NSL message
    do na ← gen_nonce (User A); do z ← add_nonce A B na;
    do nameA ← store (User A) A; do m ← pair (User A) (na, nameA)
    do em ← encrypt (User A) (pke (User A) B) m;
    return (pToNet (A, B, em))

  proto_net_handler = λB m.                   −− respond to incoming NSL messages
    do pm ← parse_msg A B m;
    case pm of
      msg1 nb nameB ⇒mk_msg2 A B nb
    | msg2 na nb nameB ⇒mk_msg3 A B nb
    | msg3 nb ⇒ return (pToUser B)
|)
```

The protocol user handler for user A initiates the protocol with another user B by constructing, stepwise, the first protocol message (whose form is $\{N_A, A\}_{K(B)}$, i.e., the public-key encryption of a nonce and user name) using the interface functions of the BPW model. The auxiliary function *add_nonce* adds the freshly generated nonce handle *na* to the set of nonces that A has created in runs with B (stored in variable *nonces*). The protocol network handler receives a message (at handle *m*), parses it, and, depending on the result, reacts by constructing a reply message (*mk_msg2* or *mk_msg3*) or returning to the user at the end of the protocol run (the third case).

Component behavior We give an operational semantics to components in terms of a derived transition system. A transition system `σ trsys` is a record with two fields: the set *init* of initial states of type `σ set` and the transition relation *trans* :: $(\sigma \times \sigma)$ *set*. A computation *m* gives rise to a transition relation *tr m* $\equiv \{(s, snd\ (m\ s)) \mid s.\ True\}$. The transition relation of a component is the union of the transition relations of its

interface functions. By adding a set of initial states, we obtain the transition system associated with a component. The runs of a transition system are defined as infinite sequences of states, $\sigma\ run = nat \Rightarrow \sigma$, starting in an initial state and where successive states are related by transitions. The runs will serve as models for LTL formulas.

4 Program and Temporal Logics

In this section, we describe the weakest pre-condition calculus and the Hoare logic, which we use for pre-/post-condition reasoning, and the linear-time temporal logic used for temporal reasoning. All these logics use HOL sets (of states) as their basic assertions and set operations as boolean connectives. Hence, when we say that a state s satisfies a basic assertion P, we mean $s \in P$. The WP calculus and the Hoare logic are both tailored to our state-exception monad, whereas the temporal logic is interpreted over standard transition systems and linked to the other logics via gluing lemmas. We first present the logics and then illustrate their combined application in Section 4.4.

4.1 Weakest pre-condition calculus

Our weakest pre-condition calculus is inspired by Pitts' evaluation logic [20], which generalizes first-order dynamic logic by interpreting it over monads. We present a semantic embedding of a variant of evaluation logic, instantiated to the state-exception monad. We have two box modalities: $[x \leftarrow m]Q\,x$ for normal termination and $[\,@\,m]Q$ for exceptional termination. These correspond to the weakest pre-condition of the computation m with respect to the post-condition Q. We do not need the dual diamond modalities for our purposes (these could be defined directly or using set complement).

$[x \leftarrow m]Q\,x \equiv \{s.\ \forall x\ t.\ (m\,s) = (Value\ x,\ t) \longrightarrow t \in Q\,x\}$
$[\,@\,m]Q \equiv \{s.\ \forall t.\ (m\,s) = (Exception,\ t) \longrightarrow t \in Q\}$

Note the dependence of the assertion Q on the result value x in $[x \leftarrow m]Q\,x$. The following general properties of the WP calculus follow directly from the definition.

