Verification of Secure Distributed Systems in Higher Order Logic: A Modular Approach Using Generic Components *

Jim Alves-Foss and Karl Levitt
Division of Computer Science
University of California at Davis
Davis, CA 95616

Abstract

In this paper we present a generalization of McCullough's restrictiveness model as the basis for proving security properties about distributed system designs. We mechanize this generalization for an event-based model of computer systems in the HOL (Higher Order Logic) system to prove the composability of the model and several other properties about the model. We then develop a set of generalized classes of system components and show for which families of user views they satisfy the model. Using these classes we develop a collection of general system components that are specializations of one of these classes and show that he specializations also satisfy the security property. We then conclude with a sample distributed secure system, based on the Rushby and Randell distributed system design and designed using our collection of components, and show how our mechanized verification system can be used to verify such designs.

1 Introduction

This paper is concerned with mechanizing the verification of distributed system design. Following the work of McCullough and his colleagues at Odyssey Research Associates [8, 9, 10, 13, 14], we characterize the security of components of a design using the *restrictiveness* model: A user of the component is prevented from observing any outputs dependent on inputs outside his view. McCullough also provides a definition of *hook-up*, where two components, each satisfying the restrictiveness model, are connected in a way that guarantees their composition also satisfies restrictiveness. This way a system can be created that is guaranteed to satisfy the security model as long as each component and their connections satisfy the model.

We extend the McCullough model by writing specifications for a class of generic building blocks, the class including filters of various kinds. Next we define a large class of components that are specializations of these filters and show that the specializations satisfy restrictiveness. These components include queues, transformers, multiplexors, de-multiplexors and switches. To establish the generality of the class of components, we show they can be connected to produce the Rushby-Randell distributed system design.

All proofs are mechanically checked using the Higher Order Logic (HOL) system developed at Cambridge University [6]. HOL was selected for this project based on its support for higher-order logic, generic specifications, and polymorphic type constructs – all in support of writing and reasoning about general classes of components.

Section 2 of this paper gives an overview of the HOL theorem proving system. Section 3 describes how to use the HOL system to develop the theories for the sequence and event system abstract data types. Section 4 presents a generalized definition of restrictiveness and theorems that we have proven about it. It also presents a definition of composability and shows how it relates to restrictiveness. Section 5 presents a description of generic system component classes and the verification of the security property applied to them. It discusses how the classes are developed in the HOL system, and shows how the generic components can be specialized to system components. Finally section 6 presents an example distributed secure system (the Rushby - Randell system) and how the generic components can facilitate its specification and verification. General conclusions of this work and plans for future work are also presented.

2 The HOL System

To formally model the security properties of a secure distributed system and to ensure the accuracy of our proofs, we felt that it was necessary to develop the proofs and properties using a mechanical verification system. This prevents proofs from containing logical mistakes, and assures that the foundations on which the work is based are sound. Due to the nature of the proofs, which include quantification over sets of objects, we felt that a system which supports higher-order logic and a typed lambda calculus would facilitate our efforts. The HOL system was selected for this project due to its support for higher-order logic, generic specifications and polymorphic type constructs. Furthermore its availability, ruggedness, local support, and a growing world-wide

*This work was supported in part by contracts from Rome Air Development Center and LLNL.

user base made it a very attractive selection. In this section we will provide a brief description of HOL.

HOL is a general theorem proving system developed at the University of Cambridge [6, 3] that is based on Church's theory of simple types, or higher-order logic [5]. Although Church developed higher-order logic as a foundation for mathematics, it can be used for reasoning about computational systems of all kinds. Similar to predicate logic in allowing quantification over variables, higher-order logic also allows quantification over predicates and functions thus permitting more general systems to be described.

HOL is not a fully automated theorem prover but is more than simply a proof checker, falling somewhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

1. Several built–in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories build on the five axioms that form the basis of higher-order logic to derive a large number of theorems that follow from them.

2. Rules of inference for higher-order logic. These rules contain not only the eight basic rules of inference from higher–order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.

3. A large collection of tactics. Examples of tactics include `REWRITE_TAC` which rewrites a goal according to some previously proven theorem or definition, `GEN_TAC` which removes unnecessary universally quantified variables from the front of terms, and `EQ_TAC` which says that to show two things are equivalent, we should show that they imply each other.

4. A proof management system that keeps track of the state of an interactive proof session.

5. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form new theories for later use. The metalanguage makes the verification system extremely flexible.

In the HOL system there are several predefined constants which can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written "`rand1 op rand2`" instead of in the usual prefix form: "`op rand1 rand2`". Table 1 shows several of HOL's built–in infix operators.

Constants can also belong another special class called binders. A familiar example of a binder is $\forall$. If `c` is a binder, then the term "`c x.t`" (where `x` is a variable) is written as shorthand for the term "`c(λ x. t)`". Table 2 shows several of HOL's built–in binders.

Operator	*Application*	*Meaning*
=	`t1 = t2`	`t1` equals `t2`
,	`t1,t2`	the pair `t1` and `t2`
$\wedge$	`t1` $\wedge$ `t2`	`t1` and `t2`
$\vee$	`t1` $\vee$ `t2`	`t1` or `t2`
$\Longrightarrow$	`t1` $\Longrightarrow$ `t2`	`t1` implies `t2`

Table 1: HOL Infix Operators

Binder	*Application*	*Meaning*
$\forall$	$\forall$ `x. t`	for all `x`, `t`
$\exists$	$\exists$ `x. t`	there exists an `x` such that `t`
ε	ε `x. t`	choose an `x` such that `t` is true

Table 2: HOL Binders

In addition to the infix constants and binders, HOL has a conditional statement that is written a $\rightarrow$ b | c, meaning "if a, then b, else c."

3 Formal System Model

The work presented in this paper involves the specification and verification of secure distributed systems. This requires the development of the system specifications in a formal framework, the definition of certain security properties for the system, and the validation that these properties are held true for the system.

We define a secure distributed system using an event-based model.[1] This model is derived from the one presented by McCullough in [10, 8], which is based on the processes (CCS, CSP) of Milner [11] and Hoare [7].

This model defines systems in terms of sequences of events where each event is either a communication event (input or output), or an internal transition event. McCullough's model of event-systems requires a 4 - tuple representation consisting of the set of events, set of inputs, set of outputs, and set of valid traces for the system. A valid trace of the system is a sequence of events that is a possible event history for the system. Any two event systems will be considered identical if there exists no difference in their behavior; that is, if the set of traces for the systems are identical.

To use the HOL theorem proving system we had to formally specify the theory of event systems and the theory of restrictiveness. This work is similar to other work on mechanizing event based systems and restrictiveness [13, 14, 4]. Once these theories were complete we could then use them to develop the basic system.

The event system model consists of a collection of four sets. These four sets are the component fields of the event system that represent the events, inputs, outputs and traces of the event system. The theory of event systems in the HOL system combines these sets into a 4 - tuple representation and enforces relationships between the fields. The relationships ensure that the inputs and outputs of the event system are disjoint subsets of the set of events and that the set of traces of the

[1] Also known as the *trace* model.

event system are sequences of events. We also developed many operations and theorems for the theory of event systems. We have included a partial description of the HOL event-system theory in Appendix B.

