Synthesis of Network Protocols

Dr. Christoph Kreitz
Dr. Douglas R. Smith

with Cordell Green, Garrin Kimmell, Jim McDonald, Eric Smith, Edwin Westbrook, Stephen Westfold

Kestrel Institute

16 April 2016

Contract Number: FA8750-12-C-0257
Effective Date: 31 Jul 2012
Expiration Date: 30 Jan 2017

Principal Investigators: Christoph Kreitz, Douglas R. Smith

Prepared for:

Air Force Research Lab/RITA
525 Brooks Road
Rome, NY 13441-4505

Table of Contents

Table of Figures

1 SUMMARY

This final report summarizes work funded by the DARPA I2O HACMS program. The project focused on the automated generation of secure and correct-by-construction network protocol codes.

Code to support communication is critical to modern systems and can be a major source of security vulnerabilities. The challenge of this project was to take a clean-slate design approach to explore ways to produce communication protocol codes together with proofs of their safety and security. Our approach is based on formal specifications of safety and security properties, automated refinement to transform high-level specifications down to code, and the emission of proofs during the development process. The technologies that we developed are applicable to a broad range of problems, and they are being applied and further developed in other programs.

The highlights of the research and development performed under this contract are listed next:

(1) System Specification — To capture the abstract essence of communication requirements we extended previous specification techniques in several directions:

- Global specification: a global view of the system and its properties as a set of interacting processes, each with their own/local observables. The global view allows capturing nonlocal properties across space, processes, and time. Ghost observables, which are variables that are only used for specification purposes, support the expression of global structure and requirements that relate the states of different processes at different times.
- Refinement to stateful and concurrent code: Kestrel's Specware system [Specware03] naturally supports the introduction of underspecified types and functions, which is necessary for an automated refinement approach. We found a natural way (i) to combine logical, algebraic, and coalgebraic types and operators and (ii) to refine them to imperative code. Support for both inductive and coinductive types has enabled a far more flexible specification and refinement language, and has supported the generation of idiomatic system code.

(2) New Transformations — We developed, implemented, and extensively used a suite of new transformations that generate correct-by-construction refinements. For example:

- MergeRules – This transformation allows us to specify a state-transforming operation by a collection of cases expressed by trace predicates. The

transformation weaves the trace predicates together to obtain a single postcondition for the operation, which can then be synthesized into code.

- C generation — We developed a code generator to allow the coalgebraic specifications to be translated into idiomatic state-changing code in C. We made progress in producing a proof-emitting C generator.

(3) Infrastructure Development – We further developed, and extensively applied a suite of transformations that generate correct-by-construction refinements for coalgebraic specifications. For example:

- Observer Maintenance — This transformation takes a state observer and an invariant that characterizes its meaning. The transformation calculates updates codes for each program operation to enforce the invariant.

- Observer Refinement — This transformation takes an observer whose observation type is abstract and provides a more concrete implementation of the type.

- Finalize Coinductive Type — This transformation and its dual (Finalize Inductive Type) allow us to incrementally add observers (resp. constructors) to a type. This supports a refinement process where we incrementally add constraints to types and their operators. The transformation generates a definition for the type and gives definitions for functions that are constrained by coinductive (resp. inductive) axioms.

- Globalize — The coalgebraic style of specification uses linear, or single-threaded, functions to express the dynamics of a state-changing system (such as a concurrent GC). The state of the computation is both an input parameter and a single output parameter. The single-threadedness allows this transformation to suppress the state parameter and treat it instead as a global variable. From purely functional specifications, we are then able to generate idiomatic imperative code.

(4) Proof Emission from Transformations — We pioneered techniques for extending our transformations so that they not only generate more concrete specifications, but they simultaneously output a machine-checkable proof that the refinement is correct. Generating proof scripts as a by-product of each refinement step is only possible because our transformations operate by performing explicit calculations in the domain theory (i.e. using domain axioms and theorems). We generate proof scripts expressed in the Isar sublanguage of Isabelle

(5) Derivation of UDP Protocol Code for the Linux kernel — In Phase 1, we specified the send and receive code for the UDP protocol via trace predicates, and applying the Merge Rules transformation to compose them into a synthesizable specification. The udp_send specification had some 37 rules partitioned into 11 modules, each representing a stage in the processing of a UDP datagram. The metaprogram for deriving C code automatically generated over 30k lines of proof, checked by Isabelle. We generated C code and successfully demonstrated it running in the Linux kernel.

(6) Derivation of a Family of Transport-level Protocol codes – In Phase 2, we focused on the synthesis of a family of Publish-Subscribe (PubSub) protocols to run on the Landshark and American-Built Automobile (ABA). We developed a family tree of metaprograms that are rooted at an abstract specification of communication requirements, such that each branch derives a different family member, namely, PubSub code for communication (i) between VM/s, (ii) within a VM, (iii) to/from a CertiKOS process, and (iv) over an IP radio link to a control unit.

(7) Datatype Refinements – We developed new datatype refinements that express the incremental implementation of abstract communication-related types. Examples include Mailbox2RingBuffer and RingBuffer2FlipFlopBuffer.

(8) Infrastructure Development – We made many improvements to Kestrel's Specware system and made it open-source via github at https://github.com/KestrelInstitute/Specware

(9) The overarch goal of this project was to express the design of communication code in terms of generally applicable design abstractions. The abstractions are incrementally translated to executable C via formal refinement steps that emit checkable proofs. The sequence of refinements and the design abstractions they embody constitute a (i) human-understandable explanation of the complex executable code, and (ii) and effective means for constructing complex code for a specific set of requirements out of reusable design abstractions.

2 INTRODUCTION

2.1 Motivation

The objective of the DARPA HACMS program was to develop clean-slate technology aimed for the construction of verified and secure code for military vehicles. Kestrel focused on the synthesis of correct-by-construction codes for transport-level communications.

2.2 Objective & Hypothesis

A long-term goal of Kestrel has been to demonstrate that the automated generation and evolution of software from requirements-level specifications provides a cost-effective alternative to current methodologies for software development. Benefits of the approach include correctness-by-construction, generation of certification evidence in the form of proofs, good performance, and productivity gains through automation. Our specific objective in this project has been to demonstrate the feasibility of automating the generation of correct-by-construction network protocol codes. In Section 4 we discuss the extent to which our results advance our long-term goal and demonstrate the claimed benefits.

2.3 What is program synthesis and what is our approach?

Broadly, program synthesis is the automated construction of programs from specifications of their intended behavior. Our specific approach is deductive and starts with the capture of program/system requirements in a formal specification. Alternative approaches include compilation from a programming language or DSL, and inductive generation of programs from examples. The full power of higher-order classical logic, as supported in Kestrel's Specware system [Specware03], is used to express requirement-level specifications as first-class entities along with operations for structuring, composing and refining specifications.

A major distinguishing feature of our refinement approach is the use of large-grain transformations that automatically calculate refinement steps. Other approaches to refinement (e.g. VDM, B, Praxis) rely on the post-hoc verification of manually created refinement steps. This is an expensive process, and it proves difficult to maintain the refinement chain under changes in requirements. Our approach also allows us to generate larger, more complex codes than synthesis systems based on SMT and other prover-based approaches. While there is a significant upfront investment in building up a domain-specific specification for network protocols (or other domains), the payoff comes downstream with the automated generation of families of codes together with

their proofs. The amortized cost over the product family and over its lifecycle should be dramatically lower than for other approaches to software production.

The synthesis approach that we developed in this project requires user input in two parts: (1) formal requirements specification, and (2) a metaprogram that controls the application of transformations.

The development of correct-by-construction code via a formal refinement process has the following form:

$$Spec_0 \rightarrow Spec_1 \rightarrow \ldots Spec_n \rightarrow Code.$$

The refinement process starts with a specification $Spec_0$ of the requirements on a desired software artifact. Each $Spec_i$, for i=0,1,...,n represents a structured specification and the arrows → are refinements. The refinement from $Spec_i$ to $Spec_{i+1}$ embodies a design decision which narrows down the number of possible implementations. The final step translates the lowest-level specification $Spec_n$ to code in a suitable programming language. Semantically the effect is to narrow down the set of possible implementations of $Spec_n$ to just one, so specification refinement can be viewed as a constructive process for proving the existence of an implementation of specification $Spec_0$; i.e. its consistency.

2.4 Network Protocols

This project focuses on the synthesis of the codes to implement a given protocol. We did not focus on the derivation of new protocols (e.g. [Datta05]), nor on the post-hoc verification of the security properties of protocols. Why is this a problem? Incorrect code for processing the sending or receiving of packets can be a source of vulnerabilities in systems. Protocol-handling code is often in the kernel and therefore must be protected.

3 METHODS, ASSUMPTIONS, PROCEDURES

Our objective was to demonstrate the feasibility of automating the generation of a family of network protocols from requirement-level specifications, together with correctness proofs. The overall approach is characterized by formal specifications, formal refinements, transformations to generate refinements, proof-emitting transformations, and metaprograms. A key assumption is that it will ultimately become practical for system developers to capture their requirements as formal specifications, and that the process of transforming those specifications to proof-carrying code can be largely automated, with some guidance, by the application of highly reusable automatic transformations.

3.1 General Approach

The synthesis approach that we developed in this project requires two forms of user input: (1) formal requirements specification, and (2) a metaprogram.

```
metaprogram =
transformation1;
transformation2;
transformation3;
…
```

Figure 1: Form of a Metaprogram

Figure 1 shows a metaprogram as a sequence of transformations. The metaprogram is a sequence of transformations to be applied to the requirement specification. The transformations are typically drawn from Specware's library. In a later section, we present a collection of new transformations that were developed as part of this project. The actual syntax/representation includes parameters to the transformations as well any theorems that should be applied.

Specware executes a metaprogram automatically. The effect, illustrated in Figure 2, is to sequentially transform the requirement specification into more refined specifications. Each transformation embodies some design knowledge, so the effect of applying a transformation is to generate (1) a refinement of the input specification into a refined specification that incorporates an instance of the transformation's design knowledge, and (2) a machine-checkable proof that the output specification is a refinement of the

input specification. The metaprogram is then an explicit and formal statement of the design content of the generated code.

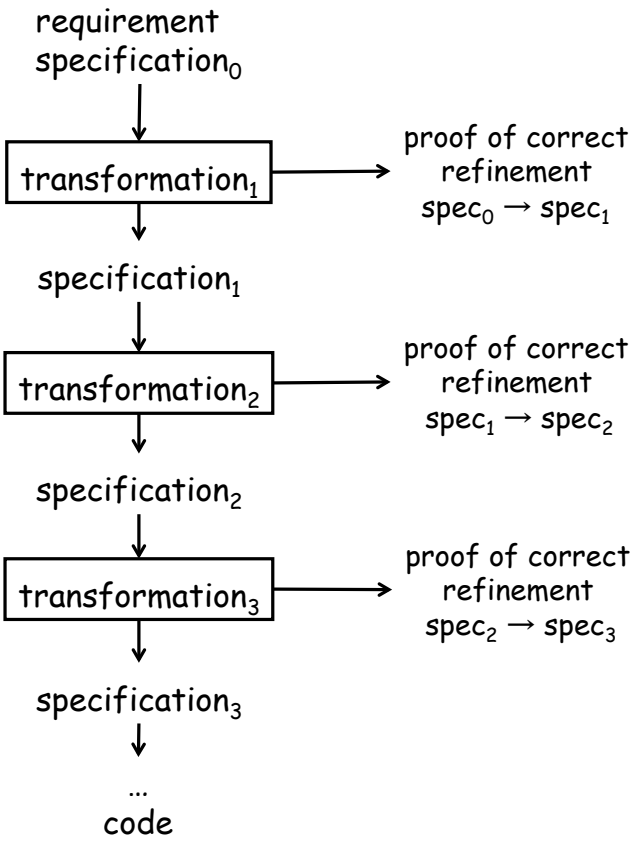

Figure 2: Executing a Metaprogram to generate code and proofs

Formal Derivation and Software Evolution

We would like to point out the consequences of this approach with respect to software evolution. Studies of evolution suggest that most changes to systems fall into a small number of categories, mainly bug-fixes, additional requirements, performance tuning, and migration. First, bug fixes are not relevant here since the code is generated with proofs of correctness (although bug fixes to the requirement specification will commonly arise). Second, the addition of requirements is facilitated in our approach because we have a formal specification of requirements. It is much easier to add requirements to a specification than to add them at the code level. Third, performance tuning also has an explicit locus in our approach, since it is manifest by extending or modifying the metaprogram — either adding new transformations, or modifying how existing transformations are applied (e.g. by adding theorems that the transformation can use). Finally, migration is often a matter of adapting the metaprogram to suit a new target

language or platform. Typically most of the metaprogram is preserved under migration with just some of the backend transformations needing to be changed. In summary, our approach, based on formal specification of requirements and the derivation structure via a metaprogram, provides good locality for the kinds of changes that arise in software evolution. This fact underlies our claim that this approach is essential to the future of Software Engineering.

3.2 Specifications and Refinement

A *specification* defines a language and constrains its possible meanings via axioms. A specification is given by a finite collection of type symbols (optionally including definitions), function symbols and their signatures (optionally including definitions), and axioms over the type and function symbols. We treat predicates as Boolean-valued functions. For purposes of this report, we focus on first-order specifications (i.e. functions do not take functions as arguments), although Specware allows higher-order specifications. The deductive closure of the axioms is a theory, so a specification is a finite presentation of a theory.

A refinement can be expressed formally via a *specification morphism*, which translates the language of one specification into the language of another specification in a way that preserves theorems. Formally, a *signature morphism* from specification S0 to specification S1 is a type-consistent map from the vocabulary of S0 (i.e. its type and function symbols) to the vocabulary of S1. A *specification morphism* from S0 to S1 is a signature morphism that preserves theorems; i.e. that translates each theorem of S0 to a theorem of S1. To establish a specification morphism, there is a *proof obligation* that each axiom of S0 translates to a theorem of S1. Let Morphism denote the type of specification morphisms (or simply morphisms).

Specification S1 is an *extension* of specification S0 if there is a specification morphism S0 → S1 whose underlying signature morphism is injective (i.e., each symbol of S1 comes from at most one symbol of S0). We use importation to express extension, allowing the construction of complex specifications by the composition of simpler specifications (which are often library specifications). More generally, specifications and their morphisms constitute a co-complete category, where the colimit operation provides a general means for constructing complex specifications. Intuitively, the colimit is the simplest specification that combines given specifications C modulo their common structure (e.g., two members of C each separately import some library specification).

As models of specification S, we admit any structure of sets and functions that interprets at least each type and function symbol in S and that satisfies the function signatures and the axioms. This semantics allows structures for extensions of S to be models of S.

The denotation of a specification morphism m is a map from models of the codomain of m into models of the domain — every model of S1 is mapped to a model of S0.

3.3 Proof-Emitting Transformations

Specification S0 *refines to* S1 if there is a specification morphism m:S0 → S1. We refer to m as a refinement and a morphism, and in context, S1 as the refinement of S0. In this paper we are interested in rules and techniques for automatically generating refinements. A s*pecification transformation* or a *transformation*, is a partial function on specifications that generates a refinement: t:Spec → Morphism. That is, if t(S) = m, then m:S→ codomain(m) is a refinement of S.

As discussed in the next section, we developed new transformations (and extended previously developed transformations) that support a coalgebraic style of specification, leading towards the generation of imperative code. Most of our transformations work by applying a sequence of equations (via rewrite rules) to parts of the given specification. The chain of equations that are applied proves the correctness of the resulting refinement. We developed techniques for saving the equation chain and emitting it as a proof structure that can be checked by an external proof checker. In our case, the proofs are expressed in the Isar format of the Isabelle proof assistant [Isabelle]. Isabelle is used to automatically check that the emitted Isar proofs are in fact proofs of the refinement proof obligations generated by Specware.

The upshot of using proof-emitting transformations is to co-generate both code and proof that the code satisfies its specification. This is in contrast to post-hoc verification approaches that seek to prove a program correct after it has been written. Generating proof-carrying code has the advantage that all design information is available to the proof generation process as the code is being constructed. We believe that this is a more economical approach to producing certifiably correct software.

3.4 Coalgebraic Specifications

Coalgebra is a relatively recent area of mathematical study, which, in a sense, is dual to algebra. It has provided a natural unifying foundation for exploring dynamical systems, including both discrete and hybrid automata. In functional languages and programming theory, it has been attractive as a way to model and reason about infinite and non-well-founded objects, such as streams and the behaviors of state machines [Rutten00, Jacobs97]. Coalgebra has also proved useful for giving a foundation to object-oriented languages and class hierarchies [Rothe01].

Our approach in HACMS was to use a mixture of algebraic and coalgebraic types in our specifications, and to develop new transformations to handle the cotypes. For our purposes, algebra, via inductive types, provides a foundation for specifying and refining finite data, such as Booleans, bounded Natural numbers, Lists, and finite Sets. Coalgebra, via coinductive types (aka cotypes), provides a natural foundation for specifying and refining stateful and concurrent computation.

There is a descriptive vocabulary that goes with cotypes. Algebraic types are characterized by their constructors, which are used inductively to build up terms for all values in the type. The inductive construction allows inductive definitions of functions and proof by induction. In contrast, cotypes are characterized by their destructors, which are operations on the cotype that decompose a cotype element into its parts. Typically destructors are categorized as *observers* (which observe an aspect of an element) or *transformers* (which transform an element into another element of the cotype). The iterated destruction of objects of the type gives rise to coinductive definition of functions and proofs by coinduction.

3.4.1 Algebras and Coalgebras

This section provides some formal background on algebra and coalgebra. It assumes knowledge of basic category theory and may be skipped.

Types and their functions are conveniently treated in the category SET of small sets, where sets are the objects and functions are the morphisms or arrows between objects. Algebras and coalgebras are characterized in terms of a polynomial functor F:SET→SET, built out of the basic functors

F: $X \longmapsto A$ (constant, for fixed set A)

F: $X \longmapsto Y+Z$ (sum)

F: $X \longmapsto Y \times Z$ (product)

F: $X \longmapsto X^A$ (dependent, for fixed set A)

in sum of product form; e.g.

$$F(X) = 1 + A \times X \qquad (1)$$

where 1 is a singleton set. Given functor F, an *algebra* is a function a:F(X) → X, and a *coalgebra* is a function c:X→F(X). For example, given the functor (1), an algebra has a carrier set X together with a function of type

a: $1+A \times X \rightarrow X$

which can be expressed in terms of constructors

c0: $1 \rightarrow X$
c1: $A \times X \rightarrow X$
and packaged as a function cotuple
a = [c0,c1] : $1+A \times X \rightarrow X$
The *initial algebra* for F is the least fixpoint of the equation (1), which is isomorphic to the set of all finite terms built up from the constructors. The initial algebra for our example is (isomorphic to) the set of finite lists over the set A, where the function/constructor c0 generates the empty list (Nil), and c1 adds an element to a list (Cons).