lemma *BoxN_true*: $[x \leftarrow p]\ true = true$
lemma *BoxN_conj*: $[x \leftarrow p]((P\,x) \cap (Q\,x)) = [x \leftarrow p](P\,x) \cap [x \leftarrow p](Q\,x)$
lemma *BoxN_monotone*: $\forall x.\ P\,x \subseteq Q\,x \Longrightarrow [x \leftarrow p]\ P\,x \subseteq [x \leftarrow p]\ Q\,x$

Analogous results can be shown for the exception modality $[\,@\,p]$. We also prove properties related to the monad operations. For the basic monad operations we have:

lemma *BoxN_return*: $[x \leftarrow \mathsf{return}\ a]\ Q\,x = Q\,a$
lemma *BoxE_return*: $[@\ \mathsf{return}\ a]\ P = true$

lemma *BoxN_bind*: $[y \leftarrow (\mathsf{do}\ x \leftarrow p; q\,x)]\ Q\,y = [x \leftarrow p][y \leftarrow q\,x]\ Q\,y$
lemma *BoxE_bind*: $[@\ \mathsf{do}\ x \leftarrow p; q\,x]\ Q = [x \leftarrow p][@\ q\,x]\ Q \cap [@\ p]\ Q$

The first lemma reflects the fact that the return value a is bound to x and the state remains unchanged. The second lemma expresses that return never produces an exception. The third lemma decomposes the modality for the binding operator into two modalities. The fourth lemma states that the weakest pre-condition of $\mathsf{do}\ x \leftarrow p; q\,x$ with respect to

exceptions and Q is composed of two conditions: (1) if p terminates with value x and $(q\ x)$ results in an exception then the resulting state satisfies Q, and (2) whenever p directly produces an exception then the resulting state also satisfies Q.

Similar equations hold for the exception operations, where throw and try_catch are dual (with respect to the type of result) to the cases for return and *bind*, respectively.

lemma *BoxN_throw*: $[z \leftarrow \mathsf{throw}\ ()](P\ z) = \mathit{true}$
lemma *BoxE_throw*: $[@\ \mathsf{throw}\ ()]P = P$

lemma *BoxN_try_catch*: $[x \leftarrow \mathsf{try}\ p\ \mathsf{catch}\ q](P\ x) = [x \leftarrow p](P\ x) \cap [@\ p][x \leftarrow q](P\ x)$
lemma *BoxE_try_catch*: $[@\ \mathsf{try}\ p\ \mathsf{catch}\ q]Q = [@\ p][@\ q]Q$

The weakest pre-conditions of the state-manipulation operations satisfy the following equations. The cases for exceptions equal *true* and are therefore not shown.

lemma *BoxN_distill*: $[x \leftarrow \mathit{distill}\ f](P\ x) = \{s.\ s \in P\ (f\ s)\}$
lemma *BoxN_xform*: $[z \leftarrow \mathit{xform}\ f](P\ z) = f^{-1}\ (P\ ())$

The first equation describes that *distill* f returns the result of applying f to the current state, but does not modify the state itself. The second equation states the weakest pre-condition for state update consists of exactly those states that are elements of $P\ ()$ under f, where f^{-1} denotes the inverse image of f.

4.2 Hoare logic

We construct our Hoare logic on top of the WP calculus. This is reflected in the following definitions of Hoare triples:

$$\{P\}\ m\ \{>Q\} \equiv P \subseteq [x \leftarrow m](Q\ x) \qquad \{P\}\ m\ \{@\ Q\} \equiv P \subseteq [@\ m]Q$$

Again, we have one type of triple for each kind of termination, where the assertion Q in $\{P\}\ m\ \{>Q\}$ depends on the value returned by computation m.

By defining the Hoare triples in terms of the WP calculus, we can switch from Hoare logic to the WP calculus at any point simply by unfolding the definition of Hoare triples. Typically, we start non-trivial proofs using the rules of Hoare logic. Later, we switch to the WP calculus and automatically rewrite the resulting goals into assertions of HOL's set theory, using the equations of Section 4.1. The resulting assertions are often automatically solved using standard reasoning tools provided by Isabelle/HOL.

Next, we present the set of proof rules of our Hoare logic. In a deep embedding of Hoare logic in HOL, we would inductively define a derivability relation (and a program syntax) as in [17]. Here, we are working with a shallow embedding and therefore we represent proof rules semantically as Isabelle/HOL rules. Deriving these rules in Isabelle corresponds to proving their soundness. The two rules for return follow directly from the WP calculus by unfolding the definitions of Hoare triples.

$$\frac{\cdot}{\{P\ x\}\ \mathsf{return}\ x\ \{>P\}}\ returnN \qquad \frac{\cdot}{\{P\}\ \mathsf{return}\ x\ \{@\ Q\}}\ returnE$$

The first rule expresses that the state and the value x are invariant in return x. The second rule captures the fact that the return statement never produces an exception and therefore vacuously establishes an arbitrary post-condition (including *false*).

The two rules for *bind* are proved using the lemma *BoxN_monotone* above.

