Appears in the 16th International Conference on Formal Methods in Computer-Aided Design (FMCAD)
Mountain View, California, USA, October 3-6, 2016

Verifiable Hierarchical Protocols with Network Invariants on Parametric Systems

Opeoluwa Matthews
Department of ECE
Duke University
luwa.matthews@duke.edu

Jesse Bingham
Intel Corporation
jesse.d.bingham@intel.com

Daniel J. Sorin
Department of ECE
Duke University
sorin@ee.duke.edu

***Abstract*—We present Neo, a framework for designing pre-verified protocol components that can be instantiated and connected in an arbitrarily large hierarchy (tree), with a guarantee that the whole system satisfies a given safety property. We employ the idea of *network invariants* to handle correctness for arbitrary depths in the hierarchy. Orthogonally, we leverage a parameterized model checker (*Cubicle*) to allow for a parametric number of children at each internal node of the tree. We believe this is the first time these two distinct dimensions of configuration have been together tackled in a verification approach, and also the first time a proof of an *observational preorder* (as required by network invariants) has been formulated inside a parametric model checker. Aside from the natural up/down communication between a child and a parent, we allow for peer-to-peer communication, since many real protocol optimizations rely on this paradigm. The paper details the Neo theory, which is built upon the *Input-Output Automata* formalism, and demonstrates the approach on an example hierarchical cache coherence protocol.**

I. Introduction

Formal verification of large-scale, modern systems protocols is currently challenging. Although theorem proving is theoretically able to verify arbitrary protocols, the manual effort required to guide a theorem prover through the verification of a modern protocol is prohibitive. Model checkers are more widely used, but they cannot handle complex, large-scale protocols. As a result of the state explosion problem, model checking proofs are successful for only a handful of protocol components—generally not sufficient to exercise all the behaviors exhibited in industrial-scale systems. Hence, there is strong motivation for architects to design protocols specifically to be verifiable with state-of-the-art model checking tools. Our solution is to construct a set of protocol components, instances of which are composed into an arbitrary hierarchy, where each component instance is independently scaled. The components are pre-verified in such a way as to guarantee that the resulting large and complex system is always correct.

Our approach involves the combination of two distinct ideas from the model checking literature: *network invariants* and *parameterized model checking*. Consider the hierarchical protocol depicted in Fig. I. We would like to design the leaf (L), internal (I), and root (R) nodes[1] so that any arbitrary nesting in the vertical direction, and any arbitrary (and independent) branching degree (number of children) at each internal and root node, yields a system that is correct. Arbitrary nesting is handled by network invariants; in particular we require (and verify) that L is a network invariant. This means that the observational behaviors of L subsumes that of any larger composition of components. For instance, the behavior along communication channel $c2$ over-approximates that of $c1$, $c3$, etc. We formulate network invariants in a novel way that not only captures the observational behaviors (messages) across an interface, but also captures what we call the *summary state* of a sub-hierarchy. These summary states are integral in defining the safety property, which, like the system itself, is hierarchically defined.

Beyond the hierarchical nesting afforded by network invariants, we employ parameterized model checking to allow arbitrary branching degrees. This entails that we prove the observational pre-order containment required of network invariants *parametrically* in a model checker; we believe this is novel. Hence, L serves as a network invariant for not just a particular I, but for all members of an infinite family $I(1), I(2), I(3), \ldots$, where $I(n)$ is an internal node configured to connect to n children.

We emphasize that network invariants and parameterized model checking are both necessary ingredients in this story; neither is capable of solving what the other does. Network invariants deal with the connection of instances of components into arbitrarily complex hierarchies, with relatively simple interfaces between constituents; parameterized model checkers typically do not support such a notion of "parameterization" when the structure is nontrivial (e.g. a tree). On the other hand, network invariants are not appropriate to deal with an internal node that is parameterized on the number of children. An example is a directory in a cache coherence protocol—the directory is an array, with one entry per child. There is no clear way to formulate this type of tightly coupled parameterization as the composition of components along with a network invariant. Fortunately, parameterized model checkers are usually targeted at precisely this style of parameterization.

Previous research on network invariants [12], [15], [1], [16], [32] tends to focus on "flat" compositions of processes with rather trivial structure; processes are arranged in a linear or circular array with only neighbor-to-neighbor communication, or the other extreme wherein each process talks to all others. The work of Clarke, Grumberg, and Jha [7], is the most closely

[1] We use the terms *component* and *node* interchangeably.

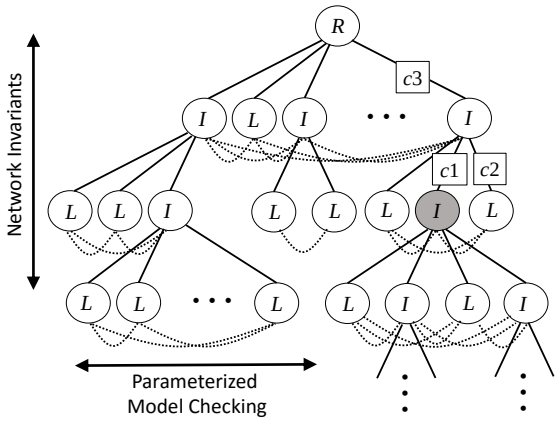

Fig. 1. A Neo hierarchy. Nodes labels R, I, and L respectively indicate root, internal and leaf nodes. Solid lines indicate parent/child communication channels, while dotted lines indicate peer/peer communication. Arbitrary nesting of tree structures in the vertical direction is handled by network invariants. Arbitrary branching widths in the horizontal direction is handled by parameterized model checking. The leaf L acts as a network invariant, which means for example that the behavior along communication channel $c2$ over-approximates that of $c1$, $c3$, etc.

related to us, since they allow hierarchical structures. Like us, they require that a small process serves as a network invariant for all (larger) composite processes.[2]. However, we extend their work in several ways:

- As mentioned above, we use parameterized model checking to facilitate arbitrary branching degree
- We use an asynchronous/interleaved execution semantics (I/O automata), while Clarke et al. use a synchronous.
- We make a (modest, but important) extension to allow processes to be given "identifiers."
- Our example cache protocol is significantly more complex than their example (a protocol that computes a parity function over the leaves of a tree).
- We express our state invariant property using summary functions, which makes the property's structure naturally echo that of the Neo system's hierarchy.