The construction of an initial algebra from constructor-based terms alone supports proof by induction over the constructors and definition of functions via inductive constraints. An inductive constraint characterizes the effect of a function on a constructor. For example, a function from finite lists to their length is completely determined by the inductive constraints
len(c0) = 0
len(c1(a,l)) = 1 + len(l).

We make use of this capability in defining a new refinement paradigm that incrementally accumulates constructors on a type and inductive constraints on functions, and finally declares that the constructor set is complete, determining a polynomial functor and corresponding initial algebra with function definitions that are complete and consistent by construction.

Dually, given functor F, a *coalgebra* is a set X together with a function $c:X \rightarrow F(X)$. The set X is called the carrier or cotype. For example, given the functor $1+A \times X$, a coalgebra has a carrier set X together with a function of type
c: $X \rightarrow 1+A \times X$
which can be expressed in terms of component destructors
d0: $X \rightarrow 1$
d1: $X \rightarrow A \times X$
such that
c = ⟨d0,d1⟩: $X \rightarrow 1+A \times X$
using the notation for a function tuple.

The *final coalgebra* for F is the greatest fixpoint of equation (1). The final coalgebra for our example is (isomorphic to) the set of finite and infinite lists (i.e. streams) over the set A, together with destructor d0 that is defined on the empty list, and function d1 that decomposes a nonempty stream into its head and the rest of the stream.

In this project, we are interested in reactive code for handling ongoing communications. We restrict attention to single-term monomial functors of the form

$$F: X \longmapsto A1 \times A2 \times \cdots Am \times X^{B1} \times X^{B2} \times \cdots \times X^{Bn}$$

where

$o_i: X \rightarrow Ai$ for i=1,..,m (observers)

$t_j: X \rightarrow X^{Bj}$ for j=1,..,n (transformers that have an X and Bj as input)

F can be treated as a function tuple

$$F = \langle o1, o2, \cdots, om, t1, t2, \cdots, tn \rangle$$

where each oi is called an *observer* because it extracts some information about the unknown type X. Each tj is called a *transformer* since it transforms an element of X (which can be thought of as a state) to another element (perhaps informed by an argument of type Bj).

Treating X as state, together with the single-threadedness of the transformers, allows us to generate imperative code for the transformers.

The definition of a final algebra from destructor-based terms alone supports proof by coinduction over the destructors and definition of functions via coinductive constraints.

A coinductive constraint characterizes the effect of a transformer on observers. A prototypical example is

$$o1(t1(b,st)) = alpha(b, st, o1\ st)$$

where the transformer t1 is partly characterized by its effect on the o1 observer: the o1 observation of the new state produced by t1 is some function (alpha) of the input state and the o1 observation on the previous state.

Related Work

We have presented coalgebra in terms of a functor, an approach sometimes called F-(co)algebra. The Coq prover uses a different but equivalent formulation that stresses the formation of the carrier of the coalgebra – a coinductive type. Cotypes in Coq are specified in terms of constructors, just like inductive types (the carrier of an algebra), but elements of the cotype are identified with possibly infinite terms over the constructors. For example, it is natural to think of a stream as an infinite term with each subterm headed by a Cons constructor:

```
CoInductive Stream (A: Set) : Set :=
| Cons : A →  Stream A →  Stream A.
```

A Coq specification for finite and infinite lists over a set A is given by adding a Nil constructor to terminate construction:

```
CoInductive LList (A: Set) : Set :=
| LNil : LList A
| LCons : A → LList A → LList A.
```

Destructors can then be defined by patterns matching on constructors, just as with inductive types. To define a transformer on a cotype, we can use pattern matching to destruct the given cotype element and the output is a constructor-headed term that may involve a corecursive call. For example, the image of a stream of A's with respect to a given function f:A→B can be defined in Coq by

```
CoFixpoint image (f: A → B)(s:Stream A) : Stream B:=
match s with Cons a tl ⇒ Cons (f a) (map f tl)
end.
```

This is formally equivalent to specifying image via coinductive constraints, as in

```
op image(f: A → B)(as:Stream A) :
        {bs:Stream B |   hd bs = f (hd as)
                      && tl bs = image f (tl as)}
```

In this simple example, the constructor-based formulation is simpler. When we get to transformers over cotypes with more internal structure, then the destructor/accessor-based coinductive constraints tend to be more concise.

Another advantage of the F-coalgebra approach over the coinductive construction approach is in the area of specifying dynamical systems (as opposed to possibly infinite data). A coalgebra embodies the state-changing dynamics of a system; e.g. for our purposes, protocol-handling processes. The state of such a system is typically finite and evolves by application of transformers. The unique homomorphism to the final coalgebra typically maps the current state to its possibly-infinite tree of behaviors (i.e. possible future successor states).

3.4.2 Specification

Here is a generic specification that illustrates the coalgebraic style that we developed in this project:

```
S = spec
  cotype State
  op obsA:State-> A
  op obsB:State-> B
  op obsC(st:State):C = h(obsA st, obsB st)
```

```
    op f(st:State)(arg:Arg):
        {st':State| obsA st' = alpha obsA st
                  & obsB st' = beta  obsB st}
    op g(st:State)(arg:Arg):
         {st':State,d:D| obsA st' = gamma (obsA st) arg
                       & obsB st' = delta (obsB st) arg
                         &       d = eps (obsA st) (obsB st) arg}
end spec
```

Spec S has two basic (undefined) observers (obsA and obsB), a defined observer (obsC), and two transformers (f and g). The latter are specified by giving coinductive constraints (postconditions) stated as the predicates of a dependent output type. That is, the output type of f is the set of all States st' such that the obsA observation is given by (alpha (obsA st)); i.e. by some function of the old obsA observation. This is a dependent type because it depends on the value of st that is bound when the transformer is called. The types A, B, and C may be algebraic (i.e. constructor-based). The Greek-letter functions (alpha, ...) capture the effect of the transformer on their particular observer.

Here is a simple specification of communication primitives using this style:

```
Comm = spec
  import Queue
  type Msg
  cotype State
  op      inbox: State→ Queue Msg
  op    outbox: State→ Queue Msg
  op  send(st:State) (m:Msg) : {st':State|     inbox st' = inbox st
                                          & outbox st'  = (outbox st) ++ [msg] }
  op  recv(st:State | inbox st ≠ empty_queue):
          {(m,st'):Msg*State|              m = top (inbox st)
                             &    inbox st' = rest (inbox st)
                             &  outbox st' = outbox st }
  end-spec
```

Spec Comm imports a (polymorphic) specification of finite queues and introduces an undefined type of messages: Msg. It then introduces an undefined cotype State that has two observers, inbox and outbox, and two transformers: send and recv. All that we know about a State is what we can observe, which is its current input queue inbox and its output queue outbox. The send transformer changes the State by adding a given message to the outbox, expressed via a coinductive constraint. The send transformer is specified to have no effect on the inbox observation. The result of send is completely

specified in terms of what observations we can make of the new State. Similarly, the recv transformer is specified to operate in a state in which the input queue inbox is nonempty, and to output the front element m of inbox, while popping m from the queue in the resulting state st' and leaving outbox unchanged. The communication channel itself moves elements from the sender's outbox to the receiver's inbox.

As a notational convention, we will often omit coinductive constraints that assert no-change in an observer.

In this style of specification, it is important to begin formalization with an understanding of what the observers are, and the distinction of identity versus value. For a single changing value, a simple observer of state is sufficient. For a structured collection of values, an observer that is parametric both on state and unique identifiers for the values is needed. That is, the observer function itself is a unique identifier, but if there is a collection of changing values, then an identifier type Id must be introduced and an observer that is parametric on Id is introduced to observe individuals of the collection.

Under refinement, it is permitted to add new types, functions, and axioms; also it is permitted to strengthen axioms and postconditions in dependent types. We make use of this capability in defining a new refinement paradigm that incrementally accumulates destructors on a type and coinductive constraints on transformers, and finally declares that the destructor set is complete, determining a polynomial functor and corresponding final coalgebra with function definitions that are complete and consistent by construction.

Section 3.5 provides two examples of this style of specification: (1) an abstract specification of communication requirements, and (2) a lower-level specification of required behavior.

3.4.3 Ghost Observers

We distinguish a special class of observers that provide information about the program and its execution, but do not usually take part in the computation. A typical example is a history observer that observes the trace of state-action pairs in the computation to the current state. Such a history observer

```
ghost op hist: State -> List (State * Action * State)
```

helps to specify key properties of a program. The intent of a ghost observer is to reify some extra-computational aspect of the (desired) computation into state for purposes of specifying requirements. Typically, we do not want to implement a ghost observer such as hist, since that would tremendously expensive. Sometimes however, some fragment of a ghost observer is needed to realize requirements. This is one way that extra state variables accumulate during development – as a way to reify into the present, some part

of the past or some reflection of the computation or code that is useful for computation itself.
Conceptually, each transformer of the target program maintains all ghost observers. For example, the send and recv transformers are refined with a coinductive constraint that records the send/recv action into hist:

```
op  send(st:State) (m:Msg) : {st':State|     inbox st' = inbox st
                                          & outbox st'  = (outbox st) ++ [msg]
                                          & hist st' = hist st ++ [<send, time st, msg>] }

op  recv(st:State | inbox st ≠ empty_queue):
        {(m,st'):Msg*State|              m = top (inbox st)
                                &    inbox st' = rest (inbox st)
                                &  outbox st' = outbox st
                                & hist st' = hist st ++ [<recv, time st, msg>]  }
```

This maintenance of hist allows us to define new ghost observers that are useful in specifying key communication properties. It is useful to define specializations of hist for particular operations/transformers

```
ghost op       sent: State → Channel → List Message
ghost op received: State → Channel → List Message
```

which can be maintained in send & recv as follows:

```
op  send(st:State) (m:Topic) :
          {st':State | … & sent st'  = (sent st) :: m  }
op  recv(st:State)(m:Topic):
          {st':State) | … & received st' = (received st) :: m }
```

As an example of the usefulness of ghosts, a perfect communication channel from process P to process Q can be specified by

$\forall$st:State. P.sent st = Q.received st

or in temporal logic

□ sent = received.

Another useful ghost observer records the changes over time of an observer:

ghost op changes: (State → E) → State → List E

which is maintained in each transformer that modifies the given observer

op g (st:State): {st':State | … e st' = alpha st
& changes(e) st' = (changes e st) :: <alpha st> }

This is also a specialization of hist and could be defined in terms of hist.

The ghost observers specified above are all that we need for synthesizing communication codes. However, in other domains, it may be useful to use ghosts to reflect not just the past, but also aspects of the present that are extra-computational, such as resource utilization (bandwidth, memory, CPU) and physical characteristics of the underlying hardware (energy, temperature, operating environment).

3.4.4 Specification of Time

Latency is intrinsic to communication channels, so we might want to put latency bounds into our requirements. The specification of a perfect communication channel that we presented above, sent = received, assumes no latency. We can define perfect communication modulo latency in terms of the following predicate (where a and b are observers of the same type and d is a duration):

$a \Rrightarrow_d b$

holds in the current state st1 if there exists a recent state st0 such that

1. a st0 = b st1
2. time st1 ≤ time st0 + d.

Formally this is

$$(a \Rrightarrow_d b)\ st1 \equiv \exists\ st0.\ st0 \in hist\ st1 \wedge (a\ st0 = b\ st1) \wedge time\ st1 \leq time\ st0 + d.$$

We can lift this relation between observers to lists (taking the image of the zip), allowing us to specify perfect communication modulo latency:

$sent \Rrightarrow_d received.$

That is, the messages received via a communication channel correspond one-to-one with messages sent and with no more than a delay of d seconds.

Periodicity is another temporal relation that arises in communication. We can specify a ghost observer period(p,a) which takes a period and an observer, and returns a history of a observations of period p up to now. For example

period(p,clock) = [0,p,2p,3p,…, np]

where np ≤ clock st < (n+1)p. The periodic sampling of hist is

period(p,hist).

Corresponding to this specification-level ghost observer (or observer constructor), we have the computational control construct: with_period(p){g} that executes transformer g with period p, such that if g has specification

op g(st:State):{st':State | c st' = e st ∧ …}

and our program consists of

with_period(p){g}

then

changes c st = period(p,c).

3.5 Specification of Protocols

The HACMS program aimed to develop formal methodology for developing safe and secure vehicles. The communication needs of vehicles have some differences from the more familiar world of email and the web. When we connect to a website or interact with a colleague via email, we require that all bits of the communication are delivered reliably. In contrast, the controller of a vehicle periodically needs sufficiently fresh information about the status of the environment and the vehicle, not all status information – the recipient of status information does not require to receive all status updates. It just requires that the recipient has access to sufficiently fresh status information when needed. This entails that old information can overwrite fresher information, regardless of whether it has been accessed. Out-of-order data is even possibly correct, as long as the received data is sufficiently fresh.

In the following subsections, we discuss the specification of communication requirements, first at a global and abstract level, and then, in Section 3.5.2, a more local, behavioral level.

3.5.1 Abstract Communication Requirements

We specify abstract communication requirements, which can be implemented by any of several abstract protocol theories. Each protocol theory is expressed in terms of a refinement from (1) a specification of abstract communication requirements (typically expressed in terms of several processes, each with their own local observers, and a global communication requirement, to (2) a communication solution that introduces communication channel and operations, and local requirements. The codomain has enough structure that, if implemented, guarantees implementation of the domain communication requirements. The codomain specification has operators that are typically expressed as a set of cases or rules, as discussed in a later subsection.

We specify here two protocol theories for Publish-Subscribe communication mechanisms (Eager and Periodic), one of which we used to generate code for the RADL architecture running on the Landshark and ABA [RADL 15].

In Phase 2, we sought to start our synthesis process at a much more abstract level than for the UDP derivation in Phase 1 (described in the next subsection). The ROS and RADL architecture on the ground team vehicles assumes a Publish-Subscribe (PubSub) communication infrastructure.

The essential communication requirement is this:

- process A with observer pub: State→Topic
- process B with observer sub: State→Topic
- global requirement: □ A.pub $\Rightarrow_f$ B.sub

That is, there are two components A and B, where component A can directly observe information of type Topic, and component B cannot directly observe Topic. Moreover, B requires to observe Topic with some degree of freshness; more specifically, B requires that its observation of type Topic agrees with A's observation of no more than f seconds in the past; i.e. with "freshness" f. Inserting communication technology to connect the processor that supports A and B allows communication software to satisfy B's requirement with some bounded latency.

A simplified version of our abstract requirement specification for a PubSub situation is as follows. The predicate $\Rightarrow_f$ is expressed textually using the higher-order function fresh (polymorphic on types a, b, and c):

```
op [a,b,c] fresh(st:State,
                 obs          :State -> c -> a,        % Publisher's offered data
                 tracking_obs:State -> b -> a,         % Subscriber observer
                 latency:Duration):Bool =              % max latency
  ex(prev_st:State) obs prev_st = tracking_obs st
                    && clock st <= clock prev_st + latency
```

which is imported via the spec Common; so

(A.pub $\Rightarrow_f$ B.sub) st

is expressed

(fresh st A.pub B.sub f).

```
BasicComm = spec
  import Common#Tracking
  type Topic
  op initializeTopic:State->Topic
```

```
  op maxLatency: Duration
end-spec

PubRqt = spec
  import BasicComm
  type Publisher
  op pub_topic: State -> Publisher -> Topic
end-spec

SubRqt = spec
  import BasicComm
  type Subscriber
  op sub_topic: State -> Subscriber -> Topic
  op set_subtopic(st:State)(sub:Subscriber)(d:Topic):
              {st':State | sub_topic st' sub = d }
  op defaultTopic:State->Topic

end-spec

SystemRqt = spec
   import PubRqt, SubRqt
   axiom Global_Requirement is
      fa(st:State,pub:Publisher,sub:Subscriber)  fresh st pub sub maxLatency
end-spec
```

The key requirement here is the axiom Global_Requirement, which asserts that the Subscriber's access to the topic, via sub, will always access a value that is no more than maxLatency seconds out of date.

There are at least two ways to refine this structured specification. The Eager solution adds a communication channel and an operation for eagerly publishing the current value of the pub_topic; i.e. whenever pub changes, its new value is sent. The Periodic solution adds a communication channel and an operation for periodically publishing the current value of the pub_topic topic; i.e. every p seconds, the current value of pub is sent to Sub. In either case, the refined structure is strong enough to prove the Global_Requirement axiom. At the same time, it introduces new local requirements, expressed as axioms, that will lead to further refinements. These refinements are formalized in Section 3.6.1. The derivation is elaborated in Section 4.3.

3.5.2 Behavioral Specification by Cases

One of the first challenges we faced was how to specify the handling of a protocol, such as UDP in the Linux kernel, where there is no given formal specification. Practically,

the handling of UDP packets is governed by RFC standards and by common practice in various operating systems. In this situation it seemed that the best we could do is to model the required behavior as gleaned from standards documents and operating system codes.

Our general hypothesis is that the required behavior of complex state transformers can be disentangled into cases, typically a normal case behavior and several (perhaps many) abnormal and error cases. Our methodology for formally specifying complex behaviors is then to express the various cases via trace predicates. The specification of the whole is then the disjunction of the cases. The cases can overlap, which means that there can be some nondeterminism to the specification. This is acceptable as long as the cases are accurately expressed. Under refinement, we are free to choose any behavior that satisfies at least one of the cases.

In specifying a state transformer, each case/rule serves to formalize the relation between the initial/input state, intermediate states, and the final/output state of the transformer. As a simple introduction, consider the following simple cases:

C1. A ∧ B ∧ α(st, st') -- when A and B hold, do α
C2. A ∧ ¬B ∧ β(st, st') -- when A and ¬B hold, do β
C3. ¬A ∧ γ(st, st') -- when ¬A holds, do γ

The disjunction of these cases can be merged straightforwardly as follows

```
C1 ∨ C2 ∨C3 =  if A
               then if B
                    then α(st, st')
                    else β(st, st')
               else γ(st, st')
```

See Section 3.6.2 for details of the mergeRules transformation. Details of the cases for UDP datagram transmission and reception are given later in Section 4.1.

3.6 Design Theories and Transformations

In this section, we present a series of novel transformations that generate refinements in our derivations. There are three sources of techniques for generating refinements:

(1) Manual Extensions — i.e. manually written extensions of a specification
(2) Transformations — transformations that generate refinements.

(3) Library Refinements — Refinements that are reusable, and are developed and proved once, and stored in a library. They are applied via a pushout operation.

As described in Section 3.1, we manually write a metaprogram, also called a derivation script, which is an executable sequence of refinement steps applied to an initial specification. Each step prescribes how to generate a refinement of the previous specification.

Each of the following subsections introduces a library refinement or a transformation for generating refinements, together with some examples. We also discuss how each technique can automatically generate a formal checkable proof as a byproduct of its action.

Coalgebraic refinements simply add further constraints to previously introduced transformers, rather than producing constructive definitions. It is only at the end that constructions are given; i.e. that a particular model is chosen. This contrasts with algebraic style refinement in which constructors are given for types and operators are inductively defined over the types. All constructions are explicit and immediate.

