Using Composition to Design Secure, Fault-Tolerant Systems

Duane Olawsky* Charles Payne† Tom Sundquist‡ David Apostal
Todd Fine

Secure Computing Corporation
2675 Long Lake Road,
Roseville, Minnesota 55113-2536

Abstract

Complex systems must be analyzed in smaller pieces. Analysis must support both bottom-up (composition) and top-down (refinement) development, and it must support the consideration of several critical properties, e.g., functional correctness, fault tolerance and security, as appropriate. We describe a mathematical framework for performing composition and refinement analysis and discuss some lessons learned from its application. The framework is written and verified in PVS [4].

1. Introduction

Complex systems must be analyzed in smaller pieces. Analysis input should be a by-product of the development process, which may be either bottom-up or top-down, so there must be support for composing smaller analysis results into larger ones or refining large results into smaller ones. In addition, the formal underpinnings must support the consideration of disparate properties, such as functional correctness, fault tolerance and security, as appropriate.

In this paper we describe a mathematical framework, called the *CSS*[1] *Framework*, for performing composition and refinement analysis.[2] The framework is written and verified in PVS [4]. We have applied it to several research examples with encouraging results. We discuss later some lessons learned from those efforts.

1.1. Related work

The CSS Framework descends from an earlier version [2] developed under the Distributed Trusted Operating Systems (DTOS) program at SCC. The DTOS framework dealt with composition only and had no support for refinement reasoning. Both frameworks are influenced heavily by the work of Abadi and Lamport on the Temporal Logic of Actions (TLA) [1, 3] and by Shankar's framework [6].

The CSS framework differs from TLA and Shankar in the following respects. First, our framework, based on *components* which are similar to TLA *formulas*, adds *agents*, because knowing the performer of actions is useful for security analysis. Second, in our framework composition invokes environment constraints automatically, unlike TLA. Finally, TLA formulas may include existential quantifiers to hide internal state. We explored the introduction of quantifiers into our framework but decided against them. They forced a great deal of complexity both in terms of verifying the framework and using it. Although quantification has advantages from a philosophical standpoint, its practical value is more suspect. In TLA the first step in a refinement proof is typically to remove the quantifiers by applying a *refinement mapping*. This step puts the proof at what is the starting point for the proof in our framework. If no refinement mapping can be found, then although the refinement may still be correct, the proof is likely to be very difficult. So, our framework makes it easier to do the refinement proofs that are potentially feasible at the expense of not supporting refinement proofs that are potentially very difficult.

2. The CSS Framework

Composition is a technique for specifying a large system by combining the specifications of smaller, simpler pieces—the system components. This technique provides advantages

*Now at Metaphase Technology Division, Structural Dynamics Research Corporation, 4201 Lexington Ave North, Arden Hills, MN 55126-6198

†Contact: *cpayne@securecomputing.com*

‡Now at Seagate Technology, 7801 Computer Avenue South, Minneapolis, MN 55435-5489

[1] *Composability for Secure Systems*, the program at SCC under which the framework was developed.

[2] This research was supported by DARPA and the Maryland Procurement Office under Air Force Research Laboratory Contract F30602-96-C-0344 and monitored by the AFRL Rome Site.

that are similar to those obtained from modular software design. It allows us to decompose the analysis of a system into the analysis of its components. Rather than analyzing the entire composite system, we focus on a single component at a time, showing that it satisfies some more localized property. We then show that the localized properties of the components together imply that the global desired property is satisfied by the system as a whole. Since the local analyses depend upon only a single component, they are reusable. They need not be redone when the component is used in a new context.

Refinement supports reasoning about a system specified at multiple levels of abstraction. Refinement analysis makes it possible to show that properties demonstrated for an abstract specification are preserved in a less abstract refinement of that specification that reflects a more detailed design. It is usually easier to prove desired system properties for an abstract specification than for a more detailed one. On the other hand, it is easier to relate a detailed specification to its implementation since it includes more design details. The goal, of course, is to know that the implementation satisfies the desired properties. Refinement analysis allows us to conclude this by proving that an abstract specification satisfies the properties, then arguing that a refined specification is consistent with the implemented system and finally, comparing the two specifications to show that the refined one is consistent with the abstract one. Thus, we only need analyze the property at the abstract level where this analysis is easiest.

The CSS framework was developed with the goals that it be small, easy to use and not too hard to verify while still providing the essential reasoning power. It consists of a set of PVS theories that must be loaded into PVS along with the specifications, which are expressed in the syntax of the framework. Supporting the use of the framework are two tools: a *specification browser* helps novice users develop and manage their specifications, and an *analyst's assistant*, actually a set of PVS strategies, facilitates many of the proofs required by the framework.