A major portion of the event system model is the set of system traces. To model these traces in the HOL system we first had to develop a theory of sequences (very similar to lists). The abstract data type specification we developed for sequences is a polymorphic abstraction. A sequence thus consists of an ordered collection of objects of some single type. Any particular sequence is either the empty sequence or is some sequence with an entry appended to the end of the sequence. HOL allowed us to define many operations on sequences to ease their manipulation. Given these operations one can develop a library of useful theorems relating to the operations and then use them in subsequent proofs. We have included a partial description of the HOL sequence theory in Appendix A.

These sequences and event system theories are used as the basis for the rest of our work. We develop a security property and generic component specifications for event systems. The security property and generic specifications rely on the existence of the data types described above. Complete listings of the libraries for these data types and the security property are given in [1, 2].

4 Formal Security Property

This section describes a formal security property that is a generalization of McCullough's Restrictiveness property. This property is true if the event system does not enable a user to deduce any information from events that the user is not authorized to see. For a classical military system to be restrictive, it must prohibit, for example, a Secret level user from being able to deduce any information from Top-Secret level events in the system.

In general, proving that a system satisfies restrictiveness is difficult. To simplify this proof, we have shown that restrictiveness satisfies McCullough's hook-up property. Thus, if components of a system are restrictive, we can use that information in the proof that the system derived from the proper hook-up of the components is restrictive.

4.1 Restrictiveness

The security property we present here is based on the restrictiveness property defined by McCullough in [8, 9, 10]. Below we present the HOL version of McCullough's restrictiveness for event systems and an auxiliary predicate, SAME-VIEW. This auxiliary predicate is true if two traces (sequences of events) are the same when restricted to the events in a particular view (i.e. When ignoring events outside the view, the sequences are the same).

```
SAME_VIEW
⊢def ∀v t1 t2.
   SAME_VIEW v t1 t2 =
    (t1 SEQ_RESTRICT v = t2 SEQ_RESTRICT v)

IS_RESTRICTIVE_DEF
⊢def ∀v es.
   IS_RESTRICTIVE v es =
    (∀a b1 b2 g1.
      let EV = EVENTS es and
          TR = TRACES es and
          INP = INPUTS es and
          I_D_V = (INPUTS es) DIFF v in
       (a IN TR ∧
        SEQ_IN_SET_STAR b1 INP ∧
        SEQ_IN_SET_STAR b2 INP ∧
        SEQ_IN_SET_STAR g1 EV ∧
        ((a SEQ_CONC b1) SEQ_CONC g1) IN TR ∧
        (SAME_VIEW v b1 b2) ∧
        (g1 SEQ_RESTRICT I_D_V = NULL_SEQ) ⟹
         (∃g2.
          (SEQ_IN_SET_STAR g2 EV ∧
           ((a SEQ_CONC b2) SEQ_CONC g2) IN TR) ∧
          (SAME_VIEW v g2 g1) ∧
          (g2 SEQ_RESTRICT I_D_V) = NULL_SEQ)))
```

Consider a set of events representing a user's view of the system *v*, an event system *es*, and a valid initial trace of the event system *a*. Let *b1* and *b2* be sequences of inputs of the event system such that they appear the same with respect to the view *v*. We know, due to the fact that this definition assumes the event system is input total, that *a* concatenated with either *b1* or *b2* is a valid trace of the system. Since these traces will appear the same with respect to the view *v*, the system will be secure only if the sets of possible future events, for each trace, appear the same with respect to the view *v*. Thus, if we let *g1* be a sequence of future events for the trace (a SEQ_CONC b1), the event system will be restrictive, with respect to *v*, if there exists a sequence of future events *g2* for the trace (a SEQ_CONC b2) that appears the same as *g1* with respect to the view *v*.

We began our work trying to duplicate the proofs presented in [8]. Two of these proofs involve showing that restrictiveness is a composable property, and that a delay-queue specification is restrictive. Working with this definition requires induction over the sequence representing the differences between the input sequence *b1* and *b2*. Although we have been able to use this induction scheme in HOL we came to the conclusion that this particular definition was very difficult to work with. The requirement that we look at future sequences of events and sequences of inputs presented a great deal of overhead.

In [8] McCullough stated this difficulty and proceeded to present definition and proofs based on a state-machine model. In the state-machine definition he changed his perspective and only considered single state transitions instead of a sequence of events as in the event system definition. This change in perspective greatly reduces the complexity of the proofs and the overhead required for sequences of events.

As we stated earlier, we wanted to continue our work using the event-based model. So, we decided to change our perspective and rework the definition of restrictiveness for event systems based on single events instead of sequences of events.

Our definition given below does not insist that the event system be input-total, only that for any two traces *t1* and *t2*, that appear the same in a user's view, if an input event following *t1* is a legal trace then that same event following *t2* must be a legal trace. This expanded definition permits us to explore component specifications which cannot be input-total. Rosenthal in [14] presents other approaches to expanding the definition of restrictiveness to deal with non input-total systems.

```
RESTRICT
⊢def ∀v es.
   RESTRICT v es =
    (∀t1 t2 e.
      let EV = EVENTS es and
          TR = TRACES es and
          INP = INPUTS es and
          seq_e = ENTRY e NULL_SEQ and
          I_U_V = (INPUTS es) UNION v in
       (t1 IN TR ∧
        t2 IN TR ∧
        SEQ_IN_SET_STAR t1 EV ∧
        SEQ_IN_SET_STAR t2 EV ∧
        e IN EV ∧
        (ENTRY e t1) IN TR ∧
        SAME_VIEW v t1 t2 ⟹
         (e IN v →
          (e IN INP →
           (ENTRY e t2) IN TR |
           (∃g1 g2.
            (t2 SEQ_CONC
               (g1 SEQ_CONC (seq_e SEQ_CONC g2)))
             IN TR ∧
            (g1 SEQ_RESTRICT I_U_V = NULL_SEQ) ∧
            (g2 SEQ_RESTRICT I_U_V = NULL_SEQ))) |
          (∃g.
           (t2 SEQ_CONC g) IN TR ∧
           (g SEQ_RESTRICT I_U_V = NULL_SEQ)))))
```

Therefore, our definition considers a system to be restrictive with respect to a user's view if, for two traces *t1* and *t2* that appear the same in the view, and an event *e* such that *t1* followed by *e* is a legal trace of the system, then:

- If *e* is in the user's view and *e* is an input event, *t2* followed by *e* is a legal trace for the system.
- If *e* is in the user's view and is a non-input event, there exists a sequence of events, containing *e*, and not containing any input events or events in the user's view, such that *t2* followed by this sequence is a legal trace of the system.
- If *e* is not in the user's view, there exists a sequence of events (possibly the NULL sequence) not containing any input events or events in the user's view, such that *t2* followed by this sequence is a legal trace of the system.

