

Presented at the Second International Conference on  
Formal Methods and Models for Codesign (MEMOCODE 2005)

## Extended Abstract: Organizing Automaton Specifications to Achieve Faithful Representation \*

Elizabeth Leonard and Myla Archer  
Center for High Assurance Computer Systems, Code 5546  
Naval Research Laboratory, Washington, DC 20375 USA  
{archer,leonard}@itd.nrl.navy.mil

### 1 Introduction

**Motivation.** In using models for verification, an important question is how faithful a model is to the thing modeled. In 2002 and 2003, we specified two applications as automaton models: the TESLA multicast stream authentication protocol [7] and a portion of the Security-Enhanced Linux (SELinux) operating system [6]. Both models used a single reference variable to capture essentially all of the information about the actual state of the system. Additional state variables in the models were defined to provide access to some of the reference variable information in a form that made specifying system transitions and reasoning about the system easier. Having used the same organizational approach to advantage in defining automaton models for two different applications, we wondered if this approach would have benefits in other applications. We decided to specify the well-known IEEE 1394 leader election algorithm (Tree Identify Protocol) [5] using this modeling technique to see if it would provide any benefit.

**The reference variable approach.** We call our approach to organizing specifications of automaton models the *reference variable approach*. A *reference variable* is a variable that captures most of the essential information about an automaton state. Variable types of sufficient complexity for this purpose typically involve sets or functions, and thus are definable only in higher-order logic. For example, the reference variable in the SELinux model is a set of objects that are members of a recursive datatype, some of whose members themselves have set components. The objects represent the processes, files, and so on currently present in the operating system. Thus, the set of objects naturally captures the state of an operating system.

For modeling any system using our approach, the specifier must first choose a reference variable that captures the state of the actual system in a natural way. Because the reference variable will usually be complex, auxiliary variables are helpful in defining the preconditions and effects of state transitions and expressing properties (e.g., invariants) one wants to prove about the system. Additionally, one may choose auxiliary variables with an eye towards sys-

tem verification using abstraction. Some of these auxiliary variables (*derived variables*) can be defined by expressions involving the reference variable. Other variables, which we will call *shadow variables*, have a relationship to the reference variable that is more difficult to express; for example, they may capture a part of the state that can only be defined in terms of the reference variable using existential quantification. For purposes of reasoning, the current value of a shadow variable is most easily retrieved if its value is maintained by directly updating the shadow variable along with the reference variable. A state invariant (proved only once) can establish that a particular relationship is maintained between the shadow variable and the reference variable.

### 2 Applying the Approach: Three Examples

**The TESLA Protocol.** TESLA [7] is a multicast stream authentication protocol in which the authentication of packets is based on keys, key commitments, and timing. Our model of TESLA for use in a theorem prover, based on the reference variable method, is described in [1]. TESLA assumes both an authentic sender  $S$  and an adversary sender  $A$ , where the adversary has full control over the network and tries to spoof the receiver with fake packets.

A natural reference variable to capture the current state of TESLA is the entire history of the protocol up to the last send or receive event. The history can be represented as the set of sent packets, annotated with sender, send time, and receive time if any. The shadow variables needed included the list  $PL$  of packets sent by the authentic sender and the sets of keys  $K$  and key commitments  $KC$  that were included in some sent packet.  $K$  and  $KC$  were needed to express the power of the adversary; in the model, this is done in the precondition of the adversary's send action.  $PL$  was especially important in formulating and proving two properties essential to the proof of correctness of the protocol. Its use circumvented long existence proofs involving the number of sent packets in the history that were sent by  $S$  and a packet that had been sent by  $S$  in a previous state.

**SELinux Security Policies.** SELinux [6] extends the Linux operating system with a flexible capability for security. We developed an automaton model of SELinux [3] that can be used to analyze policy specifications written in the

\*This research is funded by ONR

SELinux policy language. Because part of the policy definition is embedded in the operating system code, the model represents the operating system with the policy included.

The natural reference variable to model the operating system is a set of objects representing the processes, files, file descriptors, and so on in the system at a given time. Shadow variables keep track of the existence of various objects in the system, such as whether there is a process with a given process ID. Other shadow variables are used to easily produce the object having given identifying information; e.g., one shadow variable takes a process ID and returns the process with that process ID. The shadow variables are used in defining actions and properties, and we anticipate using them in abstractions.