The framework's primitive is the *component*. A component specification is a state machine model defined (similar to [3]) by the following characteristics: the initial states allowed by the component, the agents acting on behalf of the component, the transitions for the component — both performed by the component *and* what the component allows its environment to perform, and fairness constraints (both weak and strong) required for the component. Transitions are defined by the initial state, the final state and the performing agent. Note that transitions performed by the environment are those for which the performing agent is not in the component's agent set. The component specification may also include assumptions that the component makes about its environment. A *behavior* of a component is represented by all executions that start in an allowed initial state, that contain only allowed transitions and that satisfy the fairness conditions.

The basic idea of composition is that the composed components start in a common state that is acceptable to all of them, that they take turns performing transitions that are allowed by all of them, and that the fairness conditions of all of the components are satisfied. We allow components to have overlapping agents; however, if the agent sets are disjoint, then the environment transition constraints for each component are checked automatically against the transitions performed by its peers. This automatic check is important from the standpoint of reuse. It removes the need to modify the specification of a component whenever it is to be composed with a new component. It also allows the definition of a component to focus entirely on its own state. However, it also means that there is no way for these constraints to be violated by a peer in a composite system. Thus we stress that the environment transition constraints are an integral part of a component specification and that the component is not described faithfully without them.

Once the composite system is specified, we may want to prove that it satisfies some critical property. The behavior of the composite system is defined as the intersection of the behaviors of its components. It is a trivial consequence of this definition that if some component of the system satisfies the property, then so does every system containing the component. This supports reuse of analysis and the decomposition of a satisfaction proof into smaller, component-local satisfaction proofs.

Frequently, a component is designed so that it satisfies some desired property as long as certain assumptions hold true for its environment (e.g., the processes with which it communicates follow an agreed upon protocol). In this case, we say the component *conditionally satisfies* the property. To show that it satisfies the property unconditionally, we must assert that the environment validates its assumptions. There are two approaches. The first approach is to add the assumptions to the component specification (as noted earlier) and then compose the component with an environment that satisfies those assumptions. The second approach is to state the property as an implication, which makes the assumptions more explicit in the satisfaction proof.

A (possibly composite) component is a *refinement* of, or *implements*, another component if the behaviors of the first component are a subset of the behaviors of the second component. This is a useful concept since property satisfaction is preserved under refinement. Thus properties can be proven at an abstract level of specification and a refinement analysis can then be performed to show the properties remain true for a lower level specification. Since component behaviors are typically infinite sets of execution histories, which are themselves infinite sequences, an arbitrary refinement proof

can be difficult. Therefore, we defined a theorem (like [1]) that states that for any two components a and b, a implements b if the initial states of a are contained in the initial states for b, the transitions allowed by a are contained in the transitions allowed by b and the behavior of a satisfies the fairness constraints for b, subject to some refinement mapping.

When a and b are both composites, the proof that a implements b can be decomposed into a collection of smaller proofs showing that for each component c of composite b, there is some set d of the components comprising a such that the composition of d implements c. The subproofs will usually be easier than the large one. Furthermore, since each subproof focuses on a single component c composed into b, they are likely to be reusable in another refinement analysis where c is implemented by the same low-level components but is composed with a different set of components at the high level.

3. Lessons Learned

We applied the framework to several small examples in order to test various features. Our first effort was to specify an abstract box manager that defines the requirements for message passing in a modular operating system architecture, then we proposed an implementation consisting of kernels, network servers and a network. Later we refined the network server into a network protocol stack. In a parallel effort, we specified a prototype fault-tolerant architecture, then, using the lessons from the prototype, we modified the box manager implementation to be fault tolerant. Finally, we investigated the exposition of information flow policies in the framework. Since space constraints prevent us from describing the framework or its applications further here, we direct the reader to the program web page[3] for more information. The lessons described below are derived from these applications.

Common state. In the DTOS incarnation of the framework, we defined state and agent translators to relate components and their states during composition. Each component was defined on a separate state type, then a global state was defined that contained all component states. The global state included translator functions to map each component state to the global state. In addition to the work of defining the translators, significant proof effort was expended to apply the translators. As the translators were trivial projections, the extra effort yielded little insight. We realized that if we ever used non-trivial translators, the translators themselves might obscure what was really being proven.

[3] *http://www.securecomputing.com/css*

This cumbersome approach was abandoned during the box manager specification in favor of a universal, common state. Component states are defined as before, except that they import a *config* theory containing shared data types and global "configuration" information. The component state definitions are combined in a single *common state* with a separate field for each component state. Each component is defined directly on the common state but all of its accessors are to its local state, so the component really depends only on its local state. As components are added or removed, the changes to global state are isolated to the *config* and *common state* theories, so other components and their analyses are not affected.