In the following subsections we describe transformations that are specific to coalgebraic specifications. These are new transformations that we developed for our protocol derivations.

3.6.1 Eager and Periodic Protocol Theories (Refinements)

The essential publish-subscribe communication requirement is this:

- process A with observer pub: State→Topic
- process B with observer sub: State→Topic
- global requirement: □ A.pub $\Rrightarrow_f$ B.sub

That is, there are two components A and B, where component A can directly observe information of type Topic, and component B cannot directly observe Topic. Moreover, B requires to observe Topic with some degree of freshness; more specifically, B requires that its observation of type Topic agrees with A's observation of no more than f seconds in the past; i.e. with "freshness" f. Inserting communication technology to connect the hardware that supports A and B allows communication software to satisfy B's requirement with some bounded latency. If B were to require the information infrequently, then an *on-demand/pull* approach would be sufficient, via a request-response whose latency is less than the required freshness. For real-time systems,

such as in vehicle control, the need for sensor-based information is frequent and a *Publish-Subscribe/push* mechanism is sufficient.

For simplicity of notation, we sometimes omit the process qualifier when it is clear in context; e.g. $\Box$ pub $\Rightarrow_f$ sub.

We developed two protocol theories for providing push solutions to a basic communication requirement. They are both expressed as refinement morphisms, where the domain specification formalizes the communication requirements, and the codomain introduces the communication infrastructure and local requirements needed to prove that the communication requirements are satisfied.

The Eager Protocol publishes a change to pub whenever it changes; i.e. eagerly.

Eager Protocol Theory

Domain of Protocol Refinement:

- process A with observer pub: State→Topic
- process B with observer sub: State→Topic
- global requirement: $\Box$ A.pub $\Rightarrow_f$ B.sub

Codomain of Protocol Refinement

- Add a communication channel C from A to B
- operators send: State→Topic→State, recv: State→Topic→State
- local requirements
    1. $\Box$ sent = $changes_{pub}$
    2. $\Box$ sent = C.inbox $\Rightarrow_d$ C.outbox = received
    3. $\Box$ received= $changes_{sub}$
    4. $d \leq f$

Theorem: $\Box$ A.pub $\Rightarrow_f$ B.sub

The proof of the theorem in the codomain theory proceeds by induction on the steps of a computation. Assume that the value of pub changes in the transition from st to st', so

$changes_{pub}$ st' = ($changes_{pub}$ st)::newpub

and assume inductively that

(pub $\Rightarrow_f$ sub) st'.

Then,

$changes_{pub}$ st' = sent st' by local requirement 1

= C.inbox st' by local requirement 2

$\Rightarrow_f$ C.outbox st'' for some st'' such that
clock st'' ≤ f+ clock st',
by local requirement 2
= C.outbox st'' by local requirement 2
= received st'' by local requirement 2
= $changes_{sub}$ st'' by local requirement 3

which implies

$changes_{pub}$ st' = ($changes_{pub}$ st)::newpub
$\Rightarrow_f$ ($changes_{sub}$ st)::newpub using the induction hypothesis
= $changes_{sub}$ st''

which implies

(pub $\Rightarrow_f$ sub) st''.

The eagerness of this protocol theory is expressed by the first local requirement:

□ sent = $changes_{pub}$.

To enforce this requirement, the Observer Maintenance transformation is applied as follows.

Assume: sent st = $change_{pub}$ st inductive assumption
pub st' = newTopic assumed update to pub
Simplify: sent st' = $change_{pub}$ st'

Calculate:

sent st' = $change_{pub}$ st'
≡ sent st' = ($change_{pub}$ st :: newTopic)
≡ sent st' = (sent st :: newTopic).

The last formula is a spec for the enforcement operation and is satisfied by:

send newTopic.

That is, simultaneously (and atomically) with the update to pub, we send the new value newTopic to the Subscriber.

Analogously, Observer Maintenance of the third local requirement leads to updating the value of sub with the value in the Subscriber's inbox:

Assume: received st = $change_{sub}$ st
st' = recv st msg
received st' = received st :: msg
inbox st' = msg
Simplify: received st' = $change_{sub}$ st'

Calculate:

received st' = $change_{sub}$ st'

≡ (received st :: msg) = $change_{sub}$ st'

≡ ($change_{sub}$ st :: msg) = $change_{sub}$ st'

≡ ($change_{sub}$ st :: (inbox st')) = $change_{sub}$ st'

≡ ($change_{sub}$ st :: (read st')) = $change_{sub}$ st'.

The last formula is a spec for the update operation and it is satisfied by:

sub := read().

The resulting design is shown in Figure 3, where the red text indicates the injected code that enforces local requirements 1 and 3. The remaining local local requirements must be satisfied by the implementation of the communication channel C.

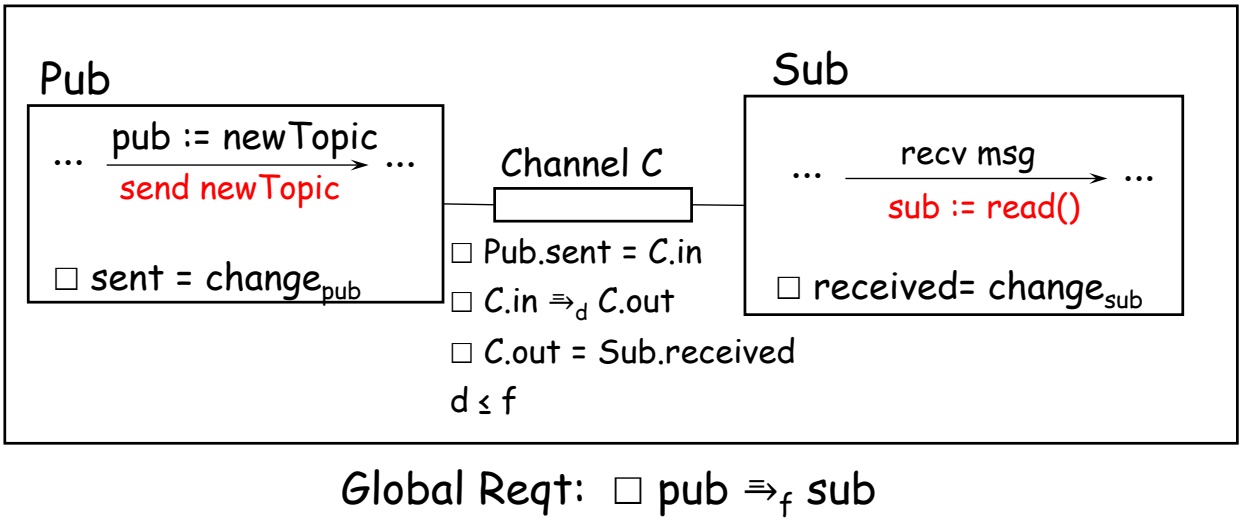

Figure 3: Eager Protocol Theory

The second protocol theory is useful when the frequency with which pub changes is higher than the frequency with which sub needs fresh values (≥ 1/f).

Periodic Protocol Theory

Domain of Protocol Refinement:

- process A with observer pub: State→Topic
- process B with observer sub: State→Topic
- global requirement: □ pub ⇛$_f$ sub

Codomain of Protocol Refinement

- Add a communication channel C from A to B
- operators send: State→Topic→State, recv: State→Topic→State
- local requirements:
    1. ☐ sent = periodic(p,pub)
    2. ☐ sent = C.in $=_d$ C.out = received
    3. ☐ received= periodic(p,sub)
    4. p+d+p ≤ f

Theorem: ☐ pub $\Rightarrow_f$ sub

We can perform similar transformations to the local requirements 1 and 3 as we performed above for the Eager Protocol. The resulting design is shown in Figure 4, where the red text indicates the injected code that enforces local requirements 1 and 3. The injected code uses a new control construct, with-period(p,act), that sets up an independent thread that executes action act every p seconds. The remaining local requirements must be satisfied by the implementation of the communication channel C.

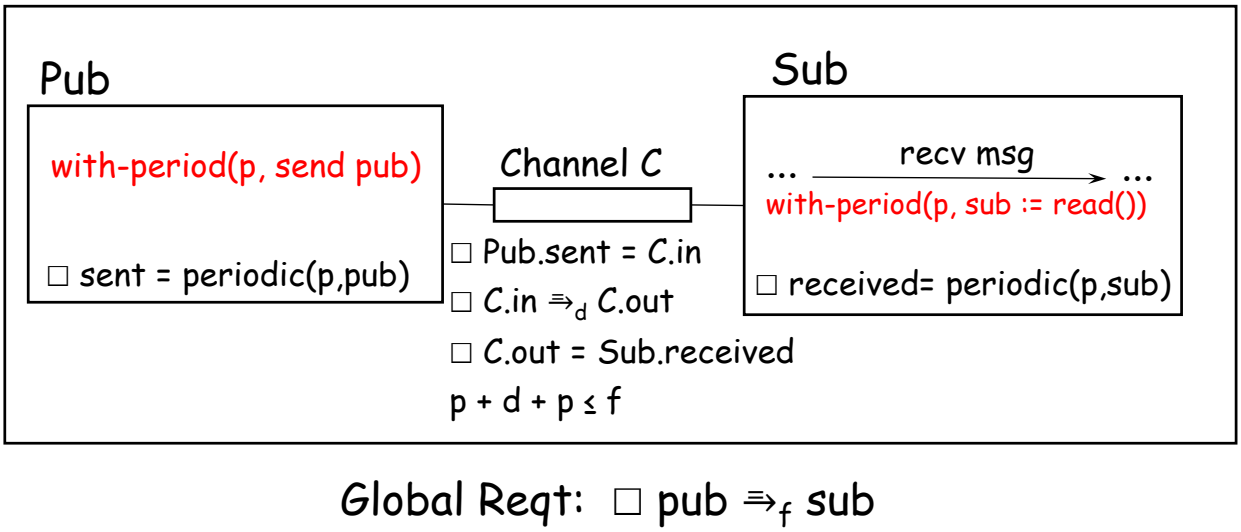

Figure 4: Periodic Protocol

Both of the protocol theories are important, but we used the Periodic Protocol in our HACMS derivations presented in Section 4.3.

3.6.2 mergeRules Transformation

Suppose we have the following simple cases:

C1. A ∧ B ∧ α(st, st') -- when A and B hold, do α

C2. A ∧ ¬B ∧ β(st, st') -- when A and ¬B hold, do β
C3. ¬A ∧ γ(st, st') -- when ¬A holds, do γ

we can straightforwardly merge the disjunction of these cases:

```
C1 ∨ C2 ∨C3 =  if A
               then if B
                    then α(st, st')
                   else β(st, st')
               else γ(st, st')
```

The mergeRules transformation also handles other first-order features including constructors for inductive types; for example, given the definition

```
type Option t = | None  | Some t
```

and cases

C1. x=None ∧ α(st, st') -- when x is None, do α
C2. x=Some y ∧ β(st, y, st') -- when x is Some y, do β

then the mergeRules transformation generates

```
C1 ∨ C2 =  case x of
             | None   -> α(st, st')
             | Some y -> β(st, st')
```

We extended mergeRules to address the following issues and features

- handling data dependencies between conjuncts
- heuristic choice of atom to branch on
- control the explosion of cases by hiding/nesting some disjunctions
- introduction of quantification
- functions with multiple inputs and outputs
- pattern matching to destructure values
- deriving preconditions to preclude false postconditions
- generating proofs as part of the construction.

We now discuss the general approach to merging rules.

Gathering rules, pre and post conditions

Each rule must have the form:

op r(s:st, i1:S1, ... ,ik:Sk | pre): { (s',o1,....oj) : (st*T1*...*Tj) | post }

From this signature, we extract a number of features:

1. The parameter s is the name of the current state. It must be the first parameter, if there are any others.
2. The type st is the state type.
3. Parameters i1 to ik are the inputs. Their associated types are S1 to Sk. Inputs are optional.
4. The predicate pre is a boolean expression that can mention s and i1 ... ik.
5. The output name s' is the name of the next state. It must be the first element of the range tuple. If there are no outputs o1 ... oj, then it should be only output.
6. The next state type st must be the same as the current state type.
7. The names o1 to oj are the outputs, with types T1 to Tj, respectively. Outputs are optional.
8. The predicate post is a postcondition. It is a boolean expression, and can mention s, i1, ... ik, s', o1, ... , oj.

All rules must have the same type, except the preconditions pre_i and postconditions $post_i$ can vary.

Converting to DNF

Given a rule pre- and post-condition, we then construct a boolean expression representing pre and post in DNF. Moreover, we track the existentially quantified variables mentioned by pre and post, as they will represent local definitions. Conversion to DNF proceeds by structural recursion on the expression pre && post according to the following rules:

1. If the top-level connective is a disjunction or conjunction, we convert the subterms to DNF and combine the results using DeMorgan to get an expression in DNF.

2. If the top-level connective is an existential quantifier, we convert the body to DNF, recording the quantified variables.

3. Otherwise, we have an atomic boolean expression and we do not process further.

After processing each rule, we have a set of pairs, where each pair (vs_i,f_i) consists of the existentially quantified variables vs_i for rule i and a boolean formula f_i in DNF, equivalent to the non-normalized formula pre && post. We union the sets of variables to

get a complete set of variables vs and take the disjunction of the individual formulae f_i to get an overall formula f, also in DNF.

Finally we calculate the set of observers for the state type st by locating all of the ops of the form:

op obs_i : st -> a

where a is any type that is not st or a tuple type with st hereditarily an element. We denote the set of all obs_i as obs.

Classifying expressions

Having calculated st, s, s', inputs, outputs, vs, obs, and f, we now classify each atomic boolean expression e_i in f according to the following syntactic criteria:

1. If e_i has the form e' = v, where v is a variable that is either the same as the poststate st' or an element of obs, then we say that e is a *constraint*. We say e *constrains* v, and that v *depends on* the set of variables defined by fv(e') $\cap$ vs.

2. If e_i has the form e' = (v_1,...,v_i), where v_i are all variables that are elements of vs then we say that e is a *definition*. We say e *defines* v_i , and that v_i *depends on* the set of variables defined by fv(e') $\cap$ vs.

3. If e_i has the form e' = C (v_1,...,v_i), where vi are all variables that are elements of vs (or a constant) and C is a data constructor, then we say that e *discriminates* e'. We say e *defines* v_i, and that v_i *depends on* the set of variables defined by fv(e') $\cap$ vs.

4. Otherwise, we say e is a *branch expression*. We say that e *depends on* the set of variables defined by fv(e) $\cap$ vs.

Simplifying clauses

A formula f in DNF will consist of the disjunction of a set of clauses c_1 ... c_n, each of which is the conjunction of a set of atomic boolean expressions. When generating a splitting tree for the merge of a collection of rules, we use a mechanism similar to resolution. Thus, we define two relations, *conflict* and *subsumption* between an atomic boolean expression and a clause.

1. If e_i is an atomic boolean expression, and c is a clause,
   then e_i *conflicts* with e_i if c contains an atomic boolean expression ~e1

(similarly, if e_i has the form ~e_i' and c contains e_i').

2. If e_i is an atomic boolean expression of the form C_i xs = e_s
   and c is a clause containing a boolean expression of the form C_i ys = e_s,
   where $j \neq i$,
   then e_1 conflicts with c.

3. If e_i is an atomic boolean expression of the form ~ (C_i x_s = e_s)
   and c is a clause containing a boolean expression of the form C_i ys = e_s,
   then e1 conflicts with c.

4. If e_i is an atomic boolean expression of the form C_i $_{xs}$ = e_s
   and c is a clause containing a boolean expression of the form ~ (C_i y_s = e_s)
   and e_i does not conflict with c,
   then c' defined by removing ~ (C_i y_s = e_s) from c is the *subsumption* of c.

5. If e_i is an atomic boolean expression of the form ~ (C_i x_s = e_s)
   and c is a clause containing a boolean expression of the form (C_i y_s = e_s)
   and e_i does not conflict with c,
   then c' defined by removing (C_i y_s = e_s) from c is the *subsumption* of c.

6. If e_i is an atomic boolean expression that does not conflict with c,
   and e_i does not occur in c,
   then c is the trivial *subsumption* of e_i and c.

7. If e_i is an atomic boolean expression that does not conflict with c,
   and e_i does occur in c,
   then c' defined by removing e_i from c is the *subsumption* of c.

We define simplification (w.r.t an atomic boolean expression e) of a formula f by first removing all clauses c in f that conflict with e, then taking the subsumption of each resulting clause (w.r.t. e).

Main Branching Tree Algorithm

Given all of this machinery, we can finally implement the main bt (branching tree) algorithm.

The algorithm takes as input:

1. A formula form in DNF (initially f)
2. A set of undefined local variables vars (initially vs)
3. A collection of *assumptions* (initially true), expressed as a formula assumptions in DNF.

The algorithm produces as output.

1. A boolean Metaslang term tm.
2. A boolean term fail in DNF.

The correctness condition for this specification is

~ fail && assumptions => tm => form

Ultimately, ~fail is the calculated precondition for the entire merged ruleset.

The algorithm is defined recursively on the structure of form:

1. If form is false,
   then tm is false and fail is assumptions.

2. If form contains a clause that is the singleton true,
   then tm is true and fail is false.

3. If each clause in form contains a constraint v = e,
   and the intersection of vars
   and the variables in e that the constraint depends on is empty,
   then tm is v = e && t',
   where t' is the result of a recursive call to bt
   with form the formula resulting from removing v = e from each clause.
   fail is the resulting fail value from the recursive call.

4. If each clause in form contains a definition (v1 ... vn) = e,
   and the intersection of vars
   and the variables in e that the constraint depends on is empty,
   then tm is ex (v1, ..., vn) (v1 .. vn) && t',
   where t' is the result of a recursive call to bt
   with form the formula resulting from removing (v1 ... vn) = e from each clause
   and inputs the result of removing v1 ... vn from inputs.

5. Case split...

6. If there exists a branch expression e that occurs in the formula,
   then tm is if e then u else v, and fail is ufail || vfail in DNF.
   The term u is the result of a recursive call to bt,
   extending the assumptions with e and simplifying the formula w.r.t e,
   yielding ufail for failure conditions.
   Similarly, the term v is the result of a recursive call to bt,
   extending the assumptions with ~e
   and simplifying the formula w.r.t ~e, yielding vfail for failure conditions.