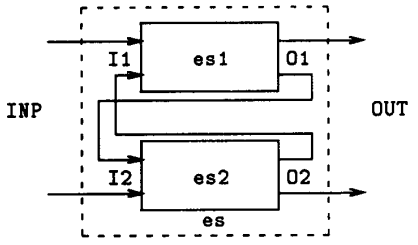

Figure 1: The hook-up of two event systems

Proving properties with respect to this definition can be accomplished using induction on the length of the traces *t1* and *t2*. This induction naturally falls out of the recursive definition of sequences presented in Appendix A, and works well with our definition based on single events. Using the HOL system we have proven the theorem below that state for input-total systems our definition implies McCullough's original definition, thus assuring that our definition relates to McCullough's.

```
RESTRICT_IMP_RESTRICTIVENESS
⊢ ∀v es.
   IS_INPUT_TOTAL es ⟹
     RESTRICT v es ⟹  IS_RESTRICTIVE v es
```

4.2 Hook-Up of Restrictiveness

The hook-up of two event systems to create a third event system is used in the construction of complicated systems. The system specification can be decomposed into a collection of simpler components that behave as the system when they are hooked-up. An advantage of this decomposition is that for some predicates (hook-up predicates) one can show that if the components of a system satisfy the predicate, then the hook-up of those components will also satisfy that predicate. This enables one to prove the predicate for a simple component specification and use the hook-up property to prove the predicate for the hook-up of the components.

If the two event systems *es1* and *es2* are running in parallel and communicate by sharing some events then the combination of these two component systems is their *hook-up*. The two component systems only share events that are an input of one component and an output of the other component. Figure 1 presents a block diagram showing the hook-up of two event systems.

```
IS_HOOK_UP_INPUT_TOTAL
⊢def ∀es es1 es2.
   IS_HOOK_UP_INPUT_TOTAL es es1 es2 =
   (let EV = EVENTS es   and INP = INPUTS es
    and OUT = OUTPUTS es and TR = TRACES es
    and E1 = EVENTS es1  and I1 = INPUTS es1
    and O1 = OUTPUTS es1 and T1 = TRACES es1
    and E2 = EVENTS es2  and I2 = INPUTS es2
    and O2 = OUTPUTS es2 and T2 = TRACES es2 in
     ((EV = E1 UNION E2) ∧
      (INP = (I1 DIFF O2) UNION (I2 DIFF O1)) ∧
      (OUT = (O1 DIFF I2) UNION (O2 DIFF I1)) ∧
      (I1 INTER I2 = EMPTY) ∧
      (O1 INTER O2 = EMPTY) ∧
      (((E1 DIFF I1) DIFF O1) INTER E2 = EMPTY) ∧
      (((E2 DIFF I2) DIFF O2) INTER E1 = EMPTY) ∧
      (∀x.
        x IN TR =
        SEQ_IN_SET_STAR x EV ∧
        (x SEQ_RESTRICT E1) IN T1 ∧
        (x SEQ_RESTRICT E2) IN T2)))
```

The above equation is the HOL definition of the hook-up predicate presented in [8]. There are several clauses in this predicate to determine if *es* is the hook-up of the component systems *es1* and *es2*.

- The events of *es* are the events of *es1* and *es2*.
- The inputs of *es* are the inputs *es1* and *es2* that are not shared events.
- The outputs of *es* are the outputs of *es1* and *es2* that are not shared events.
- The input–output pairs of *es1* and *es2* are the only shared events.
- The traces of *es* are all sequences of events of *es* such that when restricted to the events of any component system they are a trace of that component system.

This predicate is adequate for a system consisting of two *input-total* components, which are always ready to receive new input events. To expand the predicate to *non-input-total* components we only need to add the two clauses shown below, which state that for shared events, if it is possible for one component to send an output then the other component must be ready to receive it.

```
IS_HOOK_UP
⊢def ∀es es1 es2.
   IS_HOOK_UP es es1 es2 =
   ...   % same as IS_HOOK_UP_INPUT_TOTAL def. % ∧
      (∀t x.
        t IN TR ∧ x IN O1 ∧
        (ENTRY x(t SEQ_RESTRICT E1)) IN T1 ⟹
          ((ENTRY x t) SEQ_RESTRICT E2) IN T2) ∧
      (∀t x.
        t IN TR ∧ x IN O2 ∧
        (ENTRY x(t SEQ_RESTRICT E2)) IN T2 ⟹
          ((ENTRY x t) SEQ_RESTRICT E1) IN T1)
```

We have proven that our definition of restrictiveness is a composable property, using our modified definition of hook-up for non input-total systems. The following theorem is the result of this proof.

```
RESTRICT_IS_HOOK_UP
⊢ ∀es es1 es2 V.
   RESTRICT V es1 ∧ RESTRICT V es2 ∧
   IS_HOOK_UP es es1 es2 ⟹
     RESTRICT V es
```

Using this proof we can show that any event system that is the hook-up of two event systems is restrictive if it's components are restrictive. Taking this a step further we can show that any system that is a hook-up of a collection of event systems is restrictive if all the components are restrictive.

5 Specifying and Verifying System Components

In this section we discuss the development of system components. When we first started work on this phase of our research, we attempted to formalize the delay-queue specification and proofs presented by McCullough [10]. We quickly confirmed that McCullough's claims about the complexity of the proofs. Attempting to repeat proofs of this complexity for several different components with simple specifications was not appealing.

We felt that the development of generic building blocks would simplify the proof process. We decided to classify the system components into four classes. Three of the classes are sequential (i.e. "first come first serve") and the last is scheduled. All of the components described in this paper are input-total, if we expand on the work in [14] we can specify non-input total or *input limited* versions of these classes. We name these classes based on the way they relate inputs to non-inputs, on a one-to-one, one-to-many, many-to-one or many-to-many basis.

- 1-1 (Set Filter). This class consists of components that process messages on an first-come first-serve basis. Each input event is either ignored or transformed by a simple constant function and sent as an output event.
- 1-M (Generator). This class consists of components that process messages identically to the 1-1 class except that each input event generates a sequence of non-input events according to the function.
- M-1 (Programmable Filter). This class consists of components that process messages on a first-come first-serve basis. Each input event is considered either a control event or a data event. Control events do not generate any additional events, but can change the future behavior of the system. Data events are transformed based on the history of previous events, and sent as non-input events.

- M-M (Scheduled Programmable Filter). This class is the catch-all. It takes input events, and process them based on the history of events. The inputs are not necessarily processed on a first come first serve basis, and the transformations are not necessarily constant.

Once we defined these classes of components we developed a generic specification for components in each class. Every specific component can be specialized from the generic definition. Not only does this standardize and simplify the specification of components, it also aids in the proof of properties about the components. Since each component is a specific instance of the generic component, a proof of a property about the generic component will automatically apply to the specific instance. For each of our classes we proved, for a collection of views, that the generic specification is restrictive for those views.

To demonstrate our techniques we also specialized a collection of parameterized components for each class, and showed which parameters give us valid components. The remainder of this section presents in detail the steps necessary for developing the set filter class of components.