$$\frac{\{P\}\ p\ \{>\ Q\} \qquad \bigwedge x.\ \{Q\ x\}\ q\ x\ \{>\ R\}}{\{P\}\ \mathsf{do}\ x \leftarrow p;\ q\ x\ \{>\ R\}}\ bindN$$

$$\frac{\{P\}\ p\ \{@\ R\} \qquad \{P\}\ p\ \{>\ Q\} \qquad \bigwedge x.\ \{Q\ x\}\ q\ x\ \{@\ R\}}{\{P\}\ \mathsf{do}\ x \leftarrow p;\ q\ x\ \{@\ R\}}\ bindE$$

In the last premise of these rules, the symbol $\bigwedge$ denotes the universal quantifier of Isabelle's meta-logic. Thus, the corresponding Hoare triple must hold for all possible inputs to computation q. In the rule *bindE*, we distinguish two cases: the first premise covers the case where computation p terminates with an exception, while the other two deal with the case where p terminates with a value x, but $q\ x$ results in an exception.

The proof rules for the try_catch construct are dual to the rules for *bind* and can be proved using the lemma *BoxE_monotone*:

$$\frac{\{P\}\ p\ \{>\ R\} \qquad \{P\}\ p\ \{@\ Q\} \qquad \{Q\}\ q\ \{>\ R\}}{\{P\}\ \mathsf{try}\ p\ \mathsf{catch}\ q\ \{>\ \ R\}}\ try\_catchN$$

$$\frac{\{P\}\ p\ \{>\ Q\} \qquad \{Q\}\ q\ \{@\ R\}}{\{P\}\ \mathsf{try}\ p\,\mathsf{catch}\ q\ \{@\ R\}}\ try\_catchE$$

The rules for the state manipulation functions *distill* and *xform* directly reflect their WP calculus characterization in the lemmas *BoxN_distill* and *BoxN_xform*.

$$\frac{\cdot}{\{\{t.\ t \in P(f\ t)\}\}\ distill\ f\ \{>\ P\}}\ distillN \qquad \frac{\cdot}{\{f^{-1}(P\ ())\}\ xform\ f\ \{>\ P\}}\ xformN$$

Finally, we have proved various additional rules, such as the consequence rule of Hoare logic, which is used to strengthen preconditions and weaken postconditions.

Following the classical argument (e.g., [23]), we have proved the completeness of our Hoare logic rules relative to the HOL assertion language in Isabelle/HOL. The core of the argument is an induction on computations m, which establishes the derivability of $\{[x \leftarrow m]Q\ x\}\ m\ \{>\ Q\}$, for all Q. Since we are working with a shallow embedding, we have not formalized the program syntax, so we prove each case of the induction separately. Additionally, we must ensure that $[x \leftarrow m]Q\ x$ can be expressed in our assertion language (HOL set theory), which is the case by construction in our setting.

4.3 Linear-Time Temporal Logic

We now briefly sketch our embedding of LTL with past operators in Isabelle/HOL. We have chosen to keep this embedding independent of monads, so that it can be reused independently in different contexts. We interpret LTL formulas over runs and transition systems, in the standard way (see, e.g., [14]). Hence, we restrict ourselves to a summarized account.

We have formalized the syntax of LTL formulas in Isabelle/HOL as an inductive data type σ *ltl* parametrized by a type of states σ. Here we deviated from our principle of working with shallow embeddings. Since LTL does not have binding operators, the

overhead associated with a deep embedding was small and allowed us to prove some meta-theoretical results about LTL by induction on the syntax. We note however that a shallow embedding would have also been sufficient for our work. LTL formulas are interpreted in the standard way at positions of runs: $(r,\ i) \models f$ means that formula f holds at position i of run r. A transition system T satisfies a formula if all of its runs at position 0 do, which is written $T \models f$. A component, in the sense of Section 3.2, satisfies a LTL formula if its derived transition system does.

Proof rules Following the approach of [14], we derive a set of proof rules to reduce temporal reasoning to pre-/post-condition reasoning. The most important rule that we have derived is the following one for establishing an invariant P of a transition system T using an inductive invariant I and a previously proved auxiliary invariant J.

$$\frac{T \models \Box J \qquad (init\ T) \subseteq I \qquad I \cap J \subseteq P \qquad \{I \cap J\}\ (trans\ T)\ \{I\}}{T \models \Box P}\ INV$$

The Hoare triple appearing in this rule is defined over a transition relation.