Other related work looks at the problem of verifying hierarchical protocols with two levels, using abstraction and assume/guarantee reasoning [5]. Similar to us, smaller systems are verified to conclude coherence of a system for which model checking is intractable, but the approach involves manual effort and it's unclear if it scales to more elaborate hierarchies.

Parameterized model checking approaches have been widely explored in the literature [13], [2], [9]. The research includes disparate techniques such as: assume/guarantee-style abstraction [21], [22], [6], [14], [33], predicate abstraction [17], invisible invariants [26], flows [23], regular sets [4], Satisfiability Modulo Theories (SMT) [11]. We elected to use *Cubicle* [8] as it has a clean language, has published encouraging results, and is being actively maintained. Though our work is rather agnostic to the underlying model checking technique, we believe our leveraging of a parametric model checker to parametrically prove an observational pre-order is novel.

Some prior work has proposed designing systems from pre-verified components to enable scalable verification. Zhang et al. propose designing cache coherence protocols such that caches are organized in a tree hierarchy, with any scale of the system being observationally equivalent to a pre-verified small-scale system [34]. Unfortunately, [34] is not rigorously formalized. Furthermore, the definition of observational equivalence used focuses only on matching states and ignores actions, which could permit safety violations in a larger scale system. [31] and [30] present performance optimizations to [34] and [20] adapts [34] to designing verifiable power management protocols. Hence, these works inherit [34]'s flaws.

Beu et al. propose a template that allows one to link pre-verified cache coherence protocols into a hierarchy by allowing directories of lower tiers to seek permissions from higher tiers [3]. However, the work is also not rigorously formalized. Also, the pre-verified protocols are not verified in an environment where they interact with higher tiers, which could permit incoherence when they are actually linked into a hierarchy.

To illustrate our verification methodology, we design a hierarchical cache coherence protocol called NeoGerman by composing a parameterized German protocol [6] into a Neo hierarchy. We prove that our protocol is a Neo system, which implies that it behaves correctly for any arbitrary configuration of the hierarchy. While the flat German protocol is trivial, we are not aware of any work that verifies an arbitrary-dimension hierarchical version. We believe our framework is applicable to more sophisticated protocols, and we pick the hierarchical German protocol only to illustrate our approach.

We note that several proofs have been omitted due to length constraints; these proofs can be found in the complete version of this paper [19].

II. Formalizing the Neo Framework

Our framework can be thought of as a class of transition systems for which certain properties hold, as a result of which any member of this class is amenable to a much simpler verification methodology. We hope many systems protocols can be shown (or designed) to fit this class and thus inherit the simplified verification. In this section, we will define this class of transition systems and prove that given some automatedly verifiable antecedents, all members of this class are *safe*.

For any $n \geq 0$, we define $\mathbb{N}_n = \{0, 1, \ldots, n-1\}$; note that $\mathbb{N}_0 = \emptyset$. Also, if $x = (x_0, \ldots, x_k)$ is a tuple or list, we denote x_i by $x[i]$.

A. I/O Automata Theory

We start by giving a short description of the well-known I/O automata theory upon which our framework is formalized. We will only go into enough detail as is sufficient for our work; for a more complete description of I/O automata, see [29].

[2]In cases where a single terminal (what we call a *leaf*) process fails to be a networks invariant, they are able to instead employ a non-terminal, but suitably small, composition of processes.

An action signature S is a partition of a set $acts(S)$ of *actions* into three disjoint sets: $in(S)$, $out(S)$, and $int(S)$, respectively called the *input*, *output*, and *internal* actions. The set $int(S) \cup out(S)$ is denoted by $local(S)$. An *I/O automaton* (IOA) A consists of the following:[3]

- an action signature S, denoted $sig(A)$
- a set $states(A)$ called the *states*
- a nonempty set $start(A) \subseteq states(A)$ called the *start states*
- a transition relation $steps(A) \subseteq states(A) \times acts(S) \times states(A)$

$acts(S)$ is also referred to as $acts(A)$, $in(S)$ is also referred to as $in(A)$, etc. The set $in(A) \cup out(A)$ of *external actions* is referred to as $ext(A)$.

An *execution fragment* e of A is a sequence $e = s_0, a_1, s_1, \ldots, a_k, s_k$ such that, for each i, $(s_i, a_{i+1}, s_{i+1}) \in steps(A)$. If $s_0 \in start(A)$, then e is an *execution* of A. The set of executions of A is denoted by $execs(A)$. If a state s is the final state of an *execution*, then s is said to be *reachable*.

A set $\{S_0, \ldots, S_{n-1}\}$ of action signatures is said to be *compatible* if for all $i \neq j$, $out(S_i) \cap out(S_j) = \emptyset$ and $int(S_i) \cap acts(S_j) = \emptyset$. A set of IOA are said to be compatible if their action signatures are compatible.

The n-way composition $S = \prod_{i=0}^{n-1} S_i$ of compatible action signatures $\{S_0, \ldots, S_{n-1}\}$ is an action signature with $in(S) = \bigcup_{i=0}^{n-1} in(S_i) \setminus \bigcup_{i=0}^{n-1} out(S_i)$, $out(S) = \bigcup_{i=0}^{n-1} out(S_i) \setminus \bigcup_{i=0}^{n-1} in(S_i)$, and $int(S) = \bigcup_{i=0}^{n-1} int(S_i) \cup (\bigcup_{i=0}^{n-1} out(S_i) \cap \bigcup_{i=0}^{n-1} in(S_i))$ [4].