The set filter consists of components that take inputs one at a time, process them and then send an output if necessary. We can summarize the set filter with the following properties:

1. Input-total. The components are always ready to receive inputs.
2. FIFO. The components process inputs on a first-come first-serve basis.
3. Filter. Each input message is either ignored or transformed into a single output.

```
IS_SET_FILTER_DEF
⊢def ∀es fn set1.
   IS_SET_FILTER es fn set1 =
    (let EV = EVENTS es   and INP = INPUTS es
     and OUT = OUTPUTS es and TR = TRACES es  in
      ((∀x. x IN (INP INTER set1) ⟹
         (fn x) IN OUT) ∧
       (EV = INP UNION OUT) ∧
       (∀seq. seq IN TR =
          IS_SET_FILTER_TRACE seq es fn set1)))
```

To define the filter we created a predicate in the HOL logic, IS-SET-FILTER presented above. This predicate is parameterized by a set of events, a function and an event system. The event system is considered to be a filter if:

- The function maps inputs in the set to outputs.
- The events of the system are only inputs or outputs.
- The component only processes messages in the set.
- The set of traces of the event system are the traces of the filter.

The definition IS-SET-FILTER-DEF defines the predicate that determines if an event system is a filter. Although we have explicitly defined three of the fields of the event system, we still need to elaborate on the fourth field, the traces. The three definitions given below allow us to define a trace of the filter. One can see that a sequence is defined to be a valid trace of the filter if either:

- The sequence is the *Null* sequence.
- The sequence is of the form ENTRY *e seq* and *e* is a legal next event for *seq*. This means that *e* is either an input event or it is the correct output event for the next unserviced input event.

```
SET_FILTER_MSGS
⊢def (∀es set1. SET_FILTER_MSGS NULL_SEQ es set1 =
      NULL_SEQ) ∧
    (∀x s es set1.
      SET_FILTER_MSGS(ENTRY x s)es set1 =
      (x IN (INPUTS es) →
       (x IN set1 →
        ENTRY x(SET_FILTER_MSGS s es set1) |
        SET_FILTER_MSGS s es set1) |
       SEQ_TAIL(SET_FILTER_MSGS s es set1)))

IS_SET_FILTER_NEXT_EVENT
⊢def ∀e seq es fn set1.
     IS_SET_FILTER_NEXT_EVENT e seq es fn set1 =
     e IN (INPUTS es) ∨
     e IN (OUTPUTS es) ∧
     ¬(SET_FILTER_MSGS seq es set1 = NULL_SEQ) ∧
     (e = fn(SEQ_FIRST_ENTRY
             (SET_FILTER_MSGS seq es set1)))

IS_SET_FILTER_TRACE
⊢def (∀es fn set1.
      IS_SET_FILTER_TRACE NULL_SEQ es fn set1 = T) ∧
    (∀e seq es fn set1.
      IS_SET_FILTER_TRACE(ENTRY e seq)es fn set1 =
      (IS_SET_FILTER_NEXT_EVENT e seq es fn set1 →
       IS_SET_FILTER_TRACE seq es fn set1 |
       F))
```

Note that we keep track of the unserviced input events through use of the function SET-FILTER-MSGS. This function takes an entire sequence as a parameter and recursively builds a sequence of unserviced messages. All inputs are added to the end of the sequence while outputs remove an input from the front of the sequence. This function only handles the bookkeeping while the predicate IS-SET-FILTER-NEXT-EVENT handles checking whether the output events actually are correct for the next input event.

Now that we have defined what we mean by a filter we can use it in our proof of security properties. The theorem presented below states when a set filter is restrictive. It relies on the definition VISIBLE-OUT-IMP-INP

which determines if all the outputs in the view are derived from inputs also in the view [2]. Thus if a view satisfies this predicate, a filter is restrictive with respect to that view.

```
VISIBLE_OUT_IMP_INP
⊢def ∀v es fn.
    VISIBLE_OUT_IMP_INP v es fn =
    (∀x.
      x IN v ∧ x IN (OUTPUTS es) ⟹
       (∀x'. x' IN (INPUTS es) ∧ (x = fn x') ⟹
          x' IN v))

SET_FILTER_IS_RESTRICTIVE
⊢ ∀v set1 es fn.
    VISIBLE_OUT_IMP_INP v es fn ∧
    IS_SET_FILTER es fn set1 ⟹
      RESTRICT v es
```

5.1 Filter Components

In this section we describe several components that have been specialized from the set filter specification. Each of these components is defined through a predicate that determines if an event system behaves like that type of components. We then prove a theorem which determines which parameter values allow this predicate to be true. Once this is done one can easily instantiate these components with specific parameters that satisfy the conditions.

We have also shown that each component satisfies the restrictiveness property for the views described for generic filters. All the components discussed in this section share some similar properties that are a product of the event system model, the filter specification or exist for convenience in the proofs.

- All external events are labeled pairs. The first element of the pair is the name of the I/O channel where the message occurs, and the second element of the pair is the actual message.
- The names of the input and output channels for this component are disjoint.
- There exist output messages for all possible input messages.
- The components are generic enough that the actual messages and names of the I/O channels can be specified upon instantiation.

We have defined six different components that are specializations of the set filter classification. Each of these components has been formally specified, and verified as being restrictive for a set of views.

The *Delay Queue* is the simplest of the filter specializations. It receives messages through an input port, and at some later time sends them out an output port. It is parameterized by the names of the two ports and the set of messages that it handles.

The *Simple Filter* component is very similar to the delay queue component with an additional set of messages as a parameter. Any input message received by the simple filter that is not in this set is ignored and does not generate any output message.

The *Transformer* component is also very similar to the delay queue component with a function as an additional parameter. This function maps the input messages to corresponding output messages.

The *Multiplexor* component is a slightly more complicated device. It receives input messages on many channels, and places them, in order received,[3] on a single output channel. It is parameterized by a set of input port pairs and output port name. Each input port pair consists of the input port name and the set of messages that can arrive at that port.

The *De-multiplexor* component receives input messages on a single channel and places them on one of many possible output channels. It is parameterized by an input port name and a set of output port pairs. Each output port pair consists of an output port name and a set of messages that can be sent on that port. Each output port has a unique set of messages. Each input message is then uniquely routed to a particular output port.

The *Switch* component is effectively a combination of the multiplexor, de-multiplexor and transformer components. It receives messages on many input ports and sends a corresponding output to one of many output ports. The switch is parameterized by a set of input port pairs, a set of output port pairs and a transformational routing function. Each pair consists of the port name and the set of messages that can occur on that port. The routing function generates an output message for each input message and determines the correct output port for that message.

5.2 Specifying Components

Each component is specified following a standard method. In this section we present the method as it was applied to the delay queue component. This set of specifications is the first step of our method.

For this first step we present any auxiliary definitions needed to define the components, which for the delay queue is the function DELAYQ-FN. This function maps input events to output events. This mapping is accomplished by changing the I/O port identifier of the event to the output port identifier *o-port*.