$$\{P\}\ tran\ \{Q\} \equiv \forall s\ t.\ s \in P \wedge (s,t) \in tran \longrightarrow t:\ Q$$

To enable structured reasoning about components in our Hoare logic, we prove a gluing lemma for each component that reduces Hoare triples about the component's transition relation to a set of Hoare triples over the component's interface functions. Here is an example of such a gluing lemma.

Example 4 (Hoare triples for protocol system). We define a notion of Hoare triple for the global protocol system component and link it to the Hoare triples for the transition relation derived from that component. The definition overloads the notation $\{\_\}\ \_\ \{\_\}$ and uses $\{P\}\ m\ \{\&\ Q\}$ for the conjunction of $\{P\}\ m\ \{>\ \lambda x.\ Q\}$ and $\{P\}\ m\ \{@\ Q\}$.

$$\begin{array}{l}\{P\}\ proto\ \{Q\} \equiv \forall\, h\ im\ nm\ pkh\ skh\ u\ \ldots\ .\\ \quad \{P\}\ sys\_user\_handler\ \ proto\ \ u\ \ im\ \{\&\ Q\} \wedge\\ \quad \{P\}\ sys\_net\_handler\ \ proto\ \ nm\ \{\&\ Q\} \wedge\\ \quad \{P\}\ gen\_nonceM\ Adv\ \{\&\ Q\} \wedge \{P\}\ encryptM\ Adv\ pkh\ h\ \{\&\ Q\} \wedge\\ \quad \{P\}\ decryptM\ Adv\ skh\ \ h\ \ \{\&\ Q\} \wedge \ldots\end{array}$$

lemma *HoareSys_decomposition*: $\{P\}\ (glob\_trans\ proto)\ \{Q\} = \{P\}\ proto\ \{Q\}$

The relation $glob\_trans\ \ proto$ denotes the transition relation derived from the entire protocol system. Triples of the form $\{P\}\ (glob\_trans\ \ proto)\ \{Q\}$ arise as subgoals, when rule INV is applied to prove protocol invariants. Using the above lemma, these are rewritten to a set of Hoare triples over the protocol system interface functions.

4.4 Example: Nonce secrecy invariant for the NSL protocol

To illustrate the use of our reasoning tools, we state and prove an invariant of the NSL protocol: the nonces created in sessions between two honest users remain secret.

$$\begin{array}{l}nonceSecrecy \equiv \{s.\ \forall\, A\ B\ n.\\ \quad n \in Nonces\ s\ A\ B \longrightarrow secret\ \ s\ \ (mNonce\ n)\ \{User\ A,\ User\ B\}\ \}\end{array}$$

theorem *nonceSecrecy_invariant*: $NSLtrsys \models \Box nonceSecrecy$

Here, *Nonces s A B* is the set of nonces that *A* has created in runs with *B* (these are accessible via the handles in the set *nonces s A B*). The predicate *secret s m U* expresses that in state *s* message *m* can be derived by at most the parties $A \in U$ from their knowledge in *knowsM s A*. The theorem states that *nonceSecrecy* is an invariant of the transition system derived from the NSL protocol system ($\Box$ denotes "always" in LTL). The proof proceeds in several stages.

1. Apply rule INV and lemma *HoareSys_decomposition* to reduce the temporal statement to a set of preservation lemmas over the NSL system interface functions.
2. Prove the preservation of the invariant by each BPW model interface function. Here, we automate most proofs using the WP calculus.
3. Lift the previous preservation results to the NSL protocol level by automatically applying Hoare logic rules to "pull" the invariant through the protocol handlers.
4. Lift these results to the system level by applying Hoare logic rules interactively with user-supplied assertions.

We provide an example of the last point. At the NSL system level, the main cases concern the system user and network handlers. Roughly speaking, these compose the respective protocol-level handler with the user send function, which transmits network messages to the adversary. The user send function requires an additional pre-condition to preserve the invariant, which simply states that adding the sent message to the adversary knowledge does not compromise nonce secrecy. We manually apply the $bindN$ rule, which separates the respective protocol handler from the user send function. As the intermediate assertion *Q*, we use the conjunction of the invariants (main and auxiliary) with a previously established strong post-condition of the respective protocol handler, which states that the relevant nonces are fresh and encrypted and thus inaccessible to the adversary. Since these postconditions imply the pre-condition of the user send function, we use the consequence rule to complete the proof.