The n-way composition $C = \prod_{i=0}^{n-1} C_i$ of compatible IOA $\{C_0, \ldots, C_{n-1}\}$ is an IOA with the following:

- $sig(C) = \prod_{i=0}^{n-1} sig(C_i)$
- $states(C) = states(C_0) \times \cdots \times states(C_{n-1})$
- $start(C) = start(C_0) \times \cdots \times start(C_{n-1})$
- $steps(C)$ is a set of tuples of the form $(s, a, s') \in states(C) \times acts(C) \times states(C)$ that satisfy the following for all i:
  - $a \in acts(C_i)$ implies $(s[i], a, s'[i]) \in steps(C_i)$
  - $a \notin acts(C_i)$ implies $s[i] = s'[i]$

For IOA $C = \prod_{i=0}^{n-1} C_i$, for $s \in states(C)$ and for all i, define $s|C_i = s[i]$. Let $e = s_0, a_1, s_1, \ldots, a_k, s_k$ be an execution of C. Then, for all i, define $e|C_i$ as the sequence derived by modifying e as follows. Delete each a_j, s_j if $a_j \notin acts(C_i)$. Then, replace all remaining s_j with $s_j|C_i$.

Lemma 1. *Let IOA $C = \prod_{i=0}^{n-1} C_i$ and $e \in execs(C)$. Then, for all i, $e|C_i \in execs(C_i)$.*

Proof. See Tuttle et al. [29]. □

B. Defining Neo Systems

We now formalize our framework by defining a class of IOA and expressing what properties we require of processes in the class. We will eventually show that these properties can be verified automatedly and indeed imply safety of the system.

For any set of actions Σ, we define $\Sigma(n) = \Sigma \times \mathbb{N}_n$ and $\Sigma(n, m) = \Sigma \times \mathbb{N}_n \times \mathbb{N}_m$. Let U be a finite set of *upward interface actions*, let D be a finite set of *downward interface actions*, and let P be a finite set of *peer-to-peer interface actions*. We identify some classes of IOA below.

- An IOA I is an (n, m) (or n-child and m-peer) *internal node* if it supports communication with a parent, n children, and $m-1$ peers. Formally, $out(I) = U \cup D(n) \cup P(m-1)$ and $in(I) = D \cup U(n) \cup P(m-1)$.
- An IOA L is a *leaf node* if it is a $(0, m)$ internal node, for some m. Hence, $out(L) = U \cup P(m-1)$ and $in(L) = D \cup P(m-1)$. A leaf is a degenerate internal node with no children.
- An IOA R is an n-child *root node* if $out(R) = D(n)$ and $in(R) = U(n)$. An n-child root node caps a Neo hierarchy, and hence has no peer or parental communication, but still has n children.

For example, the node shaded grey in Fig. I is a $(4, 3)$-internal node. For each $i \in \mathbb{N}_{m-1}$ and process A, we define the function ϕ_i that derives a new process $\phi_i(A)$ with *tag* i by modifying A's action signature as follows. Let $shift(i, j) = j$ if $j < i$, otherwise $j + 1$.

- If a is an external action with $a \in U \cup D$, or a is an internal action, it is replaced with (a, i).
- Each input action $(p, j) \in P(m-1)$ is replaced with $(p, shift(i, j), i)$.
- Each output action $(p, j) \in P(m-1)$ is replaced with $(p, i, shift(i, j))$.

Intuitively, each upward or downward interface action is now augmented with $\phi_i(A)$'s tag i, allowing it to communicate uniquely with its parent. Each internal action is also augmented with $\phi_i(A)$'s tag i, so as to ensure disjoint sets of internal actions in compositions, as required by IOA theory. To facilitate unique peer-to-peer communication, each peer-to-peer interface action p in $\phi_i(A)$ appears in the form (p, src, dst), where src is the tag of the source process and dst is the tag of the destination process.

Given a set of leaves $Ls = \{L(1), L(2), \ldots\}$, where each $L(m)$ is an m-peer leaf, a set of internal nodes Is, and a set of root nodes Rs, we define the notions of *open Neo systems* and *closed Neo systems* inductively as follows.

- Each $L(m)$ is an m-peer open Neo system, supporting communication with $m-1$ peers.
- Given n n-peer open Neo systems $\Omega_0, \ldots, \Omega_{n-1}$ and an n-child node $A \in Is \cup Rs$, the $(n+1)$-way IOA composition

$$\Omega = A \cdot \prod_{i=0}^{n-1} \phi_i(\Omega_i) \tag{1}$$

  is an m-peer open Neo system (if $A \in Is$) or a closed Neo system (if $A \in Rs$).

We will simply write *Neo system* if we are not concerned about whether Ω is open or closed. Where Ω is an open Neo system,

[3] We deviate from Tuttle's thesis [29] in two ways: we preclude $part(A)$, and we don't require input-enabledness. This is justified since both notions are only relevant for fair executions, which for us is purely future work.

[4] Unlike in Tuttle's thesis [29], by default, we hide messages sent between component processes as internal.

we characterize IOA $\phi_i(\Omega)$, for some $i < m$, as a *tagged* open Neo system and Ω as an *untagged* open Neo system.

Lemma 2. *For Neo system Ω (1), if Ω is an open Neo system, then $in(\Omega) = D \cup P(m-1)$ and $out(\Omega) = U \cup P(m-1)$. If Ω is a closed Neo system, then $ext(\Omega) = \emptyset$.*

C. Neo System Safety

1) Summary of States: Let Sum be a finite set of *summary states* that contains a distinguished state bad. We associate *summary functions* with each $L(m)$ and the elements of Is and Rs as follows

- $sum_{L(m)}$ has type $states(L(m)) \rightarrow Sum$
- For each n-child $A \in Is \cup Rs$, $sum_A : states(A) \times Sum^n \rightarrow Sum$ is a "bad preserving" function, i.e. $bad \in \{s_0, \ldots, s_{n-1}\}$ implies $sum_A(s, s_0, \ldots, s_{n-1}) = bad$.