[2] We don't want a system to permit a user to see an output event that was generated from inputs the user was not permitted to view

[3] Recall that all events are separate so the system can always coose one to be first.

```
DELAYQ_FN
⊢def ∀o_port x. DELAYQ_FN o_port x = o_port,SND x

DELAYQ_P
⊢def ∀es msgs i_port o_port.
   DELAYQ_P es msgs i_port o_port =
   (∀x. x IN (INPUTS es) =
        (FST x = i_port) ∧ (SND x) IN msgs) ∧
   (∀x. x IN (OUTPUTS es) =
        (FST x = o_port) ∧ (SND x) IN msgs) ∧
   ¬(i_port = o_port) ∧
   IS_SET_FILTER es(DELAYQ_FN o_port)(EVENTS es)

DELAYQ_DEF
⊢def ∀msgs i_port o_port.
   DELAYQ msgs i_port o_port =
    (εes. DELAYQ_P es msgs i_port o_port)
```

These auxiliary definitions are used to simplify the main predicate, in this case DELAYQ-P [4]. This predicate determines if an event system satisfies a relationship with the set input parameters. For the delay queue those parameters are a set of messages *msgs*, and I/O port identifiers *i-port* and *o-port*. As can be seen in the definition, an event system is considered a delay queue if:

- The set of inputs consists of pairs, where the first element of the pair is the input port identifier *i-port*, and the second element of the pair is a message. There exists a pair for each message in the parameter *msgs*.
- The set of outputs consists of pairs, where the first element of the pair is the output port identifier *o-port*, and the second element of the pair is a message. There exists a pair for each message in the parameter *msgs*.
- The port identifiers are different.
- The event system is a filter.

The characteristic predicate is then used in the creation function. This function generates an event system, based on a set of parameters, that satisfies the characteristic predicate. For the delay queue, this function is defined in DELAYQ-DEF. Note that the definition uses the Hilbert Choice operator, ε, which will return an event syhstem that satisfies the characteristic predicate. If none satisfy the predicate, and arbitrary event system is returned.

To avoid the arbitrary choice of an event system we develop an existence theorem that states under which conditions there exists an event system that satisfies the characteristic predicate.

[4] We use the notation *comp-name*-P for the components characteristic predicate, *comp-name*-DEF for the component creation function.

```
DELAYQ_EXISTS
⊢ ∀msgs i_port o_port.
   ¬(i_port = o_port) ⟹
    (∃es. DELAYQ_P es msgs i_port o_port)

DELAYQ_MEMBER_LEMMA
⊢ ∀msgs i_port o_port.
   ¬(i_port = o_port) ⟹
    DELAYQ_P(DELAYQ msgs i_port o_port)
        msgs i_port o_port
```

The above DELAYQ-EXISTS theorem states that when the input and output port identifiers are distinct, there exists an event system that satisfies the characteristic predicate of a delay queue. Using this theorem we can now prove an existence lemma, DELAYQ-MEMBER-LEMMA, which states that if the input and output port identifiers are distinct, then the creation function DELAYQ always returns an event system which satisfies the characteristic predicate DELAYQ-P.

```
DELAYQ_RESTRICTIVE
⊢ ∀v msgs i_port o_port.
   let DQ = DELAYQ msgs i_port o_port in
    (¬(i_port = o_port) ∧
     VISIBLE_OUT_IMP_INP v DQ(DELAYQ_FN o_port) ⟹
      RESTRICT v DQ)
```

Now that we know when a delay queue exists, we need to show when it satisfies the security property. The above theorem states when a delay queue is restrictive. Here we once again use the predicate VISIBLE-OUT-IMP-INP since a delay queue is an specialization of a filter.

The restrictiveness theorem, in this case DELAYQ-RESTRICTIVE, is generated from the existence theorems and the generic class restrictiveness theorem. In this thoerem one can see that we define the event system component, in this case DQ, using the creation function. Using this component we develop the two clauses of the antecedent of the implication. The first clause consists of the antecedent in the component existence theorem, in this case the requirement that the input and output port identifiers are different. The second clause consists of the view restriction predicate from the generic class restrictiveness theorem. In this case the predicate is VISIBLE-OUT-IMP-INP, parameterized by the component DQ and its output function DELAYQ-FN. Given that these conditions hold we then know that the component DQ is restrictive.

Although the work presented here was generated interactively in the HOL system, several of the theorems and predicates presented in this section, such as the restrictiveness theorem and existence lemma, can be automatically generated and proven in the HOL system. Given a formal format for the steps presented above one could create a more automated system that would guide the user through this process, automatically generating the necessary proof goals, theorems and definitions as they

are needed. We are currently developing such a format and generation techniques for our collection of generic classes and components. The Ulysses system [13] provides such automatic generation for specific instances of components.

6 The Rushby Randell Distributed System

This section contains a description of a real system specification that was proven using the generic classes and components developed with the method presented above. The specification we use is based on the Distributed Secure system presented in [15].

This system consists of a collection of untrusted single level hosts, a network and a multilevel file server. A diagram of such a system, without the file server is shown in Figure 2. The major component of this system is the Trusted Network Interface Unit (TNIU). This device is an intermediary that sits between hosts on a network and the actual network. The TNIU prevents information on the network from reaching the host if the information does not come from another host with the same security classification level, the exception being the Multi-Level file server. This server is actually a collection of single level servers and a trusted intermediary that routes the messages to the servers.

The implementation of the TNIU requires that there exist a collection of security partitions. Each host and its corresponding TNIU belong to exactly one partition.

To specify the system one needs to define the scope of the system. This includes the set of data messages, security partitions, host ids, and user view's. One also needs to define functions that determine the security classification level for host ids, system events, and data messages.

Following the method presented in this paper, events must be characterized as pairs. The first element of the event pair is the name of the I/O channel on which the event occurs, the other element is the message. For a network system each message must consist of at least three components. The first component is the source host ID, the second component is the destination host ID and the third component is the data message. One can define functions that will return the correct component for a given message.

Views in this system correspond to the security partitions. There is one view per partition. That view contains all events such that the classification level of the event is dominated by the view's classification level. One determines the classification level of an event based on the classification level of the source host.

Using our method we have developed a high level specification of the system and shown that it satisfies restrictiveness[5]. Describing this system in our classification system is straightforward.

The TNIU consists of two components, a transformer that places the messages on the network and a filter that pulls messages off the network. The network is a broadcasting device that takes single input messages and places them on every single output port (a 1-M generator). The file server is a simple database (a M-1 programmable filter).

[5] Due to limitation of the security model we are using, we do not include any of the encryption requirements in our specification.

6.1 TNIU

In this section we describe how we specify a single TNIU for the system and show that it satisfies restrictiveness. After accomplishing this for all components of the system it is a simple matter to hook them up and show that the whole system satisfies restrictiveness.

In each predicate there exists the variable *sys*, which is used to characterize the system. This variable is an abstraction of the secure system we are modeling. Components of the variable consist of the set of security partitions, a hierarchy function on those partitions, a mapping of host ids to the partitions and the labeling of I/O ports for the various system components. The user can instantiate this variable to any system, as long as the naming of I/O ports is unique, there exists no direct feedback for single components, and the hierarchy on the security partitions is a partial ordering.