5 Bisimulation

Abstracting components (e.g. the state representation) can substantially improve proof automation. When doing this, we must ensure that the behavior of the abstracted component is equivalent to the original one. We use bisimilarity as our notion of equivalence. We first define bisimulation in the context of the state-exception monad and then give compositional proof rules for establishing the bisimilarity of two components. Finally, we present an example from our formalization of the BPW model.

5.1 Bisimilar components

The state-exception monad is deterministic. Therefore, two computations m_1 and m_2 are bisimilar, with respect to a witnessing relation *R*, if they produce a (unique) pair of bisimilar states and identical results, whenever started in a pair of bisimilar states.

$$
\begin{aligned}
&\mathit{bisim}\ R\ m_1\ m_2 \equiv \forall\, s\ t\ s'\ t'\ x\ y.\\
&\quad (s,\ t) \in R \wedge m_1\ s = (x, s') \wedge m_2\ t = (y, t') \longrightarrow\\
&\qquad (\exists\, v.\ x = \mathit{Value}\ v \wedge y = \mathit{Value}\ v \wedge (s',\ t') \in R)\\
&\quad \vee (x = \mathit{Exception} \wedge y = \mathit{Exception} \wedge (s',\ t') \in R)
\end{aligned}
$$

This notion is lifted pointwise to functions and afterwards to components by requiring that corresponding pairs of functions in both components match up[1]. Note that, since we are working with shallow embeddings of components and bisimulation, the notion of bisimilar components is not formalized itself in Isabelle/HOL. Instead, the proof that two components C and C' are bisimilar consists of a collection of lemmas of the form *bisim R f f'*, one for each pair of associated interface functions f of C and f' of C'.

We would like to prove such judgments compositionally by following the structure of the computations m_1 and m_2. The idea is to consider the statement *bisim R* m_1 m_2 as a Hoare-like tuple of the form $\{R\}\ m_1\ m_2\ \{R\}$, where the bisimulation relation R acts as both the pre- and postcondition. For compositional proofs, we need to generalize these judgments, as we can neither expect that intermediate pairs of states arising from program decomposition are in the bisimulation relation nor that intermediate outputs are identical. Therefore, we obtain the definition of *bisim R* m_1 m_2 as an instance of a generalized notion of a *bisimulation tuple*:

$$\begin{aligned}&\{R\}\ m_1\ m_2\ \{\lambda\, a\ b.\ S\ a\ b \mid T\} \equiv \forall\, s\ t\ \ s'\ \ t'\ \ x\ y.\\ &\quad (s,\ t) \in R \wedge m_1\ s = (x, s') \wedge m_2\ t = (y, t') \longrightarrow\\ &\qquad (\exists\ a\ b.\ x = \mathit{Value}\ a \wedge y = \mathit{Value}\ b \wedge (s',\ t') \in S\ a\ b)\\ &\quad \mid (x = \mathit{Exception} \wedge y = \mathit{Exception} \wedge (s',\ t') \in T)\end{aligned}$$

$$\mathit{bisim}\ R\ m_1\ m_2 \equiv \{R\}\ m_1\ m_2\ \{(\lambda\, a\ b.\ \{p.\ a = b\} \cap R) \mid R\}$$

Bisimulation tuples have two postconditions. The first postcondition is a parametrized relation $S\ a\ b$, required to hold whenever the computation m_1 returns the value a and m_2 returns the value b. The second postcondition T covers the case where both computations terminate with an exception. Note that $\{R\}\ m_1\ m_2\ \{\lambda a\ b.\ \mathit{true} \mid \mathit{true}\}$ expresses the property that both computations terminate with the same type of result. We call this property *equi-termination*.

5.2 Compositional proof rules for bisimulation

Since proving a bisimulation tuple $\{R\}\ m_1\ m_2\ \{S \mid T\}$ requires compositional reasoning about two computations simultaneously, we need $\mathcal{O}(n^2)$ proof rules to cover all cases, where n is the number of constructs of our "programming language". We can distinguish rules for pairs of computations with (1) identical top-level constructors from (2) those with different ones. As for (2), many combinations are only bisimilar in restricted or trivial cases. For instance, return and throw are only bisimilar, if the precondition is equivalent to *false*. A simple example of (1) is the return rule.