A simplified presentation of the mergeRules algorithm is as follows:

```
op BT (rs: Set Rule | rules simplified wrt preconditions and context): AST =
   if empty rs  then <generate precondition from path>

   else if True in? rs then done

   else if False in? rs  then BT(rs -- {False})

   else if there is a nondependent conjunct C in each rule in rs,
         with no unbound vars, and no mention of post-state
     then probable failure

   else if there is a nondependent conjunct C in each rule in rs,
         with no unbound vars, and mention of post-state
     then output C && BT( simplify rs assuming C ))

   else if there is a nondependent conjunct C in each rule in rs,
         with bindable vars and no mention of post-state
     then output ex(Vars(C))( C && BT( simplify rs assuming C ))

   else if each rule in rs contains a nondependent conjunct
         of the form c = e where c is a constructor from the sum-type  c1|...|cn
     then output case e of
                  | c1 → BT( simplify rs assuming c1=e)
                  | ...
                  | cn → BT( simplify rs assuming cn=e)

   else if there are nondependent conjuncts C and ~C in rs
         (ex(C) ex(r1,r2:Rule)(r1 in? rs && r2 in? rs => C in? r1 && ~C in? r2))
     then output if C then BT( simplify rs assuming C )
                          else BT( simplify rs assuming ~C )

   else if there is a nondependent conjunct C in the normal-case rule in rs
```

```
    then output if C then BT( simplify rs assuming C )
                 else BT( simplify rs assuming ~C )

  else if there is a nondependent conjunct C that occurs multiple times in rs
    then output if C then BT( simplify rs assuming C )
                 else BT( simplify rs assuming ~C )
```

3.6.3 Observer Maintenance Transformation

The Observer Maintenance transformation is a mechanism for introducing observers. The arguments to the transformation include a new observer of state and an invariant that characterizes the observer in terms of the current value of other observers.

The Observer Maintenance transformation is applied to an observer, say

obsE: State → E,

that we desire to maintain incrementally rather than compute on-demand. The performance improvement comes from a space-time tradeoff: we store the incrementally computed value of the observer obsE so that, when needed, we can simply access its value (knowing that the stored value equals the defined value).

In the context of a derivation, the idiom is that we introduce a fresh observer and its definition

```
op obsE(st:State):E = (e st).
```

Rather than manually enter the coinductive constraints that assert how each transformer affects this observer, we wish to use the definition to automatically calculate those constraints and add them. In a Specware metaprogram/derivation-script, we write

```
transform S by {maintain(obsE), ... other transformation cmds}
```

The Observer Maintenance transformation performs the following steps:

1. for each undefined transformer

```
op t(st:State | pre st)(args:Args):{st':State | post st args st'}
```

augment its pre- and post-conditions with the obs invariant as follows:

```
op t(st:State | pre st && obsE st = e st)(args:Args):
    {st':State | post st args st' && obsE st' = e st'}
```

2. apply simplification rules in context to normalize it to the coinductive form

```
obsE st' = delta st' (obsE st) )
```

for some function delta:State->E->E}.

3. refine the specification of t to

```
op t(st:State | pre st && obsE st = e st)(args:Args):
    {st':State | post st args st' && obsE st' = delta st' (obsE st)}
```

Note: Step 1 applies only to transformers that are specified but do not yet have a definition. Once a transformer has a definition in terms of other transformers, then the definition body presumably maintains the observer invariant by construction.

In network protocol derivations, the observer maintenance has a variety of uses. One is to incrementally maintain the stale flag for a mailbox.

3.6.3.1 Example: Maintaining a stale flag

In implementing a communication channel, it is usually important to detect when a fault or attack has happened. In the case of a mailbox, it is possible that the sender or the channel have failures that push their behavior outside their specified behavior. One approach is to inject code to detect failures, so the system can notify a user, or take other mitigation steps to adapt to the failure.

The freshness requirement on the mailbox communication allows us to derive its negation, which is a staleness condition. When we localize the staleness condition to the Subscriber process, it expresses that the last event on the inbox was a read (i.e. not a write), hence the value in the mailbox is not fresh. We show next how a formal characterization of staleness can be maintained using the Observer Maintenance transformation, leading to the injection of code to maintain a stale observer as a flag.

From the freshness requirement, we can infer the definition

```
op stale (st:State): Bool  = Read? (last (hist st)).action
```

and add the transformation

```
maintain(stale)
```

in the metaprogram/derivation. The effect of the transformation is to assert the definition as an invariant of each transformer (by adding it to the precondition and postcondition), followed by simplifications to obtain fast enforcement code to maintain the value of stale. For example, the read transformer is refined to

```
  op read(st:State | stale st = Read? (last (hist st)).action):
          {(st',pld):State*Payload
          | hist st' = (hist st) <| {state=st, time=clock st, action = Read pld}
          && pld = readMB st
          && stale st' = Read? last (hist st').action}
```

Note that the refined postcondition now enforces the invariant by ostensibly recomputing the definition of stale in the new state st'.

We then simplify:

```
Assume:  stale st = Read? last (hist st).action
         (st',pld) = read st
         hist st' = (hist st) <| {state=st, time=clock st, action = Read pld}
Simplify:  stale st' = Read? last (hist st').action
```

Calculation:

```
        stale st' = Read? (last (hist st')).action

         = { unfold (hist st') }

        Read? (last (hist st) <| {state=st, time=clock st, action = Read pld}).action

         = { last lst <| a = a }

        Read?  {state=st, time=clock st, action = Read pld}.action

         = { project }

        Read? (Read pld)

         = { eval }

        true.
```

That is, when we read from the mailbox, we concurrently set the stale flag to true.

The refined op spec is

```
  op read(st:State | stale st = Read? (last (hist st)).action ):
```

```
        {(st',pld):State*Payload
        | hist st' = (hist st)  <| {state=st, time=clock st, action = Read pld}
        && pld = readMB st
        && stale st' = true }
```

The transformation performs an analogous refinement of the Recv transformer:

```
  op recv(st:State,:ChannelId | stale st = Read? (last (hist st)).action ):
        {st':State
        | hist st' = (hist st) <| {state=st, time=clock st, action = Recv pld}
        && stale st' = Read? last (hist st').action}
```

We then calculate more efficient code to enforce the invariant:

```
Assume:  stale st = Read? last (hist st).action
          (st',pld) = read st
          hist st' = (hist st) <| {state=st, time=clock st, action = Recv pld}
Simplify:  stale st' = Read? last (hist st').action
```

Calculation:

```
      stale st' = Read? (last (hist st')).action

       = { unfold (hist st') }

      Read? (last (hist st) <| {state=st, time=clock st, action = Recv pld}).action

       = { last lst <| a = a }

      Read?  {state=st, time=clock st, action = Recv pld}.action

       = { project }

      Read? Recv pld}

       = { eval }

      false.
```

That is, when we recv from the mailbox, we concurrently set the stale flag to false. The refined op spec is

```
  op recv(st:State | stale st = Read? (last (hist st)).action):
        {(st',pld):State*Payload
        | hist st' = (hist st) <| {state=st, time=clock st, action = Recv pld}
        && pld = recvMB st
        && stale st' = false }
```

3.6.3.2 Related Work

The observer maintenance transformation builds on earlier work on strength reduction in compilers, finite differencing [Paige82, SmithD9009], and incrementalization [Liu13]. These previous transformations work by looking up the update code from pre-computed tables. Consistent with our generalization of Paige's Finite Differencing transformation [SmithD9009], we allow the maintenance of invariants over user-defined vocabulary, since we calculate the update code in the context of the application domain theory; that is, we use the axioms and theorems of the domain as part of the calculation of update code. Observer maintenance can be viewed as an adaptation of our generalization of finite differencing to coalgebraic specifications.

One point is that there is no need for intricate problem-specific conceptualization and ad hoc reasoning during design – the design concepts and inferences are generic in their outline and are only problem-specific in that they rely on problem-specific requirements/goals and problem-specific axioms and theorems. That is, the designs are generic but tailored by generic inference patterns to the specified problem.

3.6.4 StructureEx Transformation

The application of the mergeRule transformation, presented in Section 3.6.2, results in existentially quantified expressions. The structureEx transformation eliminates existential quantifiers in favor of let-bindings and substitutions. It plays a crucial role in translating logical postconditions into a more functional form. We developed this transformation and made many extensions to handle new cases that arose in our derivation.

3.6.5 FinalizeCotype Transformation: Cotype Definition and Postcondition Synthesis

During a derivation, we typically introduce a cotype without a definition, but add observers to it in subsequent refinement steps.

Observers at any stage in the refinement process come in several flavors. Some observers have a definition (and so they are eagerly computed when needed). Some are undefined but are specified by their effect on various transformers. Some observers have an invariant characterization and are incrementally computed via the Observer Maintenance transformation. Some are ghost observers and therefore have

no effect on computation, since they exist solely to increase the precision of system properties.

The finalizeCotype transformation is a packaging of two related transformations: cotype definition and postconditions synthesis.

3.6.5.1 Transformation: Cotype Definition

The cotype definition transformation introduces a definition for the cotype as a tuple, or record named fields. It works by collecting the undefined observers that are not ghosts and making them the fields of the tuple. It then gives a definition to each observer as a field access to the local cotype element (commonly the state).

Refinements of a coalgebraic specification correspond to subclassing in an object-oriented setting. If we refine a spec Sspec introducing cotype S to a spec TSspec that introduces additional observers and transformers on S, then any S operator can be applied to any T object.

As an example, consider a cotype Packet for which we want to generate formats for a family tree of Packet types/classes. We do so by introducing alternative sets of observers of packets and their parts.

```
BasicPacket = spec
  cotype Packet

  cotype Metadata
  op metadata : Packet → Metadata

 type Payload
 op payload : Packet → Payload
end-spec
```

The idea in BasicPacket is to introduce the data content of a packet as an unspecified Payload observation, and to introduce information about the payload (such as length, source, destination, format information, etc.) as a Metadata observation. Alternative refinements of BasicPacket can equip the cotype with different sets of observers. Later in the derivation the metaprogram can apply finalizeCotype to create record structures that define the various members of the Packet family.

We present two alternative refinements of BasicPacket that are simplified versions of the Packet family generated for the RADL instance running on the Landshark and ABA.

The spec IPPacket extends BasicPacket with observers that will become the fields of an IP packet:

```
IPPacket = spec                abridged set of observers of an IP packet
    import BasicPacket
    type IPAddr = Nat32

   op pktLen          : Packet → Nat16
   op pktID           : Packet → Nat16
   op pktFlags        : Packet → Nat3
   op pktTTL          : Packet → Nat8
   op pktProtocol     : Packet → Nat8
   op pktChecksum : Packet → Nat16
   op pktSrcIP    : Packet → IPAddr
   op pktDstIP    : Packet → IPAddr
end-spec
```

The following specification is a simplified specification for the message information that flows in a RADL system.

```
RADLPacket = spec
   import BasicPacket
   op channel_id: Packet → Nat4          4 bits
   op data_size  : Packet → Nat16        16 bits
   op buffer_size: Packet → Nat16
end-spec
```

We implemented the following syntax for defining a cotype in a Specware tactic/metaprogram:

transform S by { finalizeCoType(*cotype*, *list of observers*)}

The transform analyzes the spec S and produces a refined specification that gives a record definition to the argument cotype with the specified list of observers in the given order. We found the need to add the second argument (list of observers) since as we found more kinds of observers (ghost observers, defined observers, maintained observers), it became more complex to define a rule that pulled out exactly the observers that we wanted to package up. Another issue is the order of fields in the defined record and whether padding is needed for address alignment.

For example,

```
transform S by { finalizeCoType(RADLPacket,
                                 [channel_id, buffer_size, data_size, payload])}
```

introduces the following type definition in S

```
type RADLPacket =
        { channel_id : Nat4,
          buffer_size : Nat16,
          data_size   : Nat16,
          payload     : Payload
        }
```

The cotype definition transformation also gives definitions to the observers that are packaged up in the record:

```
op channel_id(pkt: Packet): Nat4   = pkt.channel_id
op buffer_size(pkt: Packet): Nat16 = pkt.buffer_size
```

and so on. The transformation also unfolds calls to them everywhere, eliminating them as functions. For example

```
op send (st:State) (chId:Nat4)  (pyld:Payload)
        {st':State |  hist st' = hist st <| {state=st, time=clock st, action= Send(chId, pyld)}
                   && outbox st' = pyld}
```

3.6.5.2 Synthesize transformers from postconditions

The second part of the finalizeCotype transformation synthesizes definitions for each transformer. It does so by translating the coinductive constraints in the postconditions of transformers into updates of the newly-introduced cotype record. For example, after a record definition has been given to State, then the recv transformer can be given a definition based on its postcondition:

```
op recv(st:State | stale st = Read? (last (hist st)).action):
        {(st',pld):State*Payload
        | hist st' = (hist st) <| {state=st, time=clock st, action = Recv pld}
        && pld = recvMB st
        && stale st' = false }
 = st << {pld = recvMB, stale = false}
```

The right-hand sides of the equations are evaluated first, and then the changes are made. Note that this transformation introduces a functional definition of the state change – it computes the new state as a function of the old/input state. A later code generation transformation will translate the body into C assignment statements to the variables for pld and stale.

3.6.6 Globalize Transformation

We extended a transformation that performs *Globalization*. Its effect is to transform the implicit state in a coalgebraic specification to explicit global/shared state. It allows us to generate truly imperative code.

Globalization can be described via the following abstract example. Here the cotype State has been defined as a pair and the observer c and transformers f and g are single-threaded on State; i.e. they take State as input and produce a State as output.

```
S = spec
 type State = {a:A, b:B}
 op c(st:State):C = h(st.a, st.b)
 op f(st:State)(arg:Arg): State =
    st << {a = alpha st.a, b = beta st.b}
 op g(st:State)(arg:Arg): State*D =
    (st << {a = gamma st.a, b = delta st.b},
     eps (st.a) (st.b))
end-spec
```

We implemented the following syntax for globalizing a cotype in a Specware tactic/metaprogram:

```
transform S by {globalize(State)}
```

The Globalization transformation on a cotype State requires that the type be single-threaded; i.e. such that there can be no two elements of the type simultaneously live during execution. Single-threadedness can be detected statically, however, the finalizeCotype transformation produces single-threaded definitions, and so it provides suitable input to the Globalization transformation.

Since Specware's Metaslang language is functional, and has no notion of state, the Globalization transformation necessarily is a step from Metaslang toward an imperative language, CommonLisp and C in our case. Its steps are to

1. Introduce a global variable of the cotype, say, var st:State.
2. For each observer and transform, eliminate State as an explicit parameter and return value, and replace local references to state by global references.
3. Replace record updates of the cotype by assignments

Shown in a pseudo-imperative notation, the effect of Globalization on S is

```
type State = {a:A, b:B}
var st:State

op c():C = h(st.a, st.b)
op f(arg:Arg): Unit =
   (  st.a := alpha st.a
   || st.b := beta st.b )

op g(arg:Arg): D =
   (  st.a := gamma st.a
   || st.b := delta st.b
   || return (eps (st.a) (st.b))
   )
```

where we use the notation (P1 || P2 || … ||Pn) to denote an atomic region in which codes P1, P2, …, Pn execute in parallel. The effect of Globalization is to introduce a global variable st of cotype State, and all accesses to st are now to the global (versus access to the parameter as before the transformation) and changes to fields of State are via destructive assignment rather than functional copy&modify. Our ad-hoc notation here treats concurrent assignment statements in an atomic region, in order that invariants are not observed to be violated.

The correctness of this transformation is straightforward in a sequential imperative language. The proof that this transformation works for distributed applications depends on assume-guarantee reasoning. We did not work on extending Globalize to emit proofs since the proof-emitting C generator discussed later in Section 3.9 was intended to subsume this transformation.

3.7 Proof Emitting Transformations

Proof-emitting transformations was a key innovation that we developed in the HACMS project. Figure 5 illustrates our approach to proof-emitting transformations and their role

in generating refinements and proofs. The figure depicts the action of applying transformation T to specification A. The result is a generated refinement from A to B, represented by morphism σ. Additionally, the transformation generates a proof term that can be used to discharge the proof obligation of the refinement.

Conceptually, we treat a transformation as a mapping from a specification A to a triple that includes (1) a specification morphism σ, (2) the refined/target specification B, and (3) a proof term. The proof term is a summary of the calculations performed in generating B from A. Specware provides a general proof-obligation-generator utility that maps a specification morphism, such as σ, to a Metaslang formula that expresses its proof obligations (i.e. that the axioms of B imply the axioms of A modulo the translation induced by morphism σ). The intent of the proof term generated by a transformation is that it can discharge the proof obligations of the generated morphism.

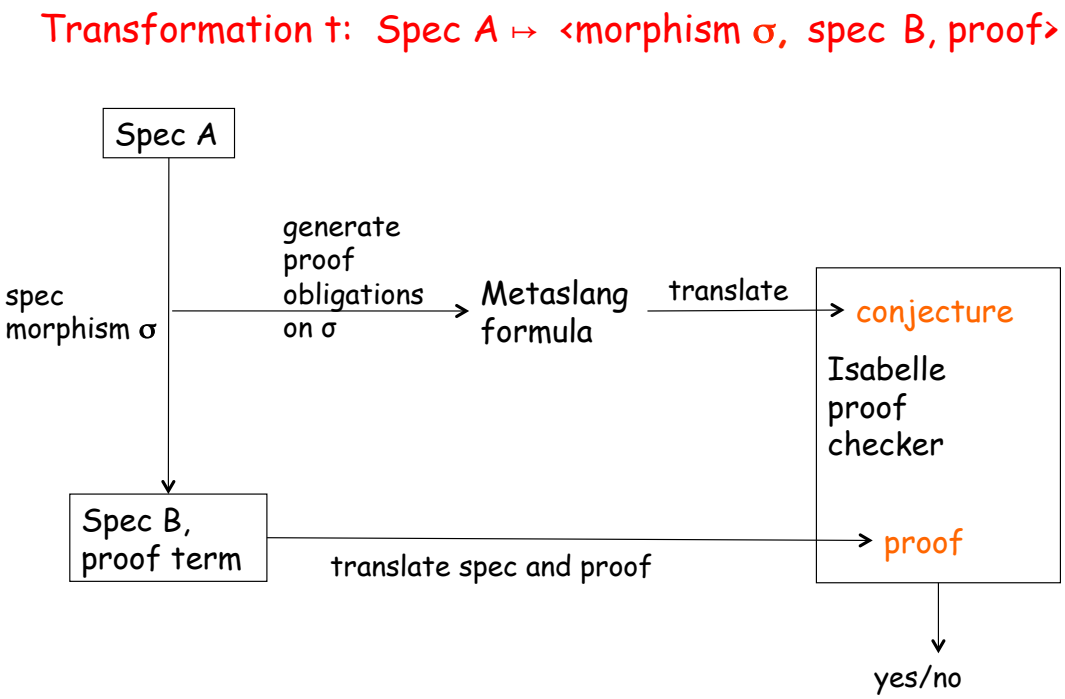

Figure 5: Proof-emitting Transformation

The goal here is to have an independent proof-checker verify that the proof term generated by the transformation does indeed prove the obligations generated by the proof-obligation-generator. One feature of the structure of Figure 5 is that the left-hand side is independent of the proof-checker. We wanted the freedom to build translators to any proof-checker that was rich enough to express the Metaslang logic. As an independent proof-checker we chose Isabelle since we already had a partial translator from the Metaslang logic of Specware to the Isabelle/HOL logic. The following

subsections describe several of the issues that arose in realizing this overall approach to generating proof-carrying code.