We extend the above elemental sum_* functions to summarize the state of an arbitrary non-leaf Neo system Ω as follows. Ω is (1), where $A \in Is \cup Rs$ and $\Omega_0, \ldots \Omega_{n-1}$ are open Neo systems. Then $sum_\Omega : states(\Omega) \rightarrow Sum$ is defined by

$$sum_\Omega(s_a, s_0, \ldots, s_{n-1}) = sum_A(s_a, sum_{\Omega_0}(s_0), \ldots, sum_{\Omega_{n-1}}(s_{n-1}))$$

2) Summary Sequence of Executions: Given an execution $e = s_0, \alpha_1, \ldots, \alpha_k, s_k$ of a Neo system Ω, we define the summary sequence $sum(e)$ as follows. Let $\alpha'_i = \alpha_i$ if $\alpha_i \in ext(\Omega)$, otherwise $\alpha'_i = \lambda$. We start with the sequence

$$sum_\Omega(s_0), \alpha'_1, \ldots, \alpha'_k, sum_\Omega(s_k)$$

and delete all $\alpha'_i, sum_\Omega(s_i)$ such that $\alpha'_i = \lambda$ and $sum_\Omega(s_i) = sum_\Omega(s_{i-1})$.

3) Safety Definition: For Neo system Ω and state $s \in states(\Omega)$, we say that s is *safe* if $sum_\Omega(s) \neq bad$. We say that Ω itself is safe if all its reachable states are safe. The primary goal of this paper is to establish that *all* Neo systems are safe, by only proving a handful of lemmas about the "ingredient" IOAs (Ls, Is, Rs) and their summary functions.

D. Neo Pre-order $\preceq$

We define a pre-order $\preceq$ on open Neo systems. Given two m-peer open Neo systems Ω_1 and Ω_2 that are either both tagged or untagged, the relation $\Omega_1 \preceq \Omega_2$ holds if, for all executions e_1 of Ω_1, there exists an execution e_2 of Ω_2 such that $sum(e_1) = sum(e_2)$.

Lemma 3. *$\preceq$ is transitive.*

Lemma 4. *$\Theta \preceq \Omega$ if and only if $\phi_i(\Theta) \preceq \phi_i(\Omega)$.*

Lemma 5. *Let Neo systems $\Omega = A \cdot \prod_{i=0}^{n-1} \phi_i(\Omega_i)$ and $\Theta = A \cdot \prod_{i=0}^{n-1} \phi_i(\Theta_i)$, where $A \in Is \cup Rs$. Suppose for some k, $\Omega_k \preceq \Theta_k$ and for all $i \neq k$, $\Omega_i = \Theta_i$. Then, for all executions e of Ω, there exists an execution e' of Θ such that $sum(e) = sum(e')$.*

Lemma 6. *Let $\Theta = A \cdot \prod_{i=0}^{n-1} \phi_i(\Theta_i)$ and $\Omega = A \cdot \prod_{i=0}^{n-1} \phi_i(\Omega_i)$ be open Neo systems such that $\Theta_i \preceq \Omega_i$, for all i. Then, $\Theta \preceq \Omega$.*

Lemma 7. (Leaf as a Network Invariant) *Suppose that the m-peer open Neo system $\Omega_L = A \cdot \prod_{i=0}^{n-1} \phi_i(L(n))$ satisifies $\Omega_L \preceq L(m)$. Then, for any m-peer open Neo system Ω, $\Omega \preceq L(m)$.*

Proof. If Ω is an m-peer leaf, then $\Omega = L(m) \preceq L(m)$. Otherwise, let $\Omega = A \cdot \prod_{i=0}^{n-1} \phi_i(\Omega_i)$ be an m-peer open Neo system. Assuming that each $\Omega_i \preceq L(n)$ (inductive hypothesis), we will prove, by structural induction on the construction of Ω, that $\Omega \preceq L(m)$. By Lemma 6 and inductive hypothesis, $\Omega \preceq \Omega_L$. By transitivity of $\preceq$ (Lemma 3) and $\Omega_L \preceq L(m)$ (assumption in lemma statement), $\Omega \preceq L(m)$. □

Theorem 1. (Every Neo system is safe.) *Suppose that for each n-child node $A \in Rs \cup Is$, $\Omega_L = A \cdot \prod_{i=0}^{n-1} \phi_i(L(n))$ is safe. Furthermore, suppose that if A is an m-peer internal node, then $\Omega_L \preceq L(m)$. Then all Neo systems are safe.*

Proof. Let Ω be an (open or closed) Neo system (1). From the assumptions of this lemma and Lemma 7, $\Omega_i \preceq L(n)$ for all i. Let e be an arbitrary execution of Ω. By an n-fold application of Lemma 5, there exists an execution e' of Ω_L such that $sum(e') = sum(e)$. By definition of sum, if no state in e' summarizes to bad, then no state in e summarizes to bad. Therefore Ω is safe. □

The significance of Theorem 1 is that if we establish Ω_L's safety, for all $A \in Rs \cup Is$, and $\Omega_L \preceq L(m)$, for all $A \in Is$, then any configuration of the Neo nodes (which would typically be closed) is safe. Parameteric model checking comes into play, since when the elements of $Rs \cup Is$ are paramterized (by number of children n and number of peers m), these safety and preorder checks are parameterized verification problems.

III. Mapping Protocols' Safety to Neo Safety

We have defined safety of a Neo system (Sect. IV-D1) to mean that no reachable state summarizes to bad, which is somewhat removed from the actual invariant one might be interested in. Here, we illustrate how an invariant of interest can be expressed in the form of the Neo safety definition. The key is that the summary functions must be forced to return *bad* whenever the specific safety property of interest is violated.

A. Cache Coherence

In a typical MOESI cache coherence protocol [27], $Sum = \{I, S, O, E, M, bad\}$, [5] and cache coherence means that if any leaf summarizes to M or E, then all other leaves must summarize to I. To ensure that Neo system safety (no reachable state summarizes to bad) implies all reachable states are cache coherent, we require some simple constraints on sum_A for each $A \in Is \cup Rs$. Let us define an ordering $<$ on Sum by $I < S, O < E, M < bad$. Recalling that sum_A has type $states(A) \times Sum^n \rightarrow Sum$, where n is the arity of A, the *cache coherence constraint* on sum_A is as follows[6]:

[5] The reasoning of this section can be extended to handle cases where Sum includes more than just these 6 elements.

