Server-Process Restrictiveness in HOL

Stephen H. Brackin\(^1\) and Shiu-Kai Chin\(^2\)** ***

\(^1\) Odyssey Research Associates, 301 Dates Drive, Ithaca NY 14850
\(^2\) Department of Electrical and Computer Engineering, Syracuse University, Syracuse NY 13244

Abstract. Restrictiveness is a security property of multilevel systems, guaranteeing that a system's behavior visible at one security level does not reveal the existence of inputs or outputs at other levels not dominated by this level. This paper gives a convenient language for specifying processes, an operational semantics for this language, and a collection of easy to understand, inductively defined security properties that can be used to substantially automate proving restrictiveness for buffered server processes specified in this language.

1 Introduction

Designing and evaluating the most trusted computer systems requires formally specifying and proving that these systems have specified security properties [7]. Restrictiveness, a security property developed by McCullough [10, 12, 11] is of particular interest for such an analysis. Restrictiveness is composable, meaning that a system composed of properly connected restrictive parts is itself restrictive.

The original version of restrictiveness, which we will call trace restrictiveness, was behavioral [10]; it defined system security in terms of possible sequences of input, output, and internal events. Unfortunately, proving theorems with this form of restrictiveness was extremely difficult, as it required complicated inductions over possible extensions to sequences [11, 3]. Different researchers chose two different approaches to this problem, both of which involve strengthening trace restrictiveness to provide more useful induction hypotheses:

- Alves-Foss and Levitt developed incremental restrictiveness, a condition on single-event extensions to event sequences; it implies trace restrictiveness and is itself composable [1, 2, 3, 4].
- McCullough [11, 13] developed what we will call state restrictiveness, which is usually simply called restrictiveness. State restrictiveness defines conditions on machine states that produce all of the system behavior observable at a security level, but allow no deductions about other behavior.

\* Supported by Rome Laboratory Contract F30602-90-C-0092
\** Supported by Rome Laboratory Contract F30602-92-C-0120
\*** The authors wish to thank Daryl McCullough and David Rosenthal for identifying subtle errors in earlier versions of this work.
Rosenthal [18, 19] identified conditions sufficient to guarantee state restrictiveness in the broad case of buffered server processes. A buffered process consists of a FIFO queue and a process being buffered; it saves its inputs on the queue until the process being buffered is ready to receive them. The process being buffered is a server process if it waits for input in a parameterized state and processes each input by producing zero or more outputs and then calling itself to again wait for input in a possibly different parameterized state.

Sutherland, McCullough, Rosenthal, and others [16] developed process specification languages (similar to subsets of CSP [9]) for conveniently defining state machines, and they identified syntactic analyses that could be performed on such specifications to generate conditions sufficient for establishing state restrictiveness of buffered server processes. They and others at Odyssey Research Associates developed the Romulus (nee Ulysses) design analysis tool, which can take a specification of a buffered server process and compute from it a list of conditions sufficient for establishing Rosenthal's conditions for guaranteeing state restrictiveness [16].

For largely historical reasons, the HOL version of the Romulus specification language was defined only partially, with axioms, giving extensibility but making the language and its semantics hard to understand [17]. The Romulus implementation also depended on a "meta"-level approach, generating Rosenthal-style verification conditions with an impure tactic that was itself hard to understand [17]. Further, the language's semantics required that all state-machine parameters and all messages received as inputs or sent as outputs be coerced to a single type :datatype, losing the advantages of HOL's strong typing and significantly complicating proofs of the verification conditions [17].

We developed the results presented here, using Slind's HOL90 Release 5, to address these limitations in Romulus. We modeled our process specification language, PSL, on the Romulus specification language [17], but defined it as a concrete recursive type using Melham's automated type definition facility [14] and gave it an operational semantics using Camilleri and Melham's inductive definitions package [6].

We defined security properties analogous to Rosenthal-style verification conditions using the inductive definitions package, and we proved simple theorems about these properties that can be used as rewrite rules in proofs and taken as alternative definitions of the security properties, definitions requiring no extensive knowledge of security theory or HOL. Finally, we used polymorphic concrete recursive types to impose strong typing on state-machine parameters and message contents, in the process developing a simple technique for effectively defining processes in terms of process-valued functions.

All of our definitions are conservative extensions of the HOL logic. In addition, having the actual security definitions, abstract syntax, and operational descriptions available at the object level clarifies the intended meaning of the security properties and simplifies the proofs.

We do not intend to argue here that one version of restrictiveness is preferable to another, or that our version is equivalent to another. For the sake of