To be clear, Specware's transformations automatically carry out the calculation, producing a refined specification together with a checkable proof of the correctness of the refinement. This means that each generated refinement also has a generated proof that discharges its obligations, without having to perform a post-hoc proof search. The Isar proof script is formulated to put Isabelle on a very tight leash – its proof steps are tightly controlled, so that it will not get in trouble by attempting to search. After all, the transformation knows the structure of the calculation, so that is reflected in the proof script. We believe that this approach to proof generation will be dramatically more economical than post-hoc verification.

3.7.1 Instrumenting transformations to record calculation chains

We extended many of our library transformations to generate proof terms. During the course of the project we tried a sequence of approaches to the structure of the proof terms. Our first attempt was to record the sequence of equations used in a rewrite rule-based simplification. This was sufficient for several transformations, but could not handle proofs involving recursive transformation of terms. Our second approach was to define transformation-specific datatypes to record transformation steps. After instrumenting several transformations this way, it became clear that there were many commonalities and we felt the need (and possibility to define) a uniform representation of calculations performed by transformations. Our third approach was to develop a uniform proof representation for all transformations. A portion of the definition of our proof term specification is:

```
type ProofInternal =
   | Proof_UnfoldDef (MSType * QualifiedId * MSVars * MSTerm * MSTerm)
   | Proof_EqSubterm (MSTerm * MSTerm * MSType * Path * ProofInternal)
   | Proof_EqTrans (MSType * MSTerm * List (ProofInternal * MSTerm))
   | Proof_ImplTrans (MSTerm * ProofInternal * MSTerm * ProofInternal * MSTerm)
   | Proof_ImplEq ProofInternal
   | Proof_Cut (MSTerm * MSTerm * ProofInternal * ProofInternal)
   | Proof_ImplIntro (MSTerm * MSTerm * String * ProofInternal)
   | Proof_Assump (String * MSTerm)
   | Proof_ForallE (Id * MSType * MSTerm * MSTerm* ProofInternal * ProofInternal)
   | Proof_EqTrue (MSTerm * ProofInternal)
   | Proof_Theorem (QualifiedId * MSTerm)
   | Proof_Tactic (Tactic * MSTerm)
```

where, for example,

- Proof_UnfoldDef (T, qid, vars, M, N) is a proof that fa(vars) M=N at type T by unfolding the definition of qid,
- Proof_EqSubterm(M,N,T,p,pf) is a proof that M = N : T from a proof pf : M.p = N.p, where M.p is the subterm of M at path p
- Proof_EqSym(pf) is a proof that N=M from pf : M=N

and so on.

To give a sense of the details, consider the following rewrite steps performed in one of the HACMS derivations:

```
{ 1: allOutNodes_of_addSupply }
      allOutNodes (addSupply H nid) (Set.set_insert(nid, black H))
—-> allOutNodes H (Set.set_insert(nid, black H))
{ 2: distribute_allOutNodes_over_set_insert }
       allOutNodes H (Set.set_insert(nid, black H))
—->  allOutNodes H (black H) \/ outNodes H nid
{ 3: Set.associative_union }
        roots H \/ (allOutNodes H (black H) \/ outNodes H nid)
—->  (roots H \/ allOutNodes H (black H)) \/ outNodes H nid
```

which is stored as the following proof term

```
EqTrans(Bool,
      roots H \/ allOutNodes H (black H) \/ outNodes H nid,
         [Sym(Theorem(Set.associative_union,
                  roots H \/ (allOutNodes H (black H) \/ outNodes H nid)
                    = roots H \/ allOutNodes H (black H)  \/ outNodes H nid)),
                roots H \/ (allOutNodes H (black H) \/ outNodes H nid),
       EqSubterm(roots H \/ (allOutNodes H (black H) \/ outNodes H nid),
                roots H \/ allOutNodes H  (Set.set_insert(nid, black H)), Bool, [1],
                      Sym(Theorem(distribute_allOutNodes_over_set_insert,
                                allOutNodes H  (Set.set_insert(nid, black H))
                                 = allOutNodes H (black H) \/ outNodes H nid))),
              roots H \/ allOutNodes H (Set.set_insert(nid, black H)),
       EqSubterm(roots H  \/ allOutNodes H
                     (Set.set_insert(nid, black H)),
                roots H   \/ allOutNodes(addSupply H nid)
                     (Set.set_insert(nid, black H)), Bool, [1],
                Sym(Theorem(allOutNodes_of_addSupply,
                          allOutNodes(addSupply H nid) (Set.set_insert(nid, black H))
                           = allOutNodes H  (Set.set_insert(nid, black H))))),
                roots H \/ allOutNodes(addSupply H nid)  (Set.set_insert(nid, black H))])
```

and is then rendered as a Isabelle/Isar proof script:

```
have subeq100:
    "(roots H \/ allOutNodes H (black H)) \/ outNodes H nid

      = roots H  \/ allOutNodes (addSupply H nid) (Set__set_insert(nid, black H))"
proof -
  have "(roots H \/ allOutNodes H (black H)) \/ outNodes H nid
        = roots H  \/ (allOutNodes H (black H)  \/ outNodes H nid)"
    proof -
      have symeq95:
           "roots H \/ (allOutNodes H (black H)   \/ outNodes H nid)
         = (roots H \/ allOutNodes H (black H)) \/ outNodes H nid"
        proof -
          show "?thesis" by (auto simp only: Set__associative_union)
        qed
      show "?thesis" by (rule symeq95[symmetric])
    qed
  also
  have "... =  roots H  \/ allOutNodes H (Set__set_insert(nid, black H))"
    proof -
      have subeq97:
           "allOutNodes H (black H) \/ outNodes H nid
             = allOutNodes H (Set__set_insert(nid, black H))"
        proof -
          have symeq96:
               "allOutNodes H (Set__set_insert(nid, black H))
                  = allOutNodes H (black H)   \/ outNodes H nid”
            proof -
              show "?thesis"
                by (auto simp only: distribute_allOutNodes_over_set_insert)
            qed
          show "?thesis" by (rule symeq96[symmetric])
        qed
      show "?thesis" by (rule arg_cong[OF subeq97])
    qed
…
  finally (HOL.trans)
  show "(roots H \/ allOutNodes H (black H)) \/ outNodes H nid
        = roots H \/ allOutNodes (addSupply H nid) (Set__set_insert(nid, black H))" .
qed
```

3.7.2 Translator from Metaslang logic to Isabelle logic

We extended an existing partial translator from Metaslang to Isabelle for two purposes. One was to translate refinement obligations and the other was to translate our proof terms into proof scripts that could be checked against the translated proof obligations.

Many aspects of the translation between these two higher-order logics were straightforward. However, completing this translator turned out to be trickier and take longer than expected. One key issue was translating Metaslang specs into Isabelle specs, and a special case is the translation of Metaslang formulas to Isabelle formulas. This was a source of ongoing difficulties since the Metaslang and Isabelle logics are similar but have many detailed differences. We worked on resolving two such differences: since Isabelle does not support predicate subtypes (including dependent types), we need to include the predicates from such types into the translation of a Metaslang expression, typically as an antecedent. We explored several variants of whether the antecedent should be normalized to the top level, or kept locally to preserve structure.

Another difference is that Isabelle does not support a name translation operation, while it is a basic operation on Metaslang specifications. This is a difficult feature to handle since the name translation must be applied recursively through the entire import structure of a specification. We completed work on handling the name translation operation on the Metaslang side. Since Isabelle doesn't have this feature, our translator from Metaslang to Isabelle had to perform a recursive copy-and-modify on the entire import structure of a specification and pass the whole structure to Isabelle, rather than appealing to Isabelle built-in specifications. This work was a part of the larger goal of supporting the generation of proofs that discharge automatically generated proof obligations for refinement steps. We continued to work on extending the Observer Maintenance and Observer Implementation transformations to emit proofs at application time.

Another difference: We worked on improving the generation of Isabelle proofs from transformation sequences. The proofs include references to particular subterms that are transformed. These are indicated by their path from the root of the term. However, translation to Isabelle does not always preserve the term structure. In particular, in quantified expressions subtype predicates are added which can complicate the subexpression paths. We made the translation more robust by exploiting the fact that these predicates are always conjoined at the beginning of a sub-formula. We made changes to our translator from Specware logic to Isabelle logic to reflect the use of named predicate subtypes – previously the translation was losing the predicate subtype, thereby causing some proofs to fail.

Another difference/change: We worked on supporting type refinement such as occurs during the finalizeCotype transformation where a previously abstract type is refined to be a record type. Isabelle requires that type symbols and their definitions be introduced at the same time, which a refinement system like Specware does not. To support this we introduced a transformation to explicate the previously implicit morphism that arises when a type symbol is defined later than its introduction in a Specware spec. The morphism is between the spec with the abstract type and the spec with the defined type. The obligations of the morphism are that the axioms on the abstract type are theorems on the defined type. In the case of finalizeCotype, the relevant axioms are that the postconditions of the state transformers are true given their preconditions. These pre- and postconditions are preserved in the final spec so the obligations are trivially true. The finalizeCotype transformation also provides bodies for the functions specified by pre- and post-conditions, so we also have the obligation that the bodies satisfy postconditions given the pre-conditions, which follows simply given that the bodies are mechanically derived from the postconditions.

We also revised our approach to a key problem in generating Isabelle proof scripts. The problem has to do with the straightforward notion of substitutivity:

if x=y then f(x)=f(y)

When we transform an expression f(x) by simplifying its subterm x to y, then we want a proof that f(x)=f(y). The problem has been identifying to Isabelle which subterms x and y are equal, since the paths to the subterms are typically modified during our translation from Specware to Isabelle. Previously we had been explicitly giving the context/path to x by means of a lambda

lambda(v)f[v]

to indicate the hole where the subterm x occurs. We found a simpler solution in using the argCong mechanism of Isabelle, which automatically searches for the subterms x and y and then infers the desired result f(x)=f(y). This helps in pushing through the proofs emitted by our finalizeCotype transformation.

Another problem has to do with the handling of conditional rewrites. In addition to conveying the condition of the rewrite to Isabelle, sometimes the variables are quantified over a subtype, so the subtype effectively becomes an additional condition. We extended our translation mechanisms accordingly.

Another problem arises due to the use of speculative rewriting in the rewrite engine. Some rewrites may not improve the code so they are applied speculatively, and if they

do not enable an improvement, then they are withdrawn and rewriting continues. Obviously we do not want that backtracking reflected in the generated proof structure, so we added a mechanism to detect backtracking and to produce a proof script reflecting the actual path to the transformed results.

Several other improvements were needed to our translation from Specware/Metaslang to Isabelle. First, the translate construct is used to rename symbols from an imported theory. For example, the theory of linear orders might have its type renamed time' in order to provide a simple appropriately named theory of time. The translate construct though caused an exponential blowup of copying in our previous implementation, so we needed to cache translated imports to avoid duplication. This problem only arose as we introduced a monad for formalizing the interleaving of threads that we need to specify and reason about the concurrent execution of mutator and collector. We also fixed errors in our spec-substitution construct, which was causing problems in translating proof terms from Metaslang to Isabelle. The solution was to apply substitutions to specs but not the spec-element terms, but instead to regenerate them, exploiting context.

We continued to develop and store proofs with theorems for the specs in the Specware library. The derivations invoke theorems from imported specs to perform rewrites and the generated Isabelle proof scripts depend on those library proofs.

3.8 Specware Infrastructure

We extended Specware's infrastructure in a number of directions to support the coalgebraic specifications and their refinement.

We improved Specware's transformation for Isomorphic Type-refinement, so that it handles patterns. We improved the rewriter's handling of curried functions. We modified Specware's type-checker algorithm to generate type-coercions, which means fewer proof obligations are generated. We improved printing of Specware specs and terms, and improved the efficiency of code generation for both in time and space usage. We extended the type-checker so it could infer tighter sub-types for the results of ops with specialized inputs. We added an indirection construct to the pragma language to allow proofs to be separated from specs, so the specs are more readable. We improved the proof obligation generated for a refined op so that it is easier to prove – making the obligation extensional and including subtype conditions of argument variables.

We implemented a version of function unfolding that works with functions specified using pre and post-conditions, by combining the postconditions. We also adapted the common expression abstraction tactic to work properly with assignment statements.

3.8.1 Higher-Order Matching Algorithm

We implemented a feature in the higher-order matcher where it avoids generating subgoals for a subtype mismatch that could be discharged by subtype obligations. Previously, if the term being matched had an associated subtype obligation, to show the rule matched, one would have to prove the obligation was true using the rewriter. This was at best inconvenient. Now we assume that obligations are proved in Isabelle.

We fixed type matching in Specware's higher-order matcher – a type variable is now bound to the least supertype of all the types it is matched against. We also made changes to the Isabelle translator since, in some cases, it was not extracting composite subtype predicates correctly for nested subtypes. We also needed to rationalize the ordering of the extracted predicates.

3.8.2 Support for calculation

Support for calculational inference was extended from equational to handle conditional equations and to handle strengthening of propositions (e.g. the Observer Refinement calculation above).

3.8.3 Tactic language

We also modified the transformation script language to make it simpler to read, write, parse, and automatically generate scripts. This has allowed us to reformulate several existing transformations into the following normal form: generate a derivation script and then run it. This normal form has several advantages:

1. it replaces the writing of arbitrary metaprograms that manipulate abstract syntax,
2. it extends the range of people who can write transformations
3. it prepares the ground for emitting proofs as a by-product of transformation.

We extended the scripting language to support verbatim text for generating into CommonLisp. This allows us to add Lisp-specific instrumentation, monitoring, and other support code as an integral part of the derivation script. We fixed the error handling for transformation moves that fail, so an error message is presented instead of going into the debugger.

3.8.4 Transformation Support

We improved Specware's transformation language machinery so that it is easier to add new transformations both for spec-level transformations and term-level transformations. Now, it is only necessary to define the transformation as a function with a suitable type, without having to add special interface code to the transformation engine.

We added support for user-defined transformations. Previously, adding a new spec transformation function would involve changes to the transformation language parser. We have now implemented a scheme whereby the signature of the transformation function determines the syntax in the transformation language. This makes it much easier for developers to incorporate new transformations into the transformation language, especially when the transformation has multiple options and lists of rewrite rules or functions as arguments. To implement this interpreter capability we had to augment the code generator to output type information for transformation functions so its arguments could be interpreted at run-time. As the interpreter has to work with objects of multiple types, we needed to tag values with their type and provide an interface to the transformation functions that accepts these tagged values.

The basis of the transformation system extension is to have the signature of the Specware transformation function determine the syntax of its use in the transformation language. For example,

```
op MSTermTransform.rewrite: Spec -> PathTerm -> RuleSpecs
                                    -> RewriteOptions -> MSTerm
type RewriteOptions =
  {trace    : Nat,        % Trace level 0, 1, 2, 3
   debug? : Bool,         % Debug matching of rules
   depth    : Nat}        % # of rewrites allowed
```

is the (slightly simplified) signature of a rewrite transformation that transforms the current term using a list of transformation rules and with three options. The "MSTermTransform." qualifier tells Specware that this is a term transformation. The spec and the term are implicit, i.e. given by the current transformation context. The syntax for using this in a transformation sequence is, for example:

```
rewrite [unfold open?, lr mapFrom_TMApply, lr filter_true]
      {trace = 2, debug?= true, depth = 5}
```

where *unfold open?* is the rule for unfolding the definition of open? and *lr thm* takes an equality theorem thm as a left-to-right transformation rule. The system allows for defaults everywhere so the options between braces can usually be completely omitted, or any subset can be specified. E.g.

```
rewrite [unfold open?, lr mapFrom_TMApply, lr filter_true]
```

or

```
rewrite [unfold open?, lr mapFrom_TMApply, lr filter_true] {depth = 5}
```

or just

```
rewrite
```

which just uses the built-in simplification included in the rewriter without any rewrite rules.

Previously, allowing all these syntactic options had to be specifically programmed, so changing an interface, in particular adding options, was a significant amount of work that required knowledge of the internals of the syntax system. Having the syntax automatically follow from the signature makes it easy for any Specware user to add new transformations or extend existing ones.

3.8.5 Tracing support

We also made improvements to the transformation system so that it prints out a much better focused presentation of its (mostly) equational calculations.

3.8.6 Specware Library

We coalesced several variants of specifications for finite sets, bags, lists, maps, stacks, as well as standard refinements of them. We extended the Specware specification libraries with more proofs of theorems, which are used to support calculations at program-synthesis time.

We improved the Specware DataStructures library, with an emphasis on pushing the proofs through Isabelle and fixing any issues revealed in the process. The DataStructures library defines and refines container data structures, including Sets, Bags, Maps, etc. We added many Isabelle proofs (including proving quite a few new, generally useful auxiliary properties). Perhaps the most interesting proofs were those justifying the correctness of the refinements (expressed as morphisms) of various

structures in terms of the others, many of which make heavy use of 'fold' operations. The library work is still ongoing, but the Sets and Bags libraries are now completely proven.

We also refurbished many of the specs in the HACMS repository, to make them compatible with the latest version of Specware and its libraries. We also worked on Specware documentation, testing, and miscellaneous maintenance tasks and improvements (e.g., modernizing the syntax of important specs).

3.8.7 Open-Sourcing Specware

We open-sourced Specware in 2015 using Github. Users can download it from https://github.com/KestrelInstitute/Specware. The README file guides users in the installation process. The Specware.org website contains a tutorial and documentation on the use of Specware.

3.9 Generator of imperative code

This section describes our working C generator based on Specware transformations. We made some efforts to extend it to emit proofs, but decided to pursue a different, more structured, approach in parallel. This new approach is described in Section 7. Unfortunately the new approach was not completed by the end of Phase 2, so we continued to use the older generator, which is described below.

Our C generator is based on a sequence of specification transformations that correspond to compiler passes and that are intended to be simple enough that we can augment them to emit proofs at application time. We completed transformations for linearizing nested terms in single-threaded state transformer definitions and related code needed to prepare for globalizing the single-threaded state in our coalgebraic operations. We made numerous other internal improvements. We worked on issues related to handling pattern-matching in the compiler – since C doesn't support patterns for de/construction, there is no direct translation of this feature of MetaSlang, so special control mechanisms are needed to handle matches that partially succeed before failure.

We extended earlier work to propagate type information through our abstract syntax trees so that ambiguous constants (e.g. 1 can be a signed or unsigned 16 or 32 bit integer) can be consistently typed when passed to C. We made many internal improvements in support of C generation. We worked to generalize and clean up the transformation sequence that generates C (about 23 transformations), and to develop a compilation specification that allows expressing some C-specific information: import

files, native library types and functions that are used in the MetaSlang specification, translation of field names, and any special-case definitions. During this period we were able to generate, compile, and run idiomatic C code on some sample specifications written in our mixed algebraic/coalgebraic style.