The predicate TNIU-P shown below is the HOL characteristic predicate for a TNIU. This predicate is true if the TNIU is the hook-up of a filter component and a transformer component. The theorem TNIU-RESTRICTIVE also shown below states that a given event system that satisfies the TNIU characteristic predicate is restrictive.

```
TNIU_P
⊢def ∀sys es host_id.
   TNIU_P sys es host_id =
    (∃es1 es2.
     (IS_HOOK_UP es es1 es2) ∧
     (TNIU_FILTER_P sys es1 host_id) ∧
     (TNIU_TRANS_P sys es2 host_id))


TNIU_RESTRICTIVE
⊢ ∀sys v es host_id.
   (TNIU_P sys es host_id) ∧
   (VISIBLE_OUT_IMP_INP v es
     (TNIU_OUT_FN sys host_id)) ⟹
    RESTRICT v es
```

The theorem TNIU-RESTRICTIVE depends on two TNIU specific definitions. The first, TNIU-P, is the TNIU characteristic predicate defined earlier, while the other, TNIU-OUT-FN, is a function that returns the output function for the TNIU for a given host id. This output function is used to ensure that all outputs in the view are derived from inputs in the view.

The proof of the theorem TNIU-RESTRICTIVE is based directly on the composability of the restrictiveness property. Given that the TNIU is the hook-up of two components, if one has already shown that these components are restrictive then the TNIU is restrictive.

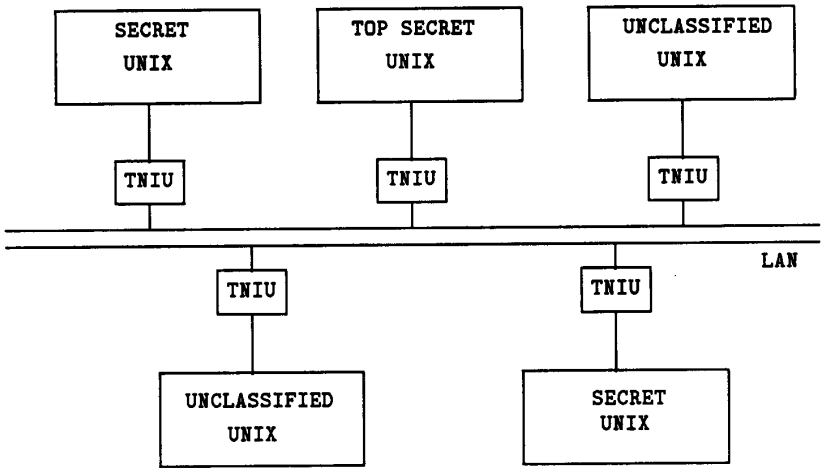

Figure 2: A distributed secure system.

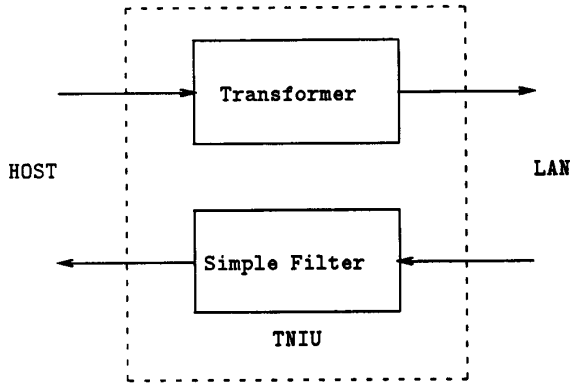

Figure 3: Decomposition of a TNIU.

The actual decomposition of the TNIU into these components is shown in Figure 3. Here we have both the filter and the transformer components of the TNIU hooked-up (although not sharing any communication channels) into a single TNIU.

TNIU Filter. The filter component of the TNIU allows messages to pass through if the destination of the message is the TNIU host and if the security classification of the source host is equivalent to the security classification of the destination host. This filter is a direct instantiation of the simple filter component described previously. The set of messages that parameterizes the simple filter consists of all messages whose destination host id is the TNIU host id and whose source host id has a classification level equivalent to that of the TNIU host.

The predicate TNIU-FILTER-P presented below is the HOL characteristic predicate for the TNIU filter component. This predicate is true if a given event system is the filter component of the TNIU for a given host id. This TNIU filter component is actually an instantiation of the generic 1-1 simple filter component.

```
TNIU_FILTER_P
⊢def ∀sys es host_id.
   TNIU_FILTER_P sys es host_id =
    SIMFIL_P es (TNIU_LEGAL_SET sys host_id)
      (TNIU_MSGS sys)
      (abs_TNIU_HOST_IN sys host_id)
      (abs_TNIU_NET_OUT sys host_id)
```

In this definition the instantiation of the simple filter includes four TNIU specific parameters. TNIU-LEGAL-SET is the function that returns the set of all possible system messages sent to a given host id. TNIU-MSGS is the set of all possible system messages. abs-TNIU-HOST-IN is the function that returns the I/O channel name of the input port for a given host id. abs-TNIU-NET-OUT is the function that returns the I/O channel name of the network output port that connects to the TNIU for a given host id.

The theorem, TNIU-FILTER-RESTRICTIVE, given below states that the TNIU filter component is restrictive. It depends on two TNIU specific definitions. The first, TNIU-FILTER-P, is the TNIU filter component characteristic predicate defined earlier, while the second, TNIU-FILTER-OUT-FN, is a function which returns the output function of the TNIU filter component for a given host id. This output function is a simple identity function that is used to ensure that all outputs in a view are derived from inputs in that view.

```
TNIU_FILTER_RESTRICTIVE
⊢ ∀sys v es host_id.
   TNIU_FILTER_P sys es host_id ∧
   VISIBLE_OUT_IMP_IMP v es
     (TNIU_FILTER_OUT_FN sys host_id) ⟹
    RESTRICT v es
```

TNIU Transformer. The transformer portion of the TNIU places the TNIU host id in the source host field of the message. This ensures that the data in that field is correct regardless of what the host placed in the message. This transformer is a direct instantiation of the transformer component described previously. The transformation function that parameterizes the component is a function that places the TNIU host id in the source field of the message and redirects the message to the network.

```
TNIU_TRANS_P
⊢def ∀sys es host_id.
   TNIU_TRANS_P sys es host_id =
    TRANS_P es (TNIU_MSGS sys)
      (TNIU_SET_SRC_FN sys host_id)
      (abs_TNIU_HOST_OUT sys host_id)
      (abs_TNIU_NET_IN sys host_id)
```

The predicate TNIU-TRANS-P presented above is the HOL characteristic predicate for the TNIU transformer component. This predicate is true if a given event system is the transformer component of a the TNIU for a given host id. The TNIU transformer component is actually an instantiation of the generic 1-1 transformer component.