[6] Note that the constraints presented here are the weakest set of constraints that allow us to prove Lemma 8 below; they do not preclude, e.g., that inconsistencies between the state s_a of A and $s_0, \ldots, s_{n-1}$ yield bad.

- Whenever there exists distinct $i, j \in \mathbb{N}_n$ such that $s_i \in \{M, E\}$ and $s_j \neq I$, we require $sum_A(s_a, s_0, \ldots, s_{n-1}) = bad$, and
- For all $i \in \mathbb{N}_n$, $s_i \leq sum_A(s_a, s_0, \ldots, s_{n-1})$ (i.e. sum_A is monotonically increasing with $<$)

Lemma 8. *If sum_A satisfies the cache coherence constraint for all $A \in Is \cup Rs$, then Neo safety implies cache coherence.*

Proof. Let s be a state of a Neo system Ω. We argue, by structural induction on Ω, that whenever s contains a cache coherency violation, $sum_\Omega(s) = bad$. The base case $\Omega \in Ls$ holds vacuously, since a leaf in isolation cannot violate cache coherency. Now choose $A \in Is \cup Rs$ with arity n. Then $s = (s_a, s_0, \ldots, s_{n-1})$, where $s_a \in states(A)$ and $s_i \in states(\Omega_i)$, $0 \leq i < n$. If s contains a cache coherency violation then there exists a leaf L in s that summarizes to M or E, and a distinct leaf L' that summarizes to something other than I. If L and L' are both components of Ω_i for some i, then from our inductive hypothesis, $sum_{\Omega_i}(s_i) = bad$, and from Sect. II-C1, we have that $sum_\Omega(s) = bad$. On the other hand, suppose L and L' are respectively components in Ω_i and Ω_j with $i \neq j$. Since the cache coherency constraint requires sum to be monotonic, it follows that $E \leq sum_{\Omega_i}(s_i)$ and $S \leq sum_{\Omega_j}(s_j)$, and again the cache coherency constraint requires $sum_\Omega(s) = bad$. □

We envision that other types of Neo systems will need similar side arguments to relate Neo safety to a more concrete property of interest, and such arguments will be as straightforward as what was required to prove Lemma 8 above.

B. Distributed Lock Management (DLM)

Distributed Lock Management (DLM) protocols are used to ensure safe access to shared resources such as disks and files. Several DLM protocols are based on the DEC VMS's DLM implementation [28], including the Oracle Cluster File System (OCFS2) that appears in the Linux Kernel [24] [18]. VMS's DLM has 6 permissions—Null (NL), Concurrent Read (CR), Concurrent Write (CW), Protected Read (PR), Protected Write (PW), and Exclusive (EX). The following combinations of permissions are prohibited: (CR,EX), (CW,EX), (CW,PW), (CW,PR), (PR,EX), (PR,PW), (PW, EX), (PW,PW), (EX,EX).

For scalable resource management, one can organize nodes in a cluster as a hierarchy according to the Neo framework. This would facilitate verification, for arbitrary system sizes, that no two nodes hold a prohibited combination of permissions. We could set $Sum = \{NL, CR, CW, PR, PW, EX, bad\}$ and define a partial order $<$ such that $NL < CR < PR < PW < EX < bad$ and $NL < CR < CW < EX < bad$; $<$ does not order PW and CW. Then, imposing similar constraints to the sum functions of Sect. III-A, one can show that sum not evaluating to bad implies that the system never violates DLM safety.

IV. Case Study: The NeoGerman Protocol

To illustrate our verification methodology, we design and verify a hierarchical cache coherence protocol called NeoGerman. Using a parametric model checker, we verify that NeoGerman is a Neo System and, consequently, satisfies the coherence invariant for arbitrary configurations.

A. NeoGerman Description

German's protocol is a simple, directory-based caching protocol proposed as a challenge for parameterized verification [10]. To make the protocol hierarchical, we made significant modifications. In particular, the directory was modified to communicate with a parent, hence serving as an internal node.[7]

1) The German Protocol: The German protocol is a flat cache coherence protocol. We use the version specified in [6], which is parameterized to have a single directory connected to an arbitrary number of private caches. Each cache block is in one of three states: I(nvalid), S(hared), or E(xclusive). The protocol uses the directory to maintain the invariant that no two caches are simultaneously in (S, E) or (E, E). The directory maintains a list of all nodes in S or E, called sharers.

If a cache sends a message to the directory to request S ($GetS$) when there is a cache in E, the cache in E gets sent an $Invalidate$ message, and the directory collects an invalidation acknowledgement ($InvAck$) from it. The directory then sends a $GrantS$ message to the requesting cache to grant it S permissions. If the directory receives a $GetE$, it invalidates all sharers and collects all their $InvAck$'s before sending a $GrantE$ message to the requesting cache.

2) Modifications to German: To turn German into an open Neo system, we modify the directory so it behaves like a private cache along a (previously non-existent) communication channel shared with a parent. Upon receiving requests from its children, the directory now has the ability to seek permissions from *its* parent. We will refer to this modified directory as the *internal directory*, to distinguish it from the original German directory that we use as a root node to close the Neo hierarchy.

The internal directory maintains a variable called $Permissions\_O$, which summarizes the permissions of the open Neo system it heads as that of a single private cache. The intent is that if, for example, $Permissions\_O$ is in I and the internal directory receives a $GetS$ from a child, the internal directory forwards the request to its parent. Upon receiving a subsequent $GrantS$ from its parent, $Permissions\_O$ changes to S and the internal directory sends a $GrantS$ to the requesting child and makes it a sharer. If the internal directory receives an $Invalidate$ from its parent, it invalidates all sharing children and collects all $InvAck$s. Finally, the internal directory sends an $InvAck$ to its parent and updates $Permissions\_O$ to I.