Language Morphisms

Language morphisms are a generalization and formalization of what had been ad hoc features for translation to Isabelle and Haskell. Special translate pragmas within a spec can now be used to define language-specific rules for translating Specware types and ops. These pragmas now have an internal structure that is parsed in a very generic manner to obtain five kinds of information:

Imports

This section simply lists a sequence of files to be imported into the generated target file. For example, a translation to C might include:

```
-import
  stdlib.h    % boilerplate
  linux/udp.h % structures specific to UDP protocol
  mycode.h    % interfaces to ad-hoc application-specific code
```

Verbatim

This section is intended to be used sparingly, but provides an escape mechanism to insert arbitrary text verbatim into the target file. It is intended to handle ad hoc problems that resist a generic solution.

For example, the dereference operator * in C is a function and can be modeled relatively simply within Specware, but the address-of operator & is not a function since substitution of equals for equals fails. There thus is no simple way to target C expressions headed by &, but some special cases can be handled on an ad hoc basis by allowing Specware operators to map to C macros that include &. For example:

```
-verbatim
  #define atomic_read_at(x) (atomic_read(&x))
```

Verbatim text may create problems for verification, but it isolates such problems to a small set of clearly identified operators.

Also, because such verbatim text must appear declaratively within the specs being used, those that lack such tricks can be known to be free of such problems — there is no programmatic mechanism secretly including such tricks as part of the translation.

Translate

This is the main section, and provides for translation of Specware names (for types, ops, and field references) to target names or terms, along with an indication as to the location of the target (primitive/syntactic or file location). Translations to complex terms are implemented as target macros.

```
-translate
  type Nat.Nat32    -> uint32_t      primitive
  op Nat.BVAND32    -> & infix     primitive
  type udp_table    -> struct udp_table      in net/udp.h
  field udp_table.csum  -> udp_table.check   in net/udp.h
  op Null_ID      -> ((Sock_ID) NULL)    macro
  op sizeof_udp_hdr   -> sizeof (struct udphdr)  macro
```

Native

This section provides a simpler form of translation where the named type or op is assumed to translate directly to the same name in the target.

```
-native
 op ntohs in /drivers/staging/rtl8712/generic.h
 op udp_hdr in linux/udp.h
```

Language morphism pragmas for any given spec are collected recursively through all imported specs, making it possible to distribute the language-specific translation rules for types and ops into the local contexts where they are introduced or defined. Alternatively, the translations could be handled en masse by one pragma in the top-level spec, for example if one wished to have alternative top-level specs with different rules targeting different compilers.

Future work could easily validate that a type or op declared to be in a target file was at least nominally present there. With language-specific parsing of the target files. it would be possible to verify appropriate typing, etc.

Slices

A perennial problem with processing specs has been that each processing context may be concerned with just some aspects of a spec, requiring ad hoc code to determine which elements of the spec to process and which elements to ignore, making such processing fragile and hard to maintain as Specware evolves.

One aspect of this problem is that alternative notions such as defined, executable, implemented, primitive, hand-coded, etc. have tended to be conflated within such processing code, sometimes confusingly (and even inappropriately) using the same tests in contexts where slightly differing ones were needed.

There also were early attempts to create more manageable artifacts by simply subtracting out undesired portions of a spec, however this led to ill-formed specs that contained the information of interest but were missing semantically important theorems, subtype predicates, etc.

Slices provide progress towards a generic solution to this problem by layering filters over specs to provide ad hoc tailored views. They leave the spec itself unaltered but add tables describing which elements of the spec have various desired attributes. Each particular processing context can then view the spec through such a filter, simplifying the processing context while avoiding logical problems associated with ill-formed specs.

We continued to develop and store proofs with theorems for the specs in the Specware library. The derivations invoke theorems from imported specs to perform rewrites and the generated Isabelle proof scripts depend on those library proofs. In several instances we found the need to add conditions to theorems to enable proofs. This then requires ensuring that the mutator and collector operations have pre/post-conditions strong enough to discharge those new conditions.

4 RESULTS AND DISCUSSION

This section presents the results of our work in Phase 1 (synthesis of code for the UDP protocol) and in Phase 2 (synthesis of transport-level communication code for the RADL architecture).

4.1 Generating UDP handling code for the Linux Kernel

In Phase 1, we decided to focus our efforts on the UDP transport-layer protocol. It is relatively simple, providing no guarantees of reliable delivery, but it is efficient and useful for applications such as video streaming. The simplicity of the protocol allowed us to make a first pass at the whole synthesis process: formal specification, transformations to generate refinements, generation of efficient C code, and co-generation of proofs.

Current implementations of UDP continue to suffer from a variety of common vulnerabilities. Here are just two recent Linux-related CVE entries out of hundreds:

- CVE-2015-8605: UDP payload length not properly checked, allowing remote attackers to cause a denial of service (in Ubuntu before 4.3.3-P1)
- CVE-2015-5366: The (1) udp_recvmsg and (2) udpv6_recvmsg functions in the Linux kernel before 4.0.6 provide inappropriate -EAGAIN return values, which allows remote attackers to cause a denial of service.

Our goal was to generate C implementations of UDP code that would plug into the Linux kernel running on the Landshark and ABA. The second key decision was to base our formal specifications on the excellent work done at the University of Cambridge on a HOL specification of TCP/UDP/IP. That work, which we refer to as the NetSem project (simply, NetSem), focused on a transition system model that conforms to relevant RFC's as well as the de facto standards provided by the BSD, Linux, and Windows implementations of the TCP/UDP/IP protocol stack. To our knowledge, the HOL specification provides the most comprehensive, precise, and validated model of the protocol stack, so we used it as a starting point and reference standard. Our project has different goals than NetSem, since our goal was to develop a specification of protocol requirements and to refine those specifications to efficient, usable code via mechanized transformations. Also, rather than performing post hoc proofs of properties, our goal was to demonstrate the co-generation of code and proofs.

Specification

We first focused on 3 NetSem rules (udp_deliver_1/2/3) that capture the essential semantics of receiving a UDP datagram in Linux, specifically the udp_rcv operation.

The normal effect is to copy the datagram received off the wire to the input queue of the appropriate socket. The NetSem rules capture the normal case and four exceptional cases. All of these were formalized as cases (trace predicates).

The various cases of the Linux processing of UDP receive can be specified as follows

1. Normal case

```
dequeue_iq (iq h) = (iq h', Some (UDP dgram))
&& dgram.is1 = Some i3
&& dgram.is2 = Some i4
&& dgram.ps1 = ps3
&& dgram.ps2 = ps4
&& dgram.data = data
&& i4 in? local_ips (ifds h)
&& ~(is_broadormulticast (ifds h) i4)
&& ~(is_broadormulticast (ifds h) i3)
&& Some sid = lookup_udp (socks h) (i3,ps3,i4,ps4) (bound h) (arch h)
&& sock    = TMApply (socks h, sid)              % bind sock
&& sock.pr = UDP_PROTO (rcvq)                    % bind rcvq
&& rcvq' = rcvq <| Dgram_msg ( Some i3, ps3, data)
&& sock' = sock << {pr=UDP_PROTO (rcvq')}  % update socket component
&& socks h' = update (socks h) sid sock'          % update host components
&& oq h' = oq h
&& udp_host_frame_ax (h,h')
```

2a. Exception case: no local/destination socket

```
vars:  dgram, i3,i4,ps3,ps4,data, icmp
dequeue_iq (iq h) = (iq h', Some (UDP dgram))
&& dgram.is1 = Some i3
&& dgram.is2 = Some i4
&& dgram.ps1 = ps3
&& dgram.ps2 = ps4
&& i4 in? local_ips (ifds h)
&& ~(is_broadormulticast (ifds h) i4)
&& ~(is_broadormulticast (ifds h) i3)
&& None = lookup_udp (socks h) (i3,ps3,i4,ps4) (bound h) (arch h)
&& icmp = ICMP {is1 = Some i4,         % Create ICMP message
                is2 = Some i3,
                … }
&& (if icmp_to_go              % update host components
     then (oq h', true) = enqueue_oq(oq h, icmp)
   else oq h' = oq h )
&& socks h' = socks h
&& udp_host_frame_ax (h,h')
```

2b,c,d. Exception cases: bad packets are dropped

```
  dequeue_iq (iq h) = (iq h', Some (UDP dgram))
  && dgram.is2 = Some i4
  && dgram.ps2 = ps4
  && i4 in? local_ips (ifds h)
  && is_broadormulticast (ifds h) i4      % broadcast or multicast dest
  && udp_host_frame_ax (h,h')
  dequeue_iq (iq h) = (iq h', Some (UDP dgram))
  && dgram.is2 = Some i4
  && dgram.ps2 = ps4
  && i4 in? local_ips (ifds h)
  && dgram.is1 = None                          % no Datagram
  && udp_host_frame_ax (h,h')
  dequeue_iq (iq h) = (iq h', Some (UDP dgram))
  && dgram.is2 = Some i4
  && dgram.ps2 = ps4
  && i4 in? local_ips (ifds h)
  && dgram.is1 = Some i3
  && is_broadormulticast (ifds h) i3   % broadcast or multicast source
  && udp_host_frame_ax (h,h')
```

Applying the mergeRules transformation to these cases results in the following specification for the udp_rev transformer:

```
op udp_rcv(h:host):
   {h': host |
     iq h' = iq_rest(iq h)
     && (case iq_head(iq h)
          of Some (UDP dgram) ->
            case dgram.is1 of
              | None -> (case dgram.is2 of Some i4 -> i4 in? local_ips(ifds h))
              | Some i3 ->
                (case dgram.is2
                  of Some i4 ->
                     let ps3 = dgram.ps1 in let ps4 = dgram.ps2 in
                     let data = dgram.data in
                     i4 in? local_ips(ifds h)
                      && (if ~(is_broadormulticast(ifds h) i4)
                           then
                             if ~(is_broadormulticast(ifds h) i3)
                              then
                                case lookup_udp(socks h) (i3, ps3, i4, ps4) (bound h) (arch h)
                                 of Some sid ->
                                    (oq h' = oq h
                                     && (let sock = Map.TMApply(socks h, sid) in
                                         (case sock.pr
                                           of UDP_PROTO rcvq ->
```

```
                (socks h'
                 = Map.update(socks h) sid
                    (sock
                      << {pr =
                           UDP_PROTO
                             (rcvq <| Dgram_msg(Some i3, ps3, data))}))
               | _ -> false)))
        | None ->
          (socks h' = socks h
           && (oq h', true) = enqueue_oq(oq h, ICMP {is1 = Some i4,… }))
    else true
  else true)))}
```

The specification of UDP send code (udp_send) is much more complex. Our specification has some 37 rules partitioned into 11 modules, each representing a stage in transmitting a UDP datagram. Without partitioning into modules we would have had thousands of cases to handle.

Summaries of the metaprograms that we wrote to generate C code for udp_rcv and udp_send are shown in Figure 6 and Figure 7.

Initial Specification: disjunctive set of cases

MQ. mergeRules + QE for udp_queue_rcv_skb
MR. mergeRules + QE for udp_rcv
R1. introduce disjunctive definitions
OR1. Introduce definitions for observers
Cot1. finalizeCotype Host
Defs1. Introduce definitions for testing
Prep. Prepare for C generation
Cgen. Generate C code

Figure 6: Metaprogram for UDP receive

Initial Specification: disjunctive set of cases

MR1. mergeRules + QE for udp_sendmsg
…
MR11. mergeRules + QE for udp_sendmsg_out
OR1. Introduce definitions for observers
Simp Simplify expressions
Cot1. finalizeCotype Host
Defs1. Introduce definitions for testing
Prep. Prepare for C generation
Cgen. Generate C code

Figure 7: Metaprogram for UDP send

Specware executes the metaprograms automatically and generates C code that we inserted and tested in the Linux kernel. Figure 8 shows some statistics about the derivation of UDP processing code.

	udp_rcv lines of text	udp_sendmsg lines of text
domain specification	740	2200
metaprogram	500	1067
generated Metaslang	1025	2470
generated CommonLisp	2300	3000
generated C	220	955
Isabelle proof scripts	9300 lines, 350 proof steps	>20450 lines, 1500 proof steps

Figure 8: Statistics on the Derivation of UDP processing code

4.2 Composition of Protocols

In Phase 1 we began investigating the possibility of composing protocols out of reusable parts. As a simple first exercise, we specified a simplified version of the http protocol and an abstract scheme/template for command processing. We instantiated the scheme with the details of the http specification (via a pushout) and then optimized the result. This exercise showed that a simple scheme (with one normal case and one exception case) could instantiate to a system specification with two normal cases and five exception cases. This gave us confidence that an abstract scheme could cover the processing of complex protocols. We performed a case study on a simplified HTTP protocol, motivated by the fact that ROS uses xmlrpc and http on top of the TCP/IP stack.

Our general approach stems from the observation that we can think of protocols as providing the means for detecting and/or mitigating the effect of specific classes of faults and attacks that can arise in a real-world communication channel. See the figures below for tables of various faults/attacks and the means for detecting (Figure 9) and for mitigating (Figure 10) them.

Threat	Model	Detection Mechanism
packet corruption	nondeterministically bit flips	parity bit, checksum
dropped packets	nondeterministically drop packets	sequence numbers
out of order packets	nondeterministically reorder packets	sequence numbers
eavesdropping	copy packets to external agent	quantum-based methods?
spoofing	false sensor reads, faked sends	cross-checking, authentication

Figure 9: Threats and Detection Mechanisms

Threat	Model	Mitigation Mechanism
packet corruption	nondeterministically bit flips	Error-Correcting Codes
dropped packets	nondeterministically drop packets	ACK + timeout + retransmit
out of order packets	nondeterministically reorder packets	sequence numbers
eavesdropping	copy packets to external agent	encryption
spoofing	false sensor reads, faked sends	authentication, detect-and-drop

Figure 10: Threats and their Mitigation Mechanisms

Our goal was to abstractly and formally specify the various means for detecting and mitigating such faults. If a channel is expected to suffer several such classes of faults, then we wanted to be able to compose a fault model from the models of the individual faults. Moreover, and most importantly, if we had abstract mitigation schemes for each fault class, then we wanted to develop a mechanism for composing a complex protocol from those mechanisms that provides mitigation for the complex fault that the channel suffers. For example, if we are dealing with a channel that may corrupt individual bits, and may drop random packets, and we have mitigation means for addressing corruption and dropped packets, then we would like to compose those means into one protocol that approximates (or achieves) ideal communication over a channel that may corrupt bits and drop packets.

We also explored various ways to express the threat/fault/attack models and the corresponding mitigation mechanisms. Our first step was to formalize the fault/attack models. Our approach is to specify the channel in terms of an input/output relation on message streams. Here we abbreviate the ghost observer sent by s, and the ghost observer received by r:

1. A medium that can drop messages can be specified by a substream relation;

$$\text{Medium}_{\text{drop}}\ s\ r = (s \sqsupseteq r)$$

where $s \sqsupseteq r$ holds exactly when each message in r occurs once in s and the messages in r occur in the same order as in s.

2. A medium that can reorder messages can be specified by a bijective relation;

$$Medium_{reorder}\ s\ r = (s \longleftrightarrow r)$$

where $s \longleftrightarrow r$ holds exactly when there is a bijection between the messages in s and the messages in r.

If we want to have a medium that has several such properties it turns out we can compose its specification via relational composition (rather than some form of conjunction, our first thought). So a medium that can both drop messages and reorder them can be specified either as

$$Medium_{reorder\&drop}\ s\ r = (s\ (\longleftrightarrow\ ;\ \supseteq)\ r)$$

i.e. first reorder, then drop messages or

$$Medium_{drop\&reorder}\ s\ r = (s\ (\supseteq;\longleftrightarrow)\ r)$$

i.e. first drop messages, then reorder where (R;S) denotes the relational composition of relations R and S:

$$x(R;S)y = \exists z(xRz \wedge zSy).$$

We could write these definitions more simply as

$$Medium_{reorder\&drop} = (\longleftrightarrow\ ;\ \supseteq)$$

and in this case we actually have the nice law that the composition is commutative:

$$(\supseteq;\longleftrightarrow) = (\longleftrightarrow\ ;\ \supseteq)$$

so

$$Medium_{reorder\&drop} = Medium_{drop\&reorder}$$

Observation 1: we want to start with the assumption of a perfect medium (s=r) and then add progressively more realistic models of the medium. The transition from (s=r) to, say, ($s \supseteq r$) is one of generalization, so our approach is to work with successively more general media, each requiring progressively more complex mitigation mechanisms. The fact of commutative composition of some threat models, as above, suggests that the corresponding mitigation mechanisms may (or should) also be commutative.

Observation 2: It is straightforward to extend the composition mechanisms above to handle latency (cf. Section 3.4.4): the latency of the composition of two relations is the sum of the latency of individual relations.

Observation 3: Specifications of systems in general, and communication systems and their protocols in particular, have at least two levels of specification. One level is the Global Specification (aka god's eye view) which allows us to assert properties about all system observables, even though no component of the system has access to all such observables (directly). The other level is the Local or component view (aka worm's eye view), which asserts constraints over the observables that are accessible to a particular component. Actual component code must operate at the local level, but most safety and security properties are most naturally expressible at the Global level. The input/output relation for the medium is expressed at the global level since no component can actually observe the relation directly.

We analyzed threat/fault/attack models and specified media that can drop, reorder, duplicate, insert, and corrupt messages in terms of input/output relations on message streams. We proved the composition of these 5 atomic threat models to be commutative. As a result, media in which all or some of these threats occur simultaneously can be expressed by a simple composition of the atomic models.

Attack	$Rel(s, r)$	Mitigation mechanism
Out of order packets	$s \leftrightarrow r$	sequence numbers
Packet drop	$s \sqsupseteq r$	ACK + timeout
Packet insertion (spoofing)	$r \sqsupseteq s$	authentication
Packet duplication	$r \sqsupseteq s \ \wedge \ sameoids(s, r)$	message identifiers
Packet corruption	$sameids(s, r) \wedge samelengths(s, r)$	checksum, parity

Figure 11: Attack Models

An unfinished goal of this project was to be able to prove that the corresponding generic mitigation protocols have the same property, which would mean that any network protocol could be specified as a simple composition (i.e. stack) of atomic micro-protocols.

4.3 Generating a family of PubSub Implementations

During Phase 2, we derived a family of Publish-Subscribe transport layer codes targeted to the CertiKOS layer on the Landshark and ABA. Figure 12 shows a roadmap of the derivations we performed. Each derivation starts from a common specification of

a communication requirement. We derived concrete C codes needed to support IPC, IVC, CertiKOS-process, and IP-based communications on the Landshark and ABA platforms. In the following sections, we give more detail of each derivation.