In this definition the instantiation of the transformer includes four TNIU specific parameters. TNIU-MSGS is the set of all possible system messages. TNIU-SET-SRC-FN is the function that returns the transformation function for a given host id. This transformation function ensures that the source host id in the message is the host id. abs-TNIU-HOST-OUT is the function that returns the I/O channel name of the output port for a given host id. abs-TNIU-NET-IN is the function that returns the I/O channel name of the network input port that connects to the TNIU for a given host id.

```
TNIU_TRANS_RESTRICTIVE
⊢ ∀sys v es host_id.
   TNIU_TRANS_P sys es host_id ∧
   VISIBLE_OUT_IMP_IMP v es
     (TNIU_TRANS_OUT_FN sys host_id) ⟹
    RESTRICT v es
```

The theorem, TNIU-TRANS-RESTRICTIVE, shown above, states that the TNIU transformer component is restrictive. It depends on two TNIU specific definitions. The first, TNIU-TRANS-P, is the TNIU transformer component characteristic predicate presented earlier, while the second, TNIU-TRANS-OUT-FN, is a function which returns the output function of the TNIU transformer component for a given host id. This output function is a simple identity function that is used to ensure that all outputs in a view are derived from inputs in that view.

6.2 Network

The network can be specified as a broadcasting device. It receives input messages from one of the TNIUs in the system. This message is the duplicated and sent to all the other TNIUs in the system.

Using the 1-M generator classification one can specify a generic component broadcasting device that behaves in this manner. This component can then be shown to satisfy the restrictiveness property. One then instantiates the broadcasting component for the particulars of this system. Due to the fact that the generic component satisfies restrictiveness one can easily show that this instantiation satisfies restrictiveness.

6.3 File Server

The file server is the most complicated component of the system, but can be specified as a simple database device. It receives messages from its TNIU and sends responses back through the TNIU. The responses are determined by previous messages received by the server. Possible messages from hosts of the system include *publish* and *acquire* requests for access to files in the server.

The server originally described by Rushby and Randell consists of a simple switch (multiplexor, de-multiplexor pair) and a collection of untrusted single level file servers. This configuration can easily be defined using a 1-1 filter switch component for the switch and a simple M-1 programmable filter database component for the file servers. Unfortunately this does not define the possible behavior of untrusted file servers, in that it requires the file servers perform as expected. An *untrusted* file server can change the contents of a file based on the history of acquire requests it has received.

To define the system as it should behave one defines the whole file server as a M-1 programmable filter database component. This component receives publish and acquire messages from the network. It will then respond as defined by the security policy. If a Secret level host publishes a file *File1* a Top Secret level host can acquire *File1* but an Unclassified level host cannot.

7 Conclusions

In this paper we presented a generalization of McCullough's restrictiveness model. We used this generalization and an event-based model of computer systems defined in the HOL system [6] to prove the composability of the model and several other properties about the model. This composability permits us to use modular specification techniques in the development of a system, and then when all components are shown to satisfy restrictiveness we automatically know that their composition satisfies restrictiveness. We then developed a set of

generalized classes of system components and showed for which families of user views they satisfied restrictiveness. Using these classes we developed a collection of general system components that are specializations of one of these classes and showed that they also satisfied the security property. These proofs were easy to develop due to the existence of proofs showing that the general models satisfy the restrictiveness property. We then defined a sample distributed secure system, based on the Rushby and Randell TNIU, and showed how our mechanized system could be applied to reasoning about it.

Continuing this work, we plan to incorporate some of the ideas presented by Rosenthal in [13, 14] to expand our mechanized system classifications to include non-input total systems. Using these expanded classifications we will apply our to a network mail server [12] and a secure labeler [16]. Once we have developed an adequate library of generic system component specifications and used them to specify sample systems, we will apply our method to a kernel for a secure distributed system. The method we have developed, and the composability of the security model we are using will enable us to fragment this task into several simple components. We expect all of these components to fit into one of our generic component classifications.

Acknowledgements

We would like to thank Phillip Windley for his well written description of the HOL system and his help on using HOL in general. We would also like to thank Amit Jasuja, Sara Kalvala, Jing Pan and Tom Schubert for their help and discussions during the development of the HOL theories and proofs presented in this paper.

References

[1] J. Alves-Foss and K. Levitt. *A Model of Event Systems in Higher Order Logic: Sequence and Event System Theories.* Technical Report CSE-90-45, Division of Computer Science, University of California, Davis, November 1990.

[2] J. Alves-Foss and K. Levitt. *A Security Property in Higher Order Logic: Restrictiveness and Hook-Up Theories.* Technical Report CSE-90-46, Division of Computer Science, University of California, Davis, December 1990.

[3] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *HDL Descriptions to Guaranteed Correct Circuit Designs*, Elsevier Scientific Publishers, 1987.

[4] A.J. Camilleri. Mechanizing CSP trace theory in Higher Order Logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, September 1990.

[5] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[6] M. Gordon. *A Proof Generating System for Higher-Order Logic.* Technical Report 103, University of Cambridge Computer Laboratory, January 1987.

[7] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London, 1985.

[8] D. McCullough. *Foundations of Ulysses: The Theory of Security.* Technical Report RADC-TR-87-222, Odyssey Research Associates, Inc., July 1988.

[9] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–186, 1988.

[10] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proc. IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[11] R.A. Milner. *Communicating and concurrency.* Prentice Hall, New York, 1989.

[12] S. Owicki. Specification and verification of a network mail server. In F.L. Bauer and M. Broy, editors, *Program Construction*, pages 198–234, Springer-Verlag, 1979.

[13] D. Rosenthal. Implementing a verification methodology for McCullough security. In *Proc. Computer Security Foundations Workshop*, pages 133–140, IEEE Computer Society Press, June 1989.

[14] D. Rosenthal. Security models for priority buffering and interrupt handling. In *Proc. Computer Security Foundations Workshop*, pages 91–97, IEEE Computer Society Press, June 1990.

[15] J. Rushby and B. Randell. A distributed secure system. *IEEE Computer*, 16(7):55–67, 1983.

[16] W.D. Young, P.A. Telega, W.E. Boebert, and R.Y. Kain. A verified labeler for the Secure Ada Target. In *Proc. National Computer Security Conference*, 1986.

A The Sequence Theory

```
let sequence = define_type 'sequence'
   'sequence = NULL_SEQ | ENTRY * sequence';;
```

The HOL definition for the sequence data type abstraction is shown above. In this definition one can observe that a sequence is defined as either as NULL-SEQ or as a sequence with an entry appended to it. This is similar to a Lisp *cons* operation except that the entry is added to the end of the sequence. The "*" in the second disjunction of the definition is a place holder for a type specifier, it permits us to define sequences as a ordered collection of objects of type "*". We can later instantiate a sequence to consist of objects of a specific type (i.e. int, bool, lists, or even more sequences).