B. Tying NeoGerman to the Neo Framework

In NeoGerman, we have $U = \{GetS, GetE, InvAck\}$, $D = \{GrantS, GrantE, Invalidate\}$ and $P = \emptyset$. The private caches correspond to tagged leaf processes of the form $\phi_i(L)$, where L is a leaf node. Each individual leaf is tagged with a parameter that enables unique communication with its

[7] The directory was used unmodified to create the root node.

directory. The directory/memory $R(n)$ of the original German protocol constitutes an n-child root node of NeoGerman. The directory $I(n)$ of the NeoGerman protocol constitutes an n-child internal node. Armed with $R(n)$, $I(n)$, L and the Cubicle process composition methods we discussed above, we have all the ingredients to build NeoGerman as a Neo system.

C. Modeling NeoGerman in a Model Checker

We modeled NeoGerman in Cubicle [8]. Cubicle is a symbolic model checker used to verify parameterized array-based systems by using a backwards reachability algorithm and an SMT solver. Its support for parametric verification allows us to verify safety properties for arbitrary configurations of a Neo hierarchy. Cubicle's processes are parameterized by indices of a built-in type *proc*. The state of an arbitrary number of processes is represented by arrays indexed by *proc*. Even though neither Cubicle nor our framework impose size restrictions on communication buffers, we model NeoGerman with a single-entry communication buffers for simplicity.

1) Representing a Process: To illustrate how we model processes in Cubicle, let set $B = \{\phi_i(A) : i \in \mathbb{N}_n\}$, where A is a Neo leaf node with $steps(\phi_i(A)) = \{(s_0,(a_0,i),s_1),(s_1,(a_1,i),s_2),(s_2,(a_2,i),s_0)\}$. Let $start(A_i) = \{s_0\}$, $in(A_i) = \{(a_0,i)\}$, $out(A_i) = \{(a_1,i)\}$, and $int(A_i) = \{(a_2,i)\}$. We would model B in Cubicle as follows [8]:

```
type state_a = s0|s1|s2 //state type declaration
array State_A[proc]:state //state array variable declaration

init (i)
{State_A[i]=s0//initialize each proc's state to start state}

transition a0(i) //input transition
requires {State_A[i]=s0} //guard
{ State_A[i]:=s1; } //state update

transition a1(i) //output transition
requires {State_A[i]=s1}
{ State_A[i]:=s2; }

transition a2(i) //internal transition
requires {State_A[i]=s2}
{ State_A[i]:=s0; }
```

2) Representing Composition: For IOA B, let $steps(B) = \bigcup_{i=0}^{n-1}\{(S_0,(a_0,i),S_1),(S_1,(a_1,i),S_2),(S_2,\pi,S_0)\}$. Let $start(B) = \{S_0\}$, $out(B) = \{(a_0,i)\}$, $in(B) = \{(a_1,i)\}$, and $int(B) = \{\pi\}$. By combining the guards and state updates of transitions with identical names, we represent the composition $C = B \cdot \prod_{i=0}^{n-1} A_i$ as follows:

```
type state_b = S0|S1|S2; type state_a = s0|s1|s2
var State_B:state_p; array State_A[proc]:state_a

init (i)
{ State_B=S0 & State_A[i]=s0 }

transition a0(i) //internal transition
requires {State_B=S0 & State_A[i]=s0}
{ State_B:=S1; State_A[i]:=s1; }

transition a1(i) //internal transition
requires {State_A[i]=s1 & State_B=S1}
{ State_A[i]:=s2; State_B:=S2; }

transition π() //internal transition
requires {State_B=S2}
{ State_B:=S0; }

transition a2(i) //internal transition
requires {State_A[i]=s2}
{ State_A[i]:=s0; }
```

[8]For all Cubicle code in this paper, we deviate slightly from Cubicle syntax for conciseness.

D. Proving the NeoGerman Hierarchy is Coherent

We leverage the Neo framework and Cubicle's parametric verification to prove that any NeoGerman configuration is coherent. Our strategy is to first define $sum_{A(n)}$ for each $A \in \{R, I\}$ and $n \geq 1$ such that it satisfies the constraints of Section III-A. Then, we prove the conditions of Theorem 1 and Lemma 8, from which coherency of Ω follows. Let $\Omega_{A(n)} = A(n) \cdot \prod_{i=0}^{n-1} \phi_i(L)$, and let $Sum = \{I, S, E, bad\}$, ordered $I < S < E < bad$. To leverage Theorem 1 and Lemma 8, we model $\Omega_{R(n)}$ and $\Omega_{I(n)}$ in Cubicle and prove the following for all n and each $A \in \{R, I\}$

$$\Omega_{A(n)} \text{ is safe} \tag{2}$$

$$\forall i : s_i \leq sum_{A(n)}(s, s_0, \ldots, s_{n-1}) \tag{3}$$

$$\forall i \neq j : s_i \in \{M, E\} \wedge s_j \neq I \Rightarrow sum_{A(n)}(s, s_0, \ldots, s_{n-1}) = bad \tag{4}$$

$$\Omega_{I(n)} \preceq L \tag{5}$$

First, we define $sum_{R(n)}$ and $sum_{I(n)}$. Unlesss the cache coherence constraints (Sec. III-A) require bad, $sum_{R(n)}(s, s_0, \ldots, s_{n-1}) = E$. Likewise, unless the cache coherence constraints require bad, $sum_{I(n)}(s, s_0, \ldots, s_{n-1}) = Permissions\_O$, where $Permissions\_O$ is a Cubicle variable of $I(n)$. Hence, $Permissions\_O$ is a function of $states(I(n))$.

1) Safety and Monotonicity of Sum: [9] To prove (2), we parametrically model check $\Omega_{R(n)}$ and $\Omega_{I(n)}$; after each $\Omega_{I(n)}$ transition, a variable Sum_Output_O representing the output of sum_* is updated to $Permissions\_O$. The following is specified as a safety violation:

```
unsafe(i,j) CacheState[i]=Bad || (CacheState[i]=E &
    CacheState[j]!=I)
```

where, for all i, $CacheState[i] \equiv sum_L(\phi_i(L))$.