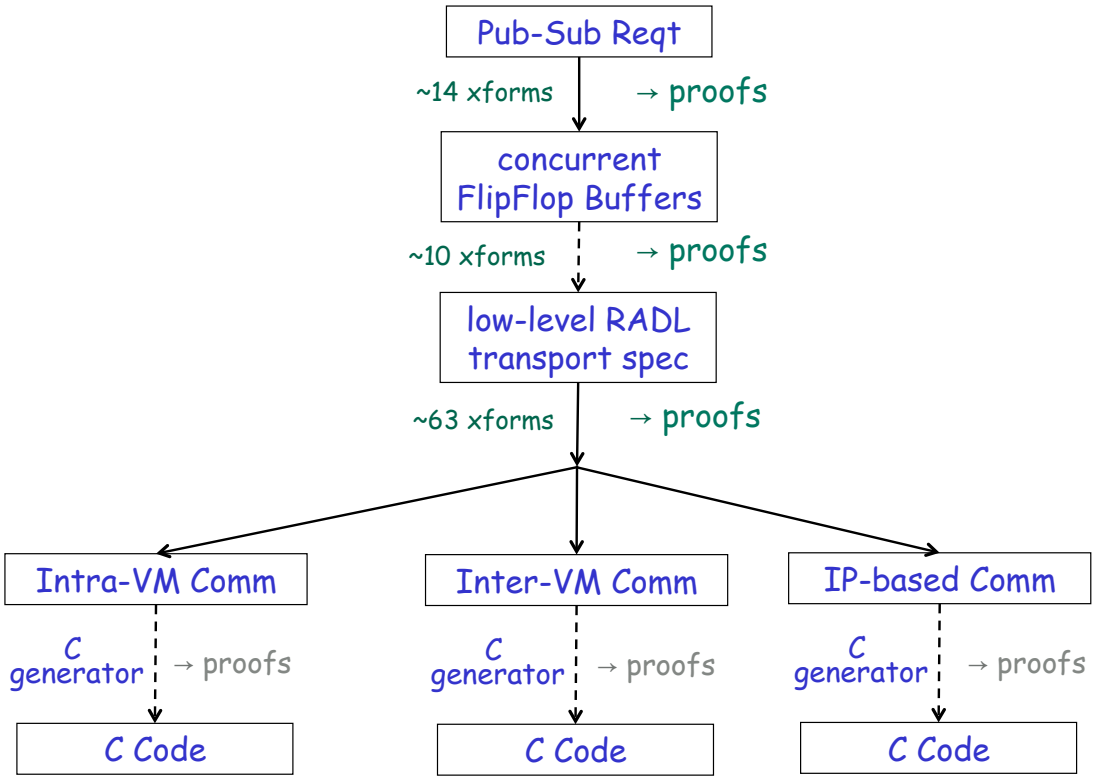

Figure 12: Derivational Family Tree of PubSub Protocol Codes

Our starting point is the most abstract logical specification of a communication requirement, as discussed in Section 3.5.1. In terms of a diagram, that specification has the structured form depicted in Figure 13:

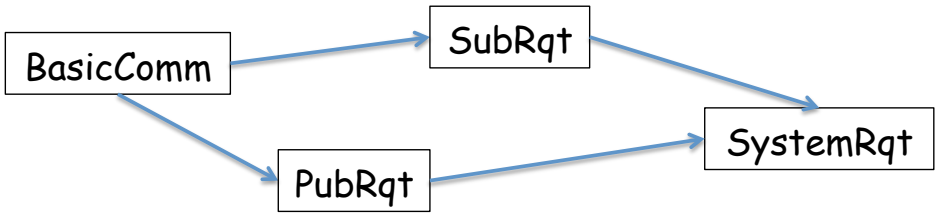

Figure 13: Specification Diagram of Communication Requirements

where the boxes denote specifications and the arrows denote morphisms (here the arrows are the special case of an import morphism).

4.3.1 Introducing a Communication Channel

We begin to satisfy the requirements of this structured specification by applying the Periodic Protocol Theory that was discussed in Section 3.6.1, which added a communication channel and an operation for periodically publishing the current value of the pub_topic. This refined structure is strong enough to prove the Global_Invariant axiom. At the same time, it introduces new local requirements, expressed as axioms, which will lead to further refinements. Diagrammatically, this is shown in Figure 14. Here, the downward arrows again denote morphisms, but in this case they are proper refinements, not just imports.

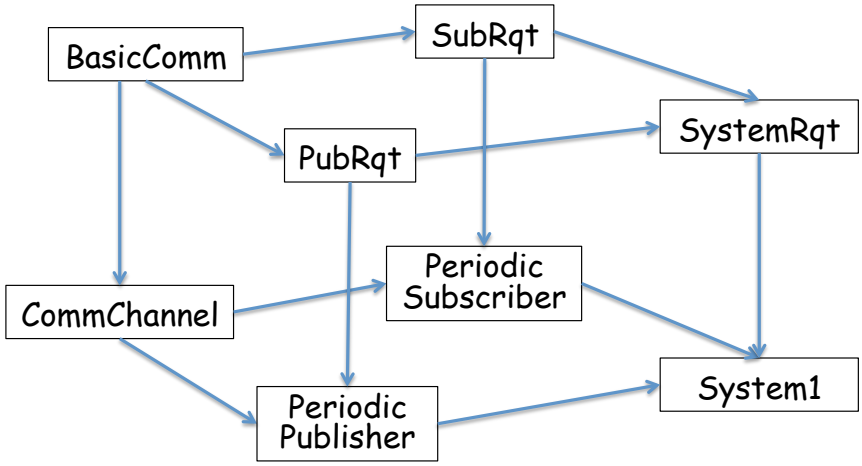

Figure 14: A Diagram Refinement - Adding Communication

The specification CommChannel introduces an underspecified type Channel together with operators send, recv, and read. To specify the semantics of a Channel, we make use of a ghost observer hist that records the sequence of events that occur on the Channel. This allows us to define other ghost observers: sent (the timed sequence of message written/sent on the channel) and received (the timed sequence of message received on the channel). These ghost observers allow us to formulate the main semantic constraint on a channel: the received messages are equal to the sent messages with a bounded latency. The behavior of the Channel is underspecified to allow for alternative refinements, such as a bounded queue, bounded FIFO queue (order-preserving), or, as in our case, a mailbox, which has different properties than a queue.

```
CommChannel = spec
  import Common#Time,
         Common#Tracking
```

```
  op inbox : State -> Channel -> Payload
  op outbox: State -> Channel -> Payload

  op medLat: Duration      % max comm latency, interpreted in later refinements

(*  We can specify the meaning of send as an update to
     (1) the outbox of the channel (shared with the sender process),
     (2) the sent ghost observer, which helps with design-time calculation,
     (3) hist.
*)

  op send(st:State)
       (ch:Channel)
       (pld:Payload):
       {st':State
       |   outbox st' ch = pld
       &&  inbox st' ch = inbox st ch
       && hist st' = hist st <| {state=st, time=clock st, action = Send (ch, pld)}
       && sent st' ch = (sent st ch)
                        <| {state=st, time=clock st, action = Send (ch, pld)}
       }

  op sent: State -> Channel -> List Event
  theorem M_sent_Invariant is         % list of sent topic msgs
    fa(st:State,chid:Channel)
      sent st chid = (filter (fn(ev:Event)-> ((embed? Send)ev.action))
                       (filterHist_byChannel st chid))

(* The recv operation, invoked by the channel (not the recipient) , places a msg
 in the recipient's inbox. *)

  op  recv(st:State)(ch:Channel)(pld:Payload):
        {st':State
        | inbox st' ch = pld      % mailbox
        && outbox st' ch = outbox st ch
        && hist st' = hist st <| {state=st, time=clock st, action = Recv (ch, pld)}
        && received st' ch = (received st ch)
                         <| {state=st, time=clock st, action = Recv (ch, pld)}
        }

  op received: State -> Channel -> List Event
  axiom M_received_Invariant is     % list of received topic msgs
    fa(st:State,chid:Channel)
      received st chid
      = (filter (fn(ev:Event)-> ((embed? Read)ev.action))
```

```
            (filterHist_byChannel st chid))

% The read operation, invoked by the recipient, returns the current msg in the inbox

  op read(st:State) (ch:Channel):
       {(st',pld):State*Payload
       | pld = inbox st ch
       && hist st' = hist st <| {state=st, time=clock st, action = Read (ch, pld)}
       && received st' = received st
       }

  axiom Channel_Reliability is
      fa(ch:Channel) fresh(received, ch, sent, ch, medLat)

  axiom Send_precedes_Recv is
   fa(st:State,chid:Channel,recv_ev:Event)
     (recv_ev in? (received st chid)
       => (ex(send_ev:Event)
            (send_ev in? (sent st chid)
              && act2payload recv_ev.action = act2payload send_ev.action
              && recv_ev.time <= send_ev.time + medLat)))

 end-spec
```

The establishment of the Global_Invariant theorem reduces to the Channel_Reliability axiom here, and is witnessed by an Isabelle-checked proof in our formal specification.

4.3.2 Refining the Communication Channel to a Mailbox

The next step is to introduce the notion of a mailbox-style communication channel that is suitable to the periodic reads/writes of publish-subscribe protocols. It provides atomic reads/writes to a shared mailbox. A mailbox communication channel provides a single data value at any one time – to formalize this, the Mailbox specification renames the Channel type as MB, and introduces new observers

op content: MB → Payload
op MB_md: MB → MBMetadata

as well as other observers.

A write will overwrite whatever is in the mailbox regardless of whether it has been read or not. Mailbox communication is suitable whenever it is required to have up-to-date information available (e.g. about sensed values) as opposed to guaranteed reception of all information. This additional semantic constraint on a Mailbox is specified as an axiom, which asserts that the content/payload of any read event is the content of that last write to the mailbox.

```
        axiom MB_semantics is
```

```
fa(st:State,chid:ChannelId, rd_ev:Event)
  (rd_ev in? (received st chid)
   =>
   act2payload(rd_ev.action)
   = act2payload(last(filter_Events_by_time
                          st (sent st chid) (rd_ev.time))))

op filter_Events_by_time (st:State)(evs:List Event)(t:Time): List(Event) =
    filter (fn(ev:Event)-> ev.time < t) evs
```

This is a common pattern in real-time control systems where the controller needs to have fresh data about the state of the plant and the environment whenever it needs to make control decisions.

To allow a control system to be aware of communication faults, flags are maintained that assert freshness-related properties. In our setting, such flags are refinements of the MB_md metadata observer above, and are formally calculated using the Observer Maintenance transformation (see Section 3.6.3).

Our approach was to formalize the concept of a mailbox and to perform a datatype refinement transformation to refine the underspecified type Channel in Specification CommChannel to a Mailbox with mailbox-specific operators, as diagrammed in Figure 15.

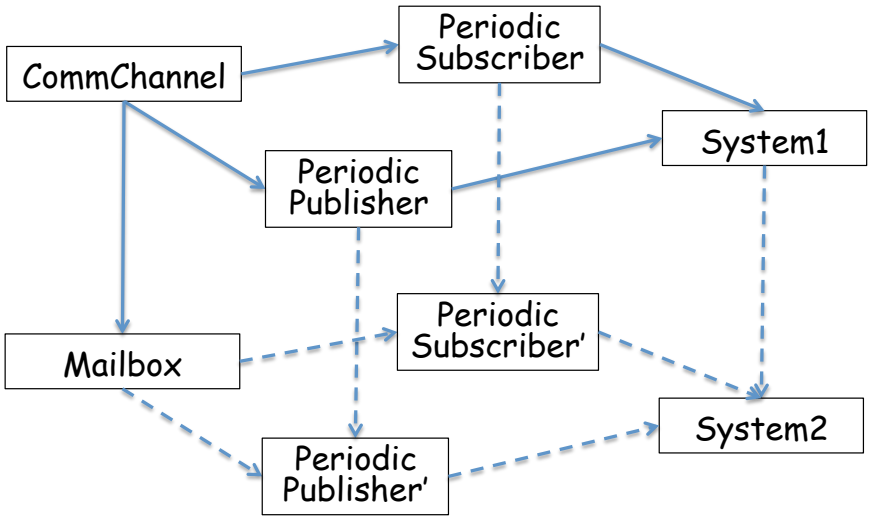

Figure 15: Datatype Refinement - CommChannel to Mailbox

This refinement is formalized as a morphism from CommChannel to Mailbox, and stored in the library. The refinement of the structured specification for System1 is computed as the (diagram) pushout of System1 with Mailbox (via CommChannel). The automatically generated (cocone) morphisms are depicted as dashed arrows. Formally, the pushout is expressed in the metaprogram/derivation as

```
        PubSub_via_Mailbox = System1[ CommMedium_to_MB ]
where
        CommMedium_to_MB = morphism CommChannel → Mailbox
                { Channel     ↦ MB
                , medLat      ↦ MB_delay
                , send        ↦ send
                , read        ↦ read
                , received    ↦ received
                , sent        ↦ sent
                }
```

4.3.3 Refining the Mailbox to a Concurrent FlipFlop Buffer

Another library theory formalizes the notion of a flip-flop buffer as a refinement of a mailbox. The flip-flop buffer has two parts, one for writing fresh topic data, and the other for concurrent reading of topic data. The polarity of write-versus-read buffer flips with each write. Ultimately, the flip-flop buffer is refined to kernel memory that is shared between processes, as managed by the hypervisor.

The FlipFlop specification mimics the Mailbox type MB as the FlipFlop type FF, and the MBmetadata type as FFMetadata. It adds new observers

```
        op ffdata_size  : FF → Nat16  % intended size in bytes of a Payload
        op ffbuf        : FF → Map(Nat1,Payload)  %  the flip-flop buffer
        op ffmetadata   : FF → FFMetadata
```

and new observers of metadata

```
        op ffsize       : FFMetadata → Nat32      % total FF size
        op readIndex : FFMetadata → Nat1          % is buffer 0 or 1 the current read buf?
        op writeIndex (ffmd:FFMetadata):Nat1 = 1 - (readIndex ffmd)
```

and others.

Again, this refinement is formalized as a morphism, from Mailbox to FlipFlop, and stored in the library with its proof. The refinement of the structured specification for System2 is computed as the (diagram) pushout of System2 with FlipFlop (via Mailbox). The automatically generated (cocone) morphisms are depicted as dashed arrows in Figure 16.

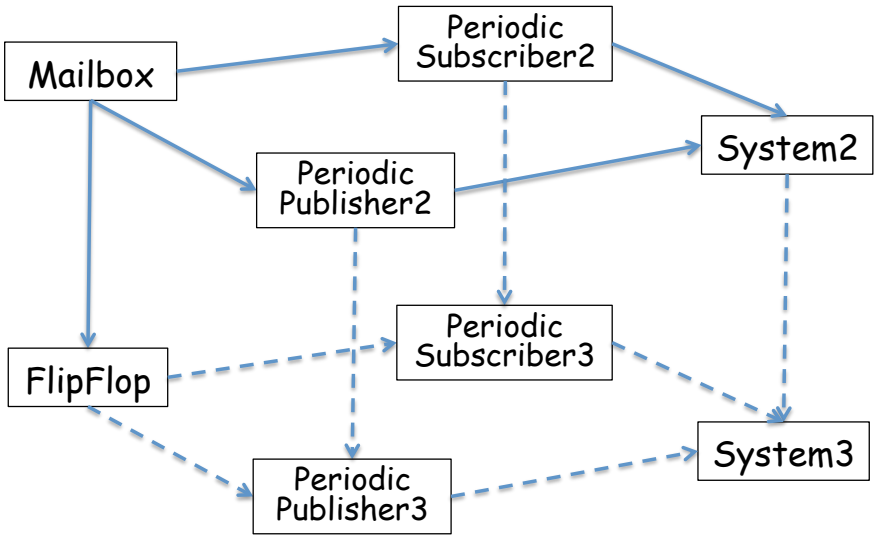

Figure 16: Refinement of Mailbox to FlipFlop

Formally, the refinement from System2 to System3 is computed by the metaprogram/derivation command

PubSub_via_FlipFlopBuffer = PubSub_via_Mailbox[Mailbox_to_FlipFlop]

where

Mailbox_to_FlipFlop = morphism MB_Channel → FlipFlop

{ MB ⟼ FF
, MBMetadata ⟼ FFMetadata
, MBbuf ⟼ FFBuf
, name ⟼ name
, contents ⟼ contents
, MB_md ⟼ ffmetadata
, get_MBname ⟼ get_FFname
, get_contents ⟼ read_Payload
, ReadResult ⟼ ReadResult
, MB_delay ⟼ FF_delay
, send ⟼ sendFF
, read ⟼ readFF
, received ⟼ received
, sent ⟼ sent
}

4.3.4 Refinement toward RADL code

So far the derivation has been generic, but now we begin to target the specific interface required by SRI's RADL architecture model [RADL15]. In particular, we formalized the articulated steps of the RADL node process:

update – open mailbox to read, show if stale
stall – open mailbox to write
release – close and release mailbox after read
commit – commit and close mailbox after write

Each is specified using cases, as with the UDP specification, and is later synthesized using the mergeRules transformation. We specified the RADL model and developed a straightforward refinement of FlipFlop to the RADL specification. Formally modeling RADL and precisely aligning our derivations with the RADL concepts took some effort.

The diagram in Figure 17 shows the refinement,

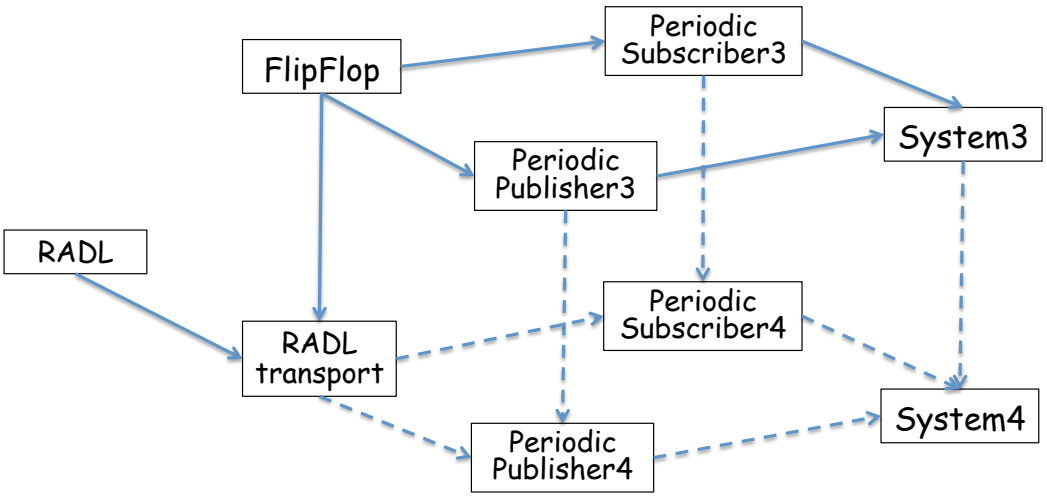

Figure 17: Refinement of FlipFlop-based System to RADL

As in previous steps, the refinement is calculated using a pushout.

4.3.5 MergeRules, finalizeCotype, and other Optimizations

At this point some seven mergeRule transformations are applied along with further optimization rules to clean up and improve the code. Then four applications of the finalizeCotype are applied, which generates record definitions for cotypes by converting observers into fields. See Section 3.6.5.