The HOL system automatically generates the abstraction and representation axioms for the sequence data type, permitting us to deal with the abstraction and ignore the internal representation.

```
SEQ_LAST_ENTRY
⊢def ∀x s. SEQ_LAST_ENTRY(ENTRY x s) = x

SEQ_HEAD
⊢def (SEQ_HEAD NULL_SEQ = NULL_SEQ) ∧
  (∀x s. SEQ_HEAD(ENTRY x s) = s)

SEQ_FIRST_ENTRY
⊢def ∀x s.
   SEQ_FIRST_ENTRY(ENTRY x s) =
    ((s = NULL_SEQ) → x | SEQ_FIRST_ENTRY s)

SEQ_TAIL
⊢def (SEQ_TAIL NULL_SEQ = NULL_SEQ) ∧
  (∀x s.
    SEQ_TAIL(ENTRY x s) =
    ((s = NULL_SEQ) → NULL_SEQ |
                      ENTRY x(SEQ_TAIL s)))
```

Once we define the recursive data type we need some simple definitions for manipulating elements of the sequence type. As seen above we defined four simple extractors, which take a sequence as a parameter and returns a component of the sequence. SEQ-LAST-ENTRY returns the last element added to a sequence and is undefined for the NULL sequence. SEQ-HEAD is the opposite of SEQ-LAST-ENTRY in that it returns the sequence that existed prior to the last element being added and returns NULL-SEQ for a NULL sequence. SEQ-FIRST-ENTRY returns the first element added in a sequence and is undefined for the NULL sequence. SEQ-TAIL is the converse of SEQ-FIRST-ENTRY in that it returns the sequence that exists after the first entry and returns NULL-SEQ for the NULL sequence.

```
SEQ_LENGTH
⊢def (SEQ_LENGTH NULL_SEQ = 0) ∧
  (∀x s. SEQ_LENGTH(ENTRY x s) = SUC(SEQ_LENGTH s))

SEQ_CONC
⊢def (∀s. s SEQ_CONC NULL_SEQ = s) ∧
  (∀s s1 x. s SEQ_CONC (ENTRY x s1) =
    ENTRY x(s SEQ_CONC s1))

SEQ_RESTRICT
⊢def (∀e. NULL_SEQ SEQ_RESTRICT e = NULL_SEQ) ∧
  (∀x s e.
    (ENTRY x s) SEQ_RESTRICT e =
    (x IN e → ENTRY x(s SEQ_RESTRICT e) |
              s SEQ_RESTRICT e))
```

We have defined several functions on elements of the sequence abstract data type. We present three of these functions above. SEQ-LENGTH is a simple recursive function that takes a sequence as a parameter and returns its length. SEQ-CONC is an infix operator that takes two sequences as parameters and returns their concatenation (similar to a Lisp *append* operation). SEQ-RESTRICT in an infix operator that takes a single sequence and a set of elements as parameters and returns the sequence with elements not in the set removed from the sequence.

```
PREFIX
⊢def (∀a. a PREFIX NULL_SEQ = (a = NULL_SEQ)) ∧
  (∀a b x.
    a PREFIX (ENTRY x b) =
     ((a = ENTRY x b) → T | a PREFIX b))

PPREFIX
⊢def ∀s1 s2.
    s1 PPREFIX s2 = ¬(s1 = s2) ∧ s1 PREFIX s2

SEQ_IN_SET_STAR
⊢def (∀set1. SEQ_IN_SET_STAR NULL_SEQ set1 = T) ∧
  (∀x s set1.
    SEQ_IN_SET_STAR(ENTRY x s)set1 =
    (x IN set1 → SEQ_IN_SET_STAR s set1 | F))
```

We have also defined some useful predicates based on sequences. We present four of these above. PREFIX is an infix operator which is true if the first sequence is a prefix of the second sequence. PPREFIX is an infix operator which is true if the first sequence is a proper prefix of the second sequence. SEQ-IN-SET-STAR takes a sequence and a set and is true if all elements of the sequence are members of the set.

In the course of our research, we have developed a large library of functions and predicates for the sequence abstract data type. To aid our proof efforts, we have also proven many theorems related to sequences and their selectors, predicates and functions. A complete listing of the library is given in [1].

B The Event system Theory

The event system is a polymorphic structure consisting of four fields. Three of the fields are sets of elements, and the fourth is a set of sequences of elements. These fields represent the set of events of the system, set of input events of the system, set of output events of the system, and the set of all valid traces (sequences of events) of the system. The inputs and outputs must be subsets of the set of events and traces must be sequences of the events. The valid traces represent all the possible sequences of events that could have occurred in the history of the event system.

To develop the event system abstract data type we need to first define a characteristic predicate that will determine the relationship between the components of the event system and a predicate that determines our representation of the abstract data type. These predicates are presented below.

```
IS_EVENT_SYSTEM
⊢def ∀events inputs outputs traces.
   IS_EVENT_SYSTEM events inputs outputs traces =
    inputs SUBSET events ∧
    outputs SUBSET events ∧
    DISJOINT inputs outputs ∧
    (∀s e. (ENTRY e s) IN traces ⟹  s IN traces) ∧
    (∀s. s IN traces ⟹  SEQ_IN_SET_STAR s events)

IS_EVENT_SYSTEM_REP
⊢def ∀r.
   IS_EVENT_SYSTEM_REP r =
    (∃ev in out tr.
     (r = ev,in,out,tr) ∧
     IS_EVENT_SYSTEM ev in out tr)
```

The predicate IS-EVENT-SYSTEM determines if a collection of sets can be combined into an event system. This is true only if the inputs and outputs are disjoint subsets of the events, and the traces are event separable sequences of events (i.e. (ENTRY *e s*) IN *traces* $\Longrightarrow s$ IN *TRACES*). The predicate IS-EVENT-SYSTEM-REP determines if a four-tuple of sets represents a valid event system. This is true only if the components of the four-tuple map to the four fields of the event system and satisfy the characteristic predicate. These two predicates are used to define the event system abstract data type in the HOL system.

```
EMPTY_EVENT_SYSTEM_REP
⊢def EMPTY_EVENT_SYSTEM_REP = EMPTY,EMPTY,EMPTY,EMPTY

EMPTY_ES_REP_IS_REP
⊢ IS_EVENT_SYSTEM_REP EMPTY_EVENT_SYSTEM_REP

ES_EXISTS_THM
⊢ ∃x. IS_EVENT_SYSTEM_REP x

event_system_AXIOM
⊢ ∃rep. TYPE_DEFINITION IS_EVENT_SYSTEM_REP rep
```

To complete the creation of the abstract data type we need to prove an existence theorem that states that there exists a value that represents an element of the abstract data type. The first definition, EMPTY-EVENT-SYSTEM-REP, is a representation of the EMPTY event system. EMPTY-ES-REP-IS-REP is the theorem that proves that this definition represents an event system. We use this theorem to prove the existence theorem ES-EXISTS-THM. The HOL system then automatically generates the abstract data type axiom event-system-AXIOM. It also generates mapping operators between events systems and their representations.

To access the fields of the abstract data type one needs selectors. The operators enable the user to access the fields of the event system while ignoring the actual representation being used. The four field selectors are presented below.

```
EVENTS
⊢def ∀es.
    EVENTS es = FST(REP_event_system es)

INPUTS
⊢def ∀es.
    INPUTS es = FST(SND(REP_event_system es))

OUTPUTS
⊢def ∀es.
    OUTPUTS es = FST(SND(SND(REP_event_system es)))

TRACES
⊢def ∀es.
    TRACES es = SND(SND(SND(REP_event_system es)))
```

Through the course of our research we have developed a large library of theorems and definitions related to the event system. A complete listing of this library is given in [1].