We have verified several properties for the example security policy that accompanies the SELinux release. Although the policy rules take several pages to state, we have verified 16 assertions about whether certain access permissions are ever allowed that are relevant to the portion of SELinux that we have modeled. We have also proved one state invariant and have just begun to prove additional invariants about SELinux. So far, the complexity of the model has not had a noticeable impact on the theorem prover.

**Leader Election for IEEE 1394.** Verification of the leader election protocol for the IEEE 1394 FireWire bus, particularly its tree identify phase *TIP*, has been a much studied problem: see, e.g., [5] and the special issue [4]. In [2], we revisited the verification in [5]. In this work, as a new case study (*LE*), we respecified the protocol using the reference variable method and proved leader uniqueness.

The protocol represents the bus as a graph in which each node waits for “be my parent” requests from its neighbors until only one neighbor has not sent a request, at which point the node sends a “be my parent” request to its remaining neighbor. Contention to be the leader can arise when there are only two parentless nodes remaining. In specifications of the protocol without timing, a contention resolution step makes an arbitrary choice of parent.

The protocol runs on a fixed graph  $G$ . In the protocol, actions of nodes are based solely on their local knowledge of their own status and their neighbors’ status. Hence, local knowledge can be used as the reference variable. In particular, the reference variable can be expressed as an element of type `Annotated_Graph`, whose elements are essentially snapshots of the local knowledge of all nodes in  $G$ . We did not need any shadow variables to specify actions or properties.

However, we did obtain some benefits from the use of a reference variable. First is the confidence that we captured the protocol without any extraneous artifacts (e.g., message queues). Second, we are able to express connectedness and acyclicity properties and use them in proofs without going outside the formalism. Moreover, one can potentially prove properties about measures of  $G$  computed from its annota-

tions. Useful properties would include: (1) if  $G$  is finite then some defined measure of the state of *LE* is finite, and (2) every transition of *LE* decreases the measure. Such lemmas would be useful in mechanizing the proof that eventually some node will have root status.

### 3 Conclusion

We have developed an approach to organizing automaton specifications in which a reference variable captures the essential state and shadow variables are used to facilitate expressiveness. We have applied our approach in specifying automaton models for three different examples. In all three cases, the resulting specification bears an obvious close relationship to the actual system being modeled.

The specifications we produced are possible to formulate only in higher order logic. Nevertheless, our experience has shown that it is feasible to establish properties of such specifications in an interactive mechanical theorem prover. In our approach, the specification and reasoning can be organized so that the harder parts of the reasoning (e.g., involving existence proofs) need only be done once. The relationships between the reference and shadow variables are proven first. After this, many properties can be specified in terms of the simpler shadow variables, greatly simplifying their proofs. Though our method is not generally suited for use with other types of analysis tools, it can provide indirect support by establishing relationships between the specification and abstractions “farmed out” to automatic analysis tools.

The reference variable approach allows system models that are clearly faithful, provide a basis for relating separate abstractions, and permit one to express and use high level properties of systems in properties and their proofs. A specification with an obvious correlation to the actual system can be helpful for convincing evaluators in a certification process that the model accurately reflects the actual system.

### References

- [1] M. Archer. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In *Wk. on Issues in the Theory of Security (WITS’02)*, Portland, OR, Jan. 2002.
- [2] M. Archer, C. Heitmeyer, and E. Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.
- [3] M. Archer, E. Leonard, and M. Pradella. Analyzing Security-Enhanced Linux policy specifications. In *IEEE 4th Int’l Wk. on Policies for Distr. Sys. and Netwks. (POLICY 2003)*.
- [4] J. Cooke, et al., eds. *Formal Aspects of Computing*, volume 14(3). Springer, April 2003.
- [5] M. Devillers, et al. Verification of a leader election protocol—formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.
- [6] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. Tech. Rept., National Security Agency, Jan. 2, 2001.
- [7] A. Perrig, et al. Efficient authentication and signing of multicast streams over lossy channels. In *Proc. of IEEE Security and Privacy Symposium (S&P2000)*, pages 56–73, May 2000.