$$\frac{R \subseteq Sn\ a\ b}{\{R\}\ (\mathsf{return}\ a)\ (\mathsf{return}\ b)\ \{Sn \mid Se\}}\ \mathit{bisim\_return}$$

More interesting are the rules for composed computations. The basic rule for the case where both computations are sequential compositions is stated as follows.

$$\frac{\{R\}\ m_1\ m_2\ \{Sn \mid Te\} \qquad \bigwedge a\ b.\ \{Sn\ a\ b\}\ (k_1\ a)\ (k_2\ b)\ \{Tn \mid Te\}}{\{R\}\ \ (\mathsf{do}\ \ a \leftarrow m_1;\ k_1\ a)\ \ (\mathsf{do}\ \ b \leftarrow m_2;\ k_2\ b)\ \ \{Tn \mid Te\}}\ \mathit{bind\_sym}$$

[1] This definition can be seen as an instantiation of the coalgebraic definition of bisimulation [10] when the component is viewed as a coalgebra [9].

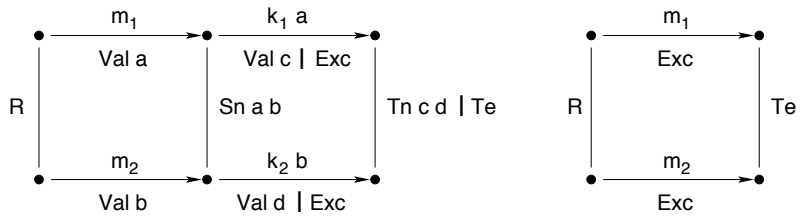

Fig. 1. Bisimulation for $bind$, symmetric case

This rule only covers the well-behaved case where the first parts of both sequential compositions equi-terminate. This situation is depicted in Figure 1, where computations are shown above the arrow and their results below. The rule $bind\_sym$ does not cover the case where m_1 and m_2 do not equi-terminate or where the second computation has a different structure. For such asymmetric cases, we derive the rule $bind\_left$ (below) and $bind\_right$ (symmetric, not shown).

$$\frac{\{R\}\ m_1\ m_2\ \{\lambda a\, b.\ \mathit{false} \mid Te\}}{\{R\}\ \ (\mathsf{do}\ a \leftarrow m_1;\ k_1\ a)\ \ m_2\ \ \{Tn \mid Te\}}\ \mathit{bind\_left}$$

When read backwards, this rule allows us to cut off the second part k_1 of a sequential composition with m_1, which is never invoked, in case both m_1 and m_2 always equi-terminate with an exception (and establish Te).

In general, when we prove a subgoal whose form matches the conclusion of rule $bind\_sym$ we must partition the relation *R* into two relations *Ra* and *Rb*, according to whether or not m_1 and m_2 equi-terminate on a pair of states in *R*. We then apply rule $bind\_sym$ in the former case and rules $bind\_left$ and $bind\_right$ in the latter case.

We have derived similar rules for the exception handling constructs. Again, all other control flow constructs including recursion are borrowed directly from HOL. All rules are sound by construction. Their completeness has not yet been investigated.

Using component invariants In non-trivial bisimulation proofs we usually need invariants of both components to strengthen the bisimulation relation. We have derived a specialized rule for sequential composition from rule $bind\_sym$, which allows us to use invariants in bisimulation proofs.

$$\frac{\begin{array}{c}\{R \cap I \times J\}\ m_1\ m_2\ \{Sn \mid Te\} \\ \bigwedge a\, b.\ \{Sn\ a\ b \cap I \times J\}\ (k_1\ a)\ (k_2\ b)\ \{Tn \mid Te\} \\ \{I\}\ m_1\ \{>\ \lambda z.\ I\} \qquad \{J\}\ m_2\ \{>\ \lambda z.\ J\}\end{array}}{\{R \cap I \times J\}\ \ (\mathsf{do}\ \ a \leftarrow m_1;\ k_1\ a)\ \ (\mathsf{do}\ \ b \leftarrow m_2;\ k_2\ b)\ \ \{Tn \mid Te\}}\ \mathit{bind\_sym\_inv}$$

The Hoare triples in premises 3 and 4 express that I and J are indeed invariants of m_1 and m_2, respectively. This rule allows us to "pull" the invariants over sequential compositions and carry them along in the proof.