Refinement. The box manager application was also the first to use the framework to specify a system for refinement. We realized many advantages by specifying more than one level of abstraction. The higher level contained just enough detail to prove the critical property, which in turn made the property easier to prove. All other necessary design detail was introduced in the refinement. We believe that critical property proofs will usually be harder than refinement proofs, so this approach reduces the overall proof effort required.

However, we discovered that planning for refinement is tricky. Special care is needed to ensure that the refinement is valid. The analyst must avoid making tacit assumptions at the lower level that are derived from the higher level. For example, if one component is split into many, the analyst must specify the allowed interactions between the low level components so that they cannot do things to each other that the high level component cannot do to itself.

The analyst should also fight the urge to overspecify the highest level. Overspecification introduces new constraints, which limits the opportunities for refinement. We also learned that the "simplest" specification can introduce undesirable constraints. Consider an abstract component c that has two variables, i for input and o for output. At any time c can copy i to o, but it makes no other changes to o. Now refine c to a chain $c_1, \ldots, c_n$ of components. The variable i corresponds to the input of c_1, and the variable o corresponds to the output of c_n. Each c_i performs a copy operation. Unfortunately, this refinement is not valid. In c, if o changes, it must change to the value in i. However, in the implementation of c, it may change to the input value for c_n and this value may not match the input value for c_1. The problem here is the specification of c is too constrained. It asserts implicitly that c can process only one value at a time. It prevents refinements where multiple values are in transit. We expect that this is a general problem for refinements.

Reuse. Just as it is difficult to plan for refinement, it is difficult to anticipate reuse. Both the box manager and

its implementation were specified with a nondeterministic message delivery mechanism. There was no constraint on message order because none was required to prove the critical property. However, when we attempted to modify the box manager implementation for fault tolerance, we discovered that our fault tolerance model explicitly assumes first-in-first-out (FIFO) communication [5], so the unmodified box manager implementation failed to meet the basic assumptions for fault tolerance! The obvious solution was to modify the implementation to exhibit FIFO behavior.

Replication. A system is *fault tolerant* if it behaves like a fault free system. Our specification for the fault tolerance application includes two levels of abstraction: the high level is the fault free system, and the low level is the fault tolerant implementation. We decided that it is sufficient to demonstrate that the fault tolerant system is a refinement of the fault free system. Refinement in the other direction is better demonstrated through other methods, such as testing, because a refinement mapping may not exist.

The most unique characteristic of a fault tolerant system is component replication. We discovered that component replication complicates the search for a refinement mapping. A fault tolerant architecture contains multiple, redundant data paths, some of which may be faulty. The outputs from these paths are combined through a majority vote (for Byzantine systems at least) into a single response, which must be the same response that a fault free system would give. The refinement mapping must choose the non-faulty path in the fault tolerant model that contains the fewest messages and define the data path in the fault free model in terms of that path. If there is more than one non-faulty path, the choice might change in any state transition. The refinement mapping must account for this situation. Note that the refinement mapping depends upon knowledge of which paths are faulty. The information is needed in the model anyway to express the hypothesis that a sufficient number of components are non-faulty; however, it is "metastate" information that is not accessible to any implementation.

Separation of concerns. The fault tolerance application gave us the opportunity to highlight separation of concerns as a major benefit of composition. For the fault tolerant prototype, we specified the unique functions of a fault tolerant architecture, i.e., data replication and voting, as separate components rather than introducing them into existing components. We also needed to model faulty behavior, so we specified a fault generator component and composed it into each data path. This approach allowed us to reuse these components with little change when we modified the box manager implementation to be fault tolerant.

Fairness. An interesting realization was that fairness conditions in a high-level specification can never be implemented entirely. There must be some fairness condition at each refinement level. For example, even though a specific scheduling algorithm might be added to lower levels to define how the high-level actions governed by fairness conditions are scheduled, it will still be necessary to include a fairness condition at the lower level requiring that the scheduler itself be treated fairly.

4. Summary and conclusions

We have a described a mathematical framework for performing composition and refinement analysis, and we have discussed lessons from its use. We continue to apply the framework on other projects at SCC and to discover new and better ways to use it. In particular, requirements-based specifications, as opposed to design-based specifications, have shown great promise, especially at the higher levels of abstraction. We anticipate that many evolutionary improvements could be made to the framework itself and to its supporting tools, particularly the analyst's assistant.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. Technical Report 118, Digital Equipment Corporation, Systems Research Center, Dec. 1993.

[2] T. Fine. A Framework for Composition. In *Proceedings of the Eleventh Annual Conference on Computer Assurance* , pages 199–212, Gaithersburg, Maryland, June 1996.

[3] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

[4] S. Owre, N. Shankar, and J. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA 94025.

[5] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[6] N. Shankar. A lazy approach to compositional verification. Technical Report TSL-93-08, SRI International, Dec. 1993.