4.3.6 Deriving IPC, IVC, raw CertiKOS, and IP_based variants

A goal of this project was to provide transport-level communication support based on IPC support provided by the underlying CertiKOS hypervisor [CertiKOS] or the commercial LynxSecure hypervisor. The CertiKOS platform supported nodes running on several cores, each running multiple VMs. Several versions of IPC were provided:

1. InterProcess Communication (IPC) – to provide publish-subscribe service between nodes/processes running on the same Linux VM. The mailbox mechanism was implemented by shared memory in the Linux kernel.
2. InterVM Communication (IVC) – to provide publish-subscribe service between nodes/processes running on different VMs. The mailbox mechanism was implemented by shared memory in the hypervisor.
3. raw CertiKOS processes – to provide publish-subscribe service between nodes/processes where one node is running as a native CertiKOS process. The mailbox mechanism was implemented by shared memory in the hypervisor.

Lastly, we also generated support for mailbox communication over the radio link between an external controller and the vehicle (Landshark or ABA) using IP as the network layer.

The RADL model of mailbox communication decomposed sending and receiving into several operations:

- **kestrel_*_update** - open mailbox to read, show if stale
- **kestrel_*_stall** - open mailbox to write
- **kestrel_*_release** - close and release mailbox after read
- **kestrel_*_commit** - commit and close mailbox after write where * can be linux_ip, linux_ipc, lynxsecureVM_ivc, or certikosVM_ivc

Our approach was to specify 4 auxiliary ops

```
op mbox_buffer_update : State * MBuf * StaleFlag * DataSize -> State * InBox
op mbox_buffer_stall  : State * MBuf * DataSize -> State * OutBox
op mbox_buffer_release: State * MBuf -> State
op mbox_buffer_commit : State * MBuf -> State
```

then to define ops for IPC based on these ops; for example,

```
def kestrel_linux_ipc_update (st:State,ibox:Linux_IPC_MBoxInfo,stale:StaleFlag)
                                    : State * InBox
= let buf = linux_ipc_buffer (st,ibox) in
  let size = data_size (linux_ipc_gen_chan (linux_ipc_chan (st,ibox))) in
      mbox_buffer_update (st, buf, stale, size)
```

Similar specs were developed for for kestrel_certikosVM_ivc_update, kestrel_lynxsecureVM_ivc_update, and so on. The spec for kestrel_linux_ip_update was slightly more complex since messages must be in their respective mailboxes before the step function can be active. Also, we needed to shift the address by the size of IP header, since the mailbox will contain the IP header as well at the beginning of the data, but the step function should not see it.

To illustrate our approach in more detail, we focus on the specification of mbox_buffer_update, which has to (1) determine the location of the mailbox slot to read from, (2) check if the mailbox content had been marked read before, and (3) mark the current read slot as read:

```
op mbox_buffer_update
        (st:State, buf:MBuf, stale:StaleFlag, size:DataSize | true)
        : { (st', i_addr): State * InBox
          |  mbox_buffer_update_0 (st,buf,stale,size, st',i_addr)
          || mbox_buffer_update_1 (st,buf,stale,size, st',i_addr)
          }
```

There are two cases to consider, depending on which slot is currently the read slot

```
op mbox_buffer_update_0 (……)
 = ex (st0:State, stval: StaleVal)
        mbuf_read_slot (st,buf) = slot0
   && stval  = mbuf_slot0_staleval  (st,buf)
   && st0    = set_stale (st,stale,stval)
   && st'    = mbuf_set_slot0_staleval (st0,buf,is_stale)
   && i_addr = toInBox (mbuf_slot0_addr (st0,buf))

op mbox_buffer_update_1 (……)
 = ex (st0:State, stval: StaleVal)
        ~(mbuf_read_slot (st,buf) = slot0)
   && stval  = mbuf_slot1_staleval  (st,buf)
   && st0    = set_stale (st,stale,stval)
   && st'    = mbuf_set_slot1_staleval (st0,buf,is_stale)
   && i_addr = toInBox (mbuf_slot1_addr (st0,buf,size))
```

We transform this specification towards code by applying the MergeRules and StructureEx transformations. That is, applying the following metacode

```
Transport_rw_mbox = Transport_prot
      { at mbox_buffer_update   { unfold mbox_buffer_update_pred  }
      ; mergeRules (mbox_buffer_update, mbox_buffer_update) ()
      ; at [mbox_buffer_update] { repeat {structureEx} ; SimpStandard   }
      }
```

results in the following refined spec

```
refine def mbox_buffer_update
  (st: State, buf: MBuf, stale: StaleFlag, size: DataSize | ~ false)
  : {(st', i_addr): (State * InBox) |
      if mbuf_read_slot(st, buf) = slot0
       then let st0 = set_stale(st, stale, mbuf_slot0_staleval(st, buf)) in
            st' = mbuf_set_slot0_staleval(st0, buf, is_stale)
            && i_addr = toInBox(mbuf_slot0_addr(st0, buf))
       else  let st0 = set_stale(st, stale, mbuf_slot1_staleval(st, buf)) in
            st' = mbuf_set_slot1_staleval(st0, buf, is_stale)
            && i_addr = toInBox(mbuf_slot1_addr(st0, buf, size))}
```

Then, we refine Channel, Data, Mbuf, StaleFlag, DataSize, InBox, and corresponding observers, followed by inlining the refined operations using unfold:

```
refine def mbox_buffer_update
   (st: State, buf: MBuf, stale: StaleFlag, size: DataSize | ~ false)
   : {(st', i_addr): (State * InBox) |
       if ((deref st buf).hdr).read_slot = slot0
         then  let st0 = assign st stale (((deref st buf).hdr).stale0) in
               st' = set st0 (((deref st0 buf).hdr).stale0) is_stale
               && i_addr = toInBox((deref st0 buf).data)
        else  let st0 = assign st stale (((deref st buf).hdr).stale1) in
              st' = set st0 (((deref st0 buf).hdr).stale1) is_stale
              && i_addr = toInBox(addAddr((deref st0 buf).data, size))}
```

The next step is to create an imperative algorithm with stateful data structures. We refine and inline primitive operations assign, is_stale, slot0, toInBox, then apply the metacode

```
Cot = transform UnfoldPrimitives  by {finalizeCoType(State)}
```

resulting in

```
refine def mbox_buffer_update
  (st: State, buf: MBuf, stale: StaleFlag, size: DataSize | ~ false)
  : {(st', i_addr): (State * InBox) |
     if ((deref st buf).hdr).read_slot = 0
      then let st0 = set st (deref st stale) (((deref st buf).hdr).stale0) in
           st' = set st0 (((deref st0 buf).hdr).stale0) 1
           && i_addr = (deref st0 buf).data
      else let st0 = set st (deref st stale) (((deref st buf).hdr).stale1) in
           st' = set st0 (((deref st0 buf).hdr).stale1) 1
           && i_addr = addAddr((deref st0 buf).data, size)}
= if ((deref st buf).hdr).read_slot = 0
   then let st0 = set st (deref st stale) (((deref st buf).hdr).stale0) in
```

```
        ( set st0 (((deref st0 buf).hdr).stale0) 1
        , (deref st0 buf).data)
    else let st0 = set st (deref st stale) (((deref st buf).hdr).stale1) in
        ( set st0 (((deref st0 buf).hdr).stale1) 1
        , addAddr((deref st0 buf).data, size))
```

Finally, we apply code generation phases, which, after preprocessing, links abstract operations to existing C primitives. The metacode

```
Derivation_transport_mbox = PrepForC_mbox
{... ; genC (  ( mbox_buffer_update, .....), ....
          ,  "../NP/Derivations/LT1/Synthesized/kestrel_mbox"  )
}
```

creates the code and stores it in .c/.h files

```
typedef const char *   constcharstar;
typedef int *          intstar      ;
struct mbox_buffer_hdr { unsigned int unused:13, stale0:1, stale1:1, read_slot:1;
};
struct mbox_buffer     { struct mbox_buffer_hdr hdr; char data[]; };
constcharstar mbox_buffer_update(MBuf buf, intstar stale, Nat_Nat32 size)
{ if (buf->hdr.read_slot == 0) {
      *stale = buf->hdr.stale0;
      buf->hdr.stale0 = 1;
      return buf->data;
     }
  *stale = buf->hdr.stale1;
  buf->hdr.stale1 = 1;
  return buf->data + size;
}
```

Altogether, we specified the mailbox communication needed on the Landshark and ABA and developed derivations that generate C. The generated C handles all send/receive of topics, but not the mailbox initialization (which has been written by hand). Working closely with the SRI and Yale teams, we delivered transport code in C to support (1) intra-VM communication, (2) inter-VM communication, (3) communication with CertiKOS processes, and (4) IP-based communication. The results were demonstrated at the end of Phase II. The Red Team found no vulnerabilities in the cyber-physical systems that contained our delivered code.

4.4 Proof generation results

A key focus of the project was generating proofs automatically from the various transformation steps in our PubSub derivation. We extended the following transformations to emit proofs at application time: Simplification, Observer Maintenance, Observer Implementation, StructureEx, finalizeCotype, and others. These constitute most of the transformations used in our derivation.

In our Phase 1 derivations for UDP codes, we had the measures shown in Figure 18.

	Transformations	Theorems	lines of ISAR/ Isabelle proof script
UDP receive code	12	48	10,948
UDP send code	12	512	109,676

Figure 18: Proofs generated as a by-product of refinement

In our Phase 2 work, our objective was to mechanize the generation of correct-by-construction C code for the entire family tree of communications codes, together with automatically generated proofs of their correctness. For the Protocol Design Theories (described in Sections 3.5.1 and 3.6.1), we used Isabelle to verify by hand once and for all

- Common Concepts (time, history, tracking,…) 7 proofs
- SharedMemory (Channels, FlipFlop) 9 proofs
- PubSubProtocol (periodic update) 1 major proof

Then, for the Protocol Design phase, where we applied the protocol theories and refined the communication channel, the proofs were generated by transforming library proofs

- PubSub_via_Mailbox 5 proofs
- PubSub_via_FlipFlopBuffer 4 proofs
- PubSub_via_RADL_FlipFlop 4 proofs

Then, the transformation for deriving the RADL protocol implementation in Specware automatically emitted proofs

- Merging cases + simplification 23 proofs
- Finalizing Types and Records 112 proofs

Finally, at the end of Phase II we were working on a new C code generator that would emit proofs. Details of this work are presented in Appendix 1.

The automatic generation of checkable proofs as a by-product of transformations is a major result of the project. We have demonstrated that a derivational approach to system code generation can produce proofs as a by-product and that *the marginal cost of producing those proofs is effectively zero*.

4.5 Rehosting Specware in Coq

We began working on rehosting the "Specware approach" to program synthesis in the Coq proof assistant. The goal was not only to ease interoperability with other teams who are performing verification using Coq, but also, as a broader impact, to help disseminate the ideas about program synthesis developed at Kestrel. Technical details of this effort are presented in Appendix 2.

4.6 Papers

We finished two papers this period that are intended for publication:

Title: Incremental Definition of Types

Authors: Douglas R. Smith, Edwin Westbrook, and Stephen J. Westfold,

Abstract: There are many situations in which it would be useful to incrementally develop the definition of a type and its operators. Examples include family trees of programs, software product lines, and object-oriented class hierarchies. In a formal specification and refinement setting, we present a refinement idiom and supporting transformations for incrementally defining both inductive and coinductive types. Inductive types are built by incrementally accumulating constructors. Coinductive types are built by incrementally accumulating observers. In each case, when the refinement process is finished, we declare the constructor (resp. observer) set is complete by applying a transformation that gives a canonical definition to the type. It also generates definitions for functions that have been characterized inductively(resp. coinductively) by their action on the constructors (resp. observers).

Title: Consistency of Refinement

Author: Douglas R. Smith

Abstract: One of the deepest issues with a specification-and-refinement approach is knowing whether the intermediate specifications are consistent. Any specification can refine to an inconsistent specification, but there is no refinement from an inconsistent specification to a program in a programming language. Once a refinement process has reached an inconsistent specification, the process is essentially stuck, and requires some form of backtracking. Hence, there is value in generating refinements that are consistent by construction. This paper develops basic metatheory about consistent refinements and their construction, and proves the most general conditions known to us under which a pushout construction generates a consistent refinement.

Several other papers are currently in progress:

1. Kimmell, Kreitz, Smith, Westbrook, and Westfold, Proof-Emitting Transformations.
2. Kimmell, Kreitz, Smith, Westbrook, and Westfold, Synthesis by Cases.
3. Pepper, Smith and Westfold, Coalgebraic Specification and Refinement

5 CONCLUSIONS

To conclude, we discuss the results of this project.

Deliverables

In Phase 1 we delivered working and verified C code for handling UDP traffic that was plug-compatible with the Linux kernel. The code did not have several vulnerabilities that occurred in earlier versions of UDP. The Red Team did not find flaws in the code.

The problem arose of how to specify UDP. The definition of UDP is partly by public RFC's and partly by established practice in the form of concrete implementations in operating systems (Unix, Linux, Windows). To be plug-compatible with the Linux kernel, which was needed in the Landshark, required that we abstract out the specification of the context of the UDP code in the Linux kernel. This was a difficult exercise and resulted in a disappointingly low-level specification. Nevertheless, we developed that low-level specification and developed the tools and formal derivation to generate C code that we could install as an alternate module to the existing UDP

processing code in the Linux Kernel. Testing revealed no semantic differences. In retrospect, picking a more clean-slate target for our synthesis technology might have resulted in a more useful product both technically and practically.

In Phase 2, our strategy was to work closely with our partners (specifically SRI and Yale), continuously supplying them with working code, even if it had to be written manually, so we could work together toward the Phase 2 integrated demonstrations. Our effort was split between (1) understanding our partner's needs and supplying them with code, and (2) developing abstract requirement specifications and working toward generating the code and proofs needed for the Phase 2 demo. By the end of phase we had generated a family of C codes for the publish-subscribe mailbox transport service. These codes were installed, tested, and demonstrated on the Landshark and ABA in August 2015. The Red Team found no vulnerabilities in our code when they performed their system security tests on the vehicles.

Security Goals

Towards the goal of enhanced security of vehicles, our approach was to generate safe and secure codes that service the communication protocols on vehicles, as opposed to generating new protocols that are secure. This approach was dictated by the need to use existing protocols. Working with protocols established elsewhere did not allow us to pursue a fully clean-slate approach, and challenged us to generate code into existing code frameworks. Absence of many of the vulnerabilities that are rife in conventionally produced protocol software, such as buffer overflows and null pointer dereferences, is an intrinsic and checkable feature of our approach.

Formal Methods Goals

Kestrel's long-term goals are to develop automated systems for refining formal requirement-level specifications into correct-by-construction code, with certificates. This project focused our efforts on co-generating code and proofs, and we successfully demonstrated how to extend our transformations to emit both refined specifications and machine-checkable proofs of the correctness each refinement step down to the code-generation level. By the end of Phase 2 we had designed and partially implemented a C generator that would emit proofs. By automating the production of proofs, we lowered the cost of providing high levels of assurance as a normal part of software development. We demonstrated that a derivational approach to algorithm generation can produce proofs as a by-product and that *the marginal cost of producing those proofs is effectively zero*.

We also developed a variety of new transformations and new protocol theories to mediate the transformation of requirements to abstract designs. We expect that these transformations and library theories will be applicable in other domains. The Specware system was further developed under this project and made open-source via Github.

A crucial aspect of software engineering that is rarely addressed in formal approaches is the cost of code maintenance and evolution. In our formal approach, there should be no maintenance in the sense of bug fixing, but there will always be a need to adapt to changing requirements and changing design decisions. The family tree of codes that we generated in Phase 2 via a family tree of derivation metaprograms provides insight toward the goal of formal tools to support evolution via changing design decisions. In this case, the family tree of derivations has (1) a common top: a common specification and initial design decisions, and (2) diverging design decisions at lower levels that target to different target platforms (e.g. intra-VM versus inter-VM). Thus we demonstrated how to evolve an implementation to a new platform by branching in the middle of a derivation and applying a variant sequence of transformations.

A larger lesson of this project was confirmation that we could express the design of communication code in terms of generally applicable design abstractions. The abstractions are incrementally translated to executable C via formal refinement steps that emit checkable proofs. The sequence of refinements and the design abstractions they embody constitute a (1) human-understandable explanation of the complex executable code, and (2) an effective means for constructing complex code for a specific set of requirements out of reusable design abstractions.

6 References

[CertiKOS] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo, CertiKOS: A Certified Kernel for Secure Cloud Computing, In Proc. 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'11), Shanghai, China, July 2011.

[Datta05] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic, "A derivation system for security protocols and its logical formalization", in Proceedings of CSFW 2003", IEEE Press, 2003, pp 109—125.

[Isabelle] Isabelle Interactive Theorem Prover, https://isabelle.in.tum.de

[Jacobs97] Jacobs, B., and Rutten, J., A tutorial on (co)algebras and (co)induction. Bulletin of the European Association for Theoretical Computer Science 62 (1997), 222—259.

[Liu13] Liu, Y., Systematic Program Design: From Clarity to Efficiency, Cambridge University Press, 2013.

[Mossakowski06] Mossakowski, T., Reichel, H., Roggenbach, M., and Schroeder, L. Algebraic-coalgebraic specification in CoCasl, J. Logic Algebraic Programming 67 (2006).

[Paige82] Paige, R. and Koenig, S., Finite Differencing of Computable Expressions, TOPLAS 4(3), 1982, 402-454.

[Rothe01] Rothe, J., Tews, H., and Jacobs, B. The coalgebraic class specification language CCSL, Journal of Universal Computer Science 7, 2 (2001), 175—193.

[Rutten00] Rutten, J. Universal coalgebra: a theory of systems, Theoretical Computer Science 249, 1 (2000), 3 — 80.

[SmithD9009] Smith, D.R. KIDS — a semi-automatic program development system, IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16, 9 (1990), 1024—1043.

[Specware03] Kestrel Institute, Specware System and documentation, 2003. http://www.specware.org/. Open source download from https://github.com/KestrelInstitute/Specware

[RADL15] Li, W., Gerard, L., and Shankar, N., Design and Verification of Multi-Rate Distributed Systems, ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2015, 20-29.

[SmithD9302] Smith, D.R. Pushouts preserve conservative extensions: Another proof of the modularization theorem, Tech. Rep. KES.U.93.1, Kestrel Institute, February 1993.

[Spitters] Spitters, B. and van der Weegen, E., Type Classes for Mathematics in Type Theory, Math. Structures in Computer Science, Vol 21, 2011, pp 795-825.

[Whalen02] Whalen, M., Schumann, J., and Fischer, B. Synthesizing certified code, in Proc. Formal Methods Europe (FME 2002) (Copenhagen, Denmark, 2002), Springer LNCS 2391, pp.431—450.

9 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ABA	American Built Automobile
DARPA	Defense Advanced Research Projects Agency
HACMS	High-Assurance Military Systems
RADL	Robot Architecture Definition Language
RFC	Request For Comments
ROS	Robot Operating System
TCP	Transmission Control Protocol
UDP	User Datagram Protocol