Example 5 (Bisimulation of BPW model interface functions). Recall from the introduction that we have formalized two versions of the BPW model: a first one with DAG-based messages and a second one, where these messages are abstracted into inductively

defined terms. In order to allow us to work with the more abstract second formalization for concrete protocol verification, we have proved its bisimilarity with the first one. The state of the DAG-based version is a record containing a knowledge map *knowsI* of type $user \Rightarrow hnd \rightharpoonup ind$, which maps handles to indices for each user, and a table *db* of type $ind \Rightarrow 'd\ entry$, which maps indices to message entries. The latter correspond to message constructors, which may contain indices themselves. The bisimulation relation *I2M* expresses that for a given user and handle, the DAG-based message i in *knowsI* is abstracted to the message term *m* in *knowsM*.

$$I2M \equiv \{\ (s,\ t).\quad (\forall\ u.\ dom\ (knowsI\ s\ u)\ =\ dom\ (knowsM\ t\ u))\ \wedge \\ (\forall\ u\ h\ i\ m.\ knowsI\ s\ u\ h = Some\ i\ \wedge\ knowsM\ t\ u\ h = Some\ m \longrightarrow message\ s\ i\ m)\ \}$$

We prove bisimilarity for each pair of matching BPW model interface functions. Let us consider the case of the length query function. There we must show that the bisimulation relation is preserved and that both versions return the same message length. We obtain the subgoal below after unfolding the definitions of the length query functions.

$$\{I2M\ Int\ (finiteKnowsI\ \ Int\ \ lengthInv\ )\ \times\ finiteKnowsM\} \\ (\mathsf{do}\ (i,\ e)\ \leftarrow\ lookupI\ u\ h;\ \mathsf{return}\ \ (len\ \ e)) \\ (\mathsf{do}\ m \leftarrow lookupM\ u\ h;\mathsf{return}\ \ (len\_ofM\ m)) \\ \{\lambda la\ \ lb\ .\ \{p.\ \ la\ =\ lb\}\ \ Int\ \ I2M\ |\ \ I2M\}$$

Note the auxiliary invariants in the pre-condition of the bisimulation tuple. In particular, the invariant *lengthInv* of the DAG-based formalization states that a concrete message i and its abstract counterpart *m* (related by *message s i m*) have identical lengths. We apply the rule $bisim\_sym\_inv$ to separate the message lookup operation from the return statements. The intermediate relation *Sn* in this rule is instantiated to the following relation, which states that the index i pointing to entry *e* as returned by *lookupI* is abstracted to the term-based message *m* returned by *lookupM*.

$$\lambda(i,\ e)\ \ m.\ I2M \bigcap \{(s,\ t).\ \ knowsI\ s\ \ u\ \ h\ =\ Some\ i\ \wedge\ db\ s\ \ i\ =\ e\ \wedge\ message\ s\ \ i\ \ m\}$$

The first premise of the rule $bisim\_sym\_inv$ is covered by a previously proved lemma about the lookup functions. Thanks to the intermediate relation above, we can easily derive the second premise using the rule $bisim\_return$ and the invariant *lengthInv*. The remaining premises are discharged by previously proved invariant preservation results.

In summary, the rules allow compositional bisimulation proofs. They capture repetitive reasoning patterns and thus help us to concentrate on the essential aspects of the proofs, namely the invariants and intermediate relations.

6 Conclusions and Related Work

We have presented a general-purpose toolbox for modeling state-based components using monads, for reasoning about their properties using program and temporal logics, and for reasoning about their equivalence using a compositional proof system for bisimulation. Since our principal motivation was pragmatic rather than meta-theoretical, we have used shallow embeddings and introduced new concepts and proof rules only where we deemed them necessary to raise the level of abstraction. This allows us to exploit Isabelle/HOL's specification language, its extensive libraries, and its reasoning tools as

much as possible. For example, we have only formalized sequential composition and exception handling, while all further control structures are borrowed from HOL. Our experience in a substantial case study showed that this provides appropriate abstraction in modeling component-based systems and in proving complex theorems about them.

Related Work Filiâtre [5] appears to be the first to use monads to formalize an ML-like imperative language in the proof assistant Coq. He gives a functional translation of this language and generates proof obligations from pre-/post-condition annotations. Krstić and Matthews [11] interpret BDD algorithms written in C in a state-exception monad isomorphic to ours. They verify the functional correctness of these algorithms directly in Isabelle/HOL without additional program logics. In fact, they state that the complexity of the BDD programs forced them to use the monadic approach as a more flexible alternative to Hoare logic. However, our work shows that monads and Hoare logic can be combined to take advantage of features of both.