For (3), $sum_{R(n)}$ is monotonic by definition. To prove $sum_{I(n)}$ is monotonically increasing, we model check $\Omega_{I(n)}$, specifying the following as safety violations:

```
unsafe(i) {Sum_Output_O!=E & CacheState[i]=E}
unsafe(i) {Sum_Output_O=I & CacheState[i]=S}
```

To prove (4), we model check $\Omega_{I(n)}$ with the following:

[9]As a result of the limitations of Cubicle, we need to *prove* (3) and (4) for reachable states, rather than writing code that clearly satisfies these constraints for any state.

```
unsafe (i j) {not ((CacheState[i]=E &
    CacheState[j]!=Invalid) => Sum_Output_O=Bad)}
```

2) Observational Process Pre-order: To prove (5), we employ a similar approach to Park et al. [25], with an important difference that we generalize to a parametric setting to verify *our* pre-order. Park et al. show how to prove that a process A implements a process B in a model checker by expressing B as a function. A is model-checked and, on each transition t, B's function is called to give B's next state, given A's state at the start of t. An assertion checks that a simulation relation holds, given t's action and the states of A and B at the start and end of t. Cubicle does not support functions and in-line assertions due to its underlying algorithm, so we must rely only on safety properties. As a result of these limitations, we must prove a stricter pre-order $\preceq_c$ based on a slightly different function sum_c defined below.

Let IOA A execution $e = s_0\alpha_1 s_1 \ldots \alpha_k s_k$. Then, $sum_c(e)$ is a sequence derived as follows. Replace each s_i with $sum_A(s_i)$. Replace each $\alpha_i \in int(A)$ with the symbol λ. For IOA A_1 and A_2, $A_1 \preceq_c A_2$ implies for any execution e_1 of P_1, there exists an execution e_2 of A_2 such that $sum_c(e_1) = sum_c(e_e)$.

Lemma 9. *$\Theta \preceq_c \Omega$ implies $\Theta \preceq \Omega$.*

The definition of sum_c implies that, to prove $\Omega_{I(n)} \preceq_c L$, we must match every $\Omega_{I(n)}$ execution with an equal-length execution of L. Hence, we make a trivial modification to the L IOA in NeoGerman by adding λ to $int(L)$ such that, for all $s \in states(L)$, $(s, \lambda, s) \in steps(L)$. This allows L to make as many stuttering steps as needed to match execution fragments of $\Omega_{I(n)}$ that have only internal steps with no change in summary state.

The key to our approach in proving the pre-order is that in the same Cubicle file, we model both $\Omega_{I(n)}$ and L [10] and instrument the code of both processes such that they transition in lockstep, starting with $\Omega_{I(n)}$. Our instrumentation also guides L to pick transitions that match each $\Omega_{I(n)}$ transition. We use a safety property to check that, after each L transition, the states and actions of L and $\Omega_{I(n)}$ correspond as required for sum_c to be equal. We also use a safety property to check that, after each $\Omega_{I(n)}$ transition, there always exists an L transition that *can* fire. If both safety checks pass, then we know that $\Omega_{I(n)} \preceq_c L$ and, thus, $\Omega_{I(n)} \preceq L$ (Lemma 9).

Matching Executions:

1) To force $\Omega_{I(n)}$ and L to transition in lockstep, a variable L_to_run is initialized to false. It is set to true after each $\Omega_{I(n)}$ transition and set to false after each L transition. Then, the expression L_to_run=*False* is conjuncted to the guard of each $\Omega_{I(n)}$ transition and L_to_run=*True* is conjuncted to the guard of each L transition.

2) To access the most recent actions of L and $\Omega_{I(n)}$, we update a variable O_action only at the end of each $\Omega_{I(n)}$ transition and variable L_action only at the end of each L transition. For external transitions, O_action and L_action are updated to the transition's name. Otherwise, they are updated to $lambda$.

3) To guide L to make a matching external step to each external step of $\Omega_{I(n)}$'s, we conjunct to the guard of each L external transition named $trans\_name$ the expression O_action=$trans\_name$.

4) To guide L to make a matching step to each internal step of $\Omega_{I(n)}$'s, a variable $Forced\_Transition$ is updated to some value int_name after each $\Omega_{I(n)}$ internal transition. Then, the guard of the desired L internal transition is conjuncted with the expressions $Forced\_Transition$=int_name and O_action=$lambda$.

Note that the above modifications maintain the integrity of the pre-order check. All modifications to L's guards involve logical conjunctions, which could only restrict L's transitions. And the only modification to $\Omega_{I(n)}$'s guards is conjuncting L_to_run=*False*, which holds after every L transition.

Safety Checks:

We must check that, after each L transition, the actions and summaries of states of L and $\Omega_{I(n)}$ match. Where $sum_L(S_L) \equiv Cache\_State\_L$, the following illustrates our safety checks.

```
unsafe() {Sum_Output_O!=Cache_State_L & L_to_run=False}
unsafe() {O_action!=L_action & L_to_run=False}
```

Finally, we must check that after each $\Omega_{I(n)}$ transition, there exists an L transition that can fire. To do that, we express a safety property that says that if L_to_run=*True*, the conjunction of the guards of all L transitions must not evaluate to *False*. With both safety checks passing, we can conclude that $\Omega_{I(n)} \preceq_c L$, and, consequently, $\Omega_{I(n)} \preceq L$ (Lemma 9).

This completes our proof that NeoGerman is a Neo hierarchy and thus CCsatisfies coherence for any arbitrary configuration. The full NeoGerman Cubicle model and proofs can be viewed at: http://people.duke.edu/~om26/papers/FMCAD16.

V. Characterizing the Scope of our Framework

To characterize the scope of our framework, we define a fragment of first order formulas over leaf states that we can verify using our approach and define summary functions that are guaranteed to verify a given property. Let $LP = \{\ell_1, \ldots, \ell_m\}$ be a set of predicates over the leaf states $states(L)$. We show how we can verify any invariant of the form

$$\forall x_1, \ldots, x_k. Distinct(x_1, \ldots, x_k) \Rightarrow P(x_1, \ldots, x_k) \quad (6)$$

where the x_i's range over leaves, $Distinct(x_1, \ldots, x_k)$ indicates that the x_i's are pairwise not equal, and $P(x_1, \ldots, x_k)$ is a propositional formula over the atoms $\{\ell_j(x_i) | 1 \leq j \leq m \wedge 1 \leq i \leq k\}$. For example, where $LP = \{E, S, I\}$, the cache coherence invariant we verified for NeoGerman could be expressed as $\forall x_1, x_2. Distinct(x_1, x_2) \Rightarrow (E(x_1) \Rightarrow I(x_2))$.