There is a sizable body of work on modeling Java(-like) languages and associated Hoare logics, including the Bali [22] and the LOOP [7] projects. The latter is most closely related to our work. They have formalized a coalgebraic semantics of Java and a corresponding Hoare logic in PVS. These handle non-termination, normal termination, and several forms of abrupt termination. In [9], the connection to monads is investigated. We do not model partiality, but we treat normal termination and exceptions similarly. Jacobs [8] develops WP calculi to improve proof automation and implements two of them as new PVS commands. In contrast, our Hoare logic is directly formulated in terms of the WP calculus and we use the standard Isabelle simplifier to rewrite WP expressions. Our approach is simpler, but does not admit alternative search strategies.

Nipkow has formalized Hoare logics for a family of while-languages and proved soundness and completeness results for each of them [17]. He models exceptions by adding an error flag to the state. As a consequence, most proof rules require pre-conditions depending on this flag. In contrast, in the state-exception monad, exceptions are more akin to values, which makes the rules for exception handling duals of those for return and *bind* and leads to a more abstract and modular treatment.

There exist several shallow embeddings of temporal logics in HOL. Långbacka has formalized the temporal logic of actions (TLA) in the HOL prover [13]. The Isabelle distribution [1] contains an axiomatization of TLA in HOL by Merz and a formalization of a temporal logic for I/O automata by Müller in HOLCF.

Benton [3] has proposed a relational Hoare logic (RHL) related to our bisimulation proof system to prove the correctness of optimizing program transformations for a while-language. He shows how to embed standard Hoare logic, as well as other logics, into RHL. We have chosen to follow a pragmatic approach and integrate our separate Hoare logic into the bisimulation proof system, which allows us to use the underlying WP calculus for automating proofs.

Future Work One direction for future work is to extend the modeling language, e.g. by adding object-oriented features such as inheritance, overriding and late binding along the lines of [16, 4] or by lifting the limitations on the definition of monad-based general recursive functions. Another interesting topic is the completeness of the bisimulation proof system and the possibility of adapting it to different kinds of monads.

References

[1] Isabelle home page. `http://isabelle.in.tum.de`, 2007.

[2] M. Backes, B. Pfitzmann, and M. Waidner. A universally composable cryptographic library. IACR Cryptology ePrint Archive 2003/015, Jan. 2003.

[3] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of Principles of Programming Languages (POPL)*, 2004.

[4] A. Brucker and B. Wolff. A package for extensible object-oriented data models with an application to IMP++. In *International Verification Workshop (VERIFY)*, August 2006.

[5] J.-C. Filliâtre. Proof of imperative programs in type theory. In *International Workshop, TYPES '98*, volume 1657 of *Lecture Notes in Computer Science*. Springer, 1998.

[6] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in Isabelle/HOLCF. In *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 2603 of *Lecture Notes in Computer Science*, pages 147–162, 2005.

[7] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, 2000.

[8] B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.

[9] B. Jacobs and E. Poll. Coalgebras and monads in the semantics of Java. *Theoretical Computer Science*, 291(3):329–349, 2003.

[10] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 6:222–259, 1997.

[11] S. Krstić and J. Matthews. Verifying BDD algorithms through monadic interpretation. In *Proceedings of VMCAI 2002*, volume 2294 of *LNCS*, pages 182–195. Springer, 2002.

[12] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. *Software - Concepts and Tools*, 17:93–102, 1996.

[13] T. Långbacka. A HOL formalisation of the temporal logic of actions. In *Theorem Proving in Higher Order Logics (TPHOL)*, volume 859 of *LNCS*, pages 332–345. Springer, 1994.

[14] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–139, 1991.

[15] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[16] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 349–366, 1998.

[17] T. Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[18] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[19] L. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.

[20] A. M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer, Berlin, 1991.

[21] C. Sprenger, M. Backes, D. Basin, B. Pfitzmann, and M. Waidner. Cryptographically sound theorem proving. In *19th IEEE Computer Security Foundations Workshop, Venice, Italy*, pages 153–166. IEEE Computer Society, July 2006.

[22] D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002.

[23] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.