To verify that (6) is invariant, we construct summary functions such that the state of a NEO hierarchy summarizes to bad

[10] Observe that L is identical to each $\phi_i(L)$, except L has no indices in transitions and its state is not a *proc* array.

if and only if (6) is false of the system's state. The summary functions have co-domain Sum, where Sum is the (finite) set

$$Sum \;=\; \left(2^{LP} \to \{0, \ldots, k\}\right) \cup \{bad\}$$

When bad is returned to node A's summary function, this indicates that (6) fails to hold of the sub-hierarchy rooted at A. Otherwise, a function f is returned, with the interpretation that $f(LP')$ is the number of distinct leaves under A with states satisfying exactly the predicates $LP' \subseteq LP$; if there are k or more such leaves, $f(LP') = k$.

The leaf summary function sum_L simply returns the function that maps all sets to 0, except the exact subset of LP that holds of the leaf's state, which is mapped to 1. However, if $k = 1$ and P does not hold of the leaf's state, bad is returned. Where A is an n-child internal or root node, it is relatively straightforward to define how sum_A (which is independent of its first argument $s_A \in states(A)$) depends on arguments $(g_0, \ldots, g_{n-1}) \in Sum^n$ and under what conditions it should return bad. sum_A returns the function that maps each LP' to $g_0(LP') + \cdots + g_{n-1}(LP')$, saturating at k, unless any g_i is bad or the counts of $(g_0, \ldots, g_{n-1})$ for each LP' indicate that some x_i's below A violate $P(x_1, \ldots, x_k)$, in which case bad is returned.

VI. Conclusion

We present the Neo framework that leverages network invariants and parameterized model checking together to enable the design and automated verification of hierarchical (tree) protocols that, for any size or configuration of the hierarchy, satisfy a safety property. We use our framework to design and verify a hierarchical cache coherence protocol called NeoGerman, using Cubicle as our parameteric model checker. Significantly, we prove an observational pre-order in a parametric setting. We believe there are no fundamental limitations that prevent our framework from being used to design and verify more complex, industrial-strength *hierarchical* protocols, especially given that model checkers like Cubicle have already been used to parametrically verify several industrial-strength *flat* protocols.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant CCF-142-1167. We thank John Erickson and Kenneth McMillan for providing helpful advice for this work.

References

[1] P. A. Abdulla and B. Jonsson. On the existence of network invariants for verifying parameterized systems. In *Correct System Design*. Springer, 1999.

[2] P.A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2013.

[3] J. G. Beu, J. A. Poovey, E. R. Hein, and T. M. Conte. High-speed formal verification of heterogeneous coherence hierarchies. In *HPCA*. IEEE, 2013.

[4] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, 2000.

[5] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *HLVDT*. IEEE, 2007.

[6] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*. Springer, 2004.

[7] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *CONCUR*. Springer, 1995.

[8] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems. In *CAV*. Springer, 2012.

[9] Z. Ganjei, A. Rezine, P. Eles, and Z. Peng. Abstracting and counting synchronizing processes. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2015.

[10] S. German. Personal correspondence. 2008.

[11] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by smt solving: Termination and invariant synthesis. 2010.

[12] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action. In *CONCUR*. Springer, 2002.

[13] J. Kloos, R. Majumdar, F. Niksic, and R. Piskac. Incremental, inductive coverability. In *CAV*. Springer, 2013.

[14] S. Krstić. Parameterized system verification with guard strengthening and parameter abstraction. In *Automated Verification of Infinite-State Systems*, 2005.

[15] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *PODC*. ACM, 1989.

[16] M. Kyas. Verifying a network invariant for all configurations of the futurebus+ cache coherence protocol. *Electronic Notes in Theoretical Computer Science*, 2001.

[17] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, 2004.

[18] LWN.net. The ocfs2 filesystem. http://lwn.net/Articles/137278/.

[19] O. Matthews, J. Bingham, and D. J. Sorin. Verifiable hierarchical protocols with network invariants on parametric systems. Extended version of this paper (with proofs), 2016. http://people.duke.edu/~om26/papers/FMCAD16/fmcad16_neo_extended.pdf.

[20] O. Matthews, M. Zhang, and D. J Sorin. Scalably verifiable dynamic power management. In *HPCA*. IEEE, 2014.

[21] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, 1999.

[22] K. L. McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In *CHARME*. Springer, 2001.

[23] J. O'Leary, M. Talupur, and M. R. Tuttle. Protocol verification using flows: An industrial experience. In *FMCAD*, 2009.

[24] oss.oracle.com. General-purpose cluster file system. https://oss.oracle.com/projects/ocfs2/.

[25] S. Park, S. Das, and D. L. Dill. Automatic checking of aggregation abstractions through state enumeration. *Computer-Aided Design of Integrated Circuits and Systems*, 2000.

[26] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, 2001.

[27] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 2011.

[28] D. W. Thiel. The VAX/VMS distributed lock manager. *VAX cluster Systems*, page 29, 1987.

[29] M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Massachusetts Institute of Technology, 1987.

[30] G. Voskuilen and T.N. Vijaykumar. Fractal++: Closing the performance gap between fractal and conventional coherence. In *ISCA*. IEEE, 2014.

[31] G. Voskuilen and T.N. Vijaykumar. High-performance fractal coherence. In *ASPLOS*. ACM, 2014.

[32] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*. Springer, 1990.

[33] M. Zhang, J. D. Bingham, J. Erickson, and D. J Sorin. Pvcoherence: Designing flat coherence protocols for scalable verification. In *HPCA*. IEEE, 2014.

[34] M. Zhang, A. R. Lebeck, and D. J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *MICRO*. IEEE, 2010.