Model Checking in Bits and Pieces

Kedar S. Namjoshi
Bell Labs, Alcatel-Lucent
kedar@research.bell-labs.com

Fully automated verification of concurrent programs is a difficult problem, primarily because of *state explosion*: the exponential growth of a program state space with the number of its concurrently active components. It is natural to apply a divide and conquer strategy to ameliorate state explosion, by analyzing only a single component at a time. We show that this strategy leads to the notion of a "split" invariant, an assertion which is globally inductive, while being structured as the conjunction of a number of local, per-component invariants. This formulation is closely connected to the classical Owicki-Gries method and to Rely-Guarantee reasoning. We show how the division of an invariant into a number of pieces with limited scope makes it possible to apply new, localized forms of symmetry and abstraction to drastically simplify its computation. Split invariance also has interesting connections to parametric verification. A quantified invariant for a parametric system is a split invariant for every instance. We show how it is possible, in some cases, to invert this connection, and to automatically generalize from a split invariant for a small instance of a system to a quantified invariant which holds for the entire family of instances.

1 Introduction

Concurrency was once limited to the internals of operating systems and networking. It is now in the mainstream of programming, largely due to the availability of cheap hardware and the ubiquitous presence of multi-core processors. Designing a correct concurrent program or protocol, however, is a hard problem. Intuitively, this is because the designer must coordinate the behavior of multiple, simultaneously active threads of execution. Verification of an existing program is an even harder problem, as the analysis process must reconstruct the invariants which guided the original design of the program. These informal statements can be made precise through complexity theory: verification of an N-process concurrent program is PSPACE-hard in N, even if the state space of each component is fixed to a small constant. In practice, the difficulty manifests itself as model checking tools run into *state explosion*: the exponential growth of a program state space with the number of its concurrent components.

A common strategy when faced with a large problem is to break it into smaller and simpler subproblems. In program verification, this "divide and conquer" strategy is known as *compositional* or *modular* verification. The essential idea is to verify a program "in bits and pieces", analyzing only a single component at a time, along with an abstraction of the environment of the component (i.e., the rest of the program). The foundations of compositional methods were created by Owicki and Gries [27] and Lamport [21]. These methods are based on a simple proof rule; however, formulating the right combination of assertions for the proof rule can be a difficult and frustrating task. Also, it is not clear that doing so reduces the manual proof effort in any appreciable way, as Lamport points out in [22]. On the other hand, for *fully automated* proof methods such as model checking or static analysis, there is indeed much to be gained by a divide-and-conquer strategy, as the state space of a single component is much smaller than that of the full program.

Submitted to:

In this article, we focus on the simplest, but fundamental verification task: constructing an inductive program invariant. We show how the construction of a *compositional* inductive invariant may be formulated as a simultaneous least fixpoint calculation. That paves the way for a variety of simplifications and generalizations. We show how the fixpoint can be computed in parallel, how the fixpoint computation may be simplified drastically by analyzing the *local symmetries* of a process network, and how it may be generalized through the use of *local abstraction*. As much of this exposition is based on published work, in this article we keep a light touch on the theory, emphasizing instead the intuition which lies behind the theoretical ideas.

To illustrate the many aspects of the theory, we use a running example of a Dining Philosophers' protocol. This is chosen for two reasons: it is particularly amenable to compositional reasoning, and the protocol is flexible enough to operate on arbitrary networks, making it easy to illustrate the effects of localized symmetry and local abstraction, and their influence on parametric reasoning.

2 Split Invariance

Methods for program verification are based on two fundamental concepts: that of *inductive invariance* and *ranking*. An *inductively invariant* set is closed under program transitions and includes all reachable program states. A *ranking* function which decreases for every non-goal state shows that the program always progresses towards a goal. The strongest (smallest) inductive invariant set is the set of reachable states. The standard model checking strategy – without abstraction – is to compute the set of reachable states in order to show that a property is invariant (i.e., it includes all reachable states). The reachability calculation can be prohibitively expensive due to state explosion: for instance, the model-checker SPIN [19] runs out of space checking the exclusion property for approximately 10 Dining Philosophers on a ring. The divide-and-conquer approach to invariance, which we discuss in this paper, is to calculate an inductive invariant which is made up of a number of local invariant pieces, one per process. A rather straightforward implementation of this calculation verifies the exclusion property for 3000 philosophers in about 1 second. In this section, we develop the basic theory behind the compositional reasoning approach. Subsequent sections explore connections to symmetry, abstraction, and parametric verification, as well as some of the limitations of compositional reasoning.

A Note on Notation. We use the notation developed by Dijkstra and Scholten in [11]. Validity of a formula ϕ is denoted $[\phi]$. (We usually omit the variables on which ϕ depends when that can be determined from context.) Existential quantification of a set of variables V is denoted $(\exists V : \phi)$. Thus, if f and g are formulas representing sets, $[f \Rightarrow g]$ denotes the property that the set f is a subset of the set g. The advantage of this notation is in its succinctness and clarity, as will be seen in the rest of the paper.

2.1 Basics

A *program* is defined symbolically as a tuple (V, I, T), where V is a non-empty set of typed *variables*, I is a Boolean-valued *initial* assertion defined over V, and $T(V, V')$ is a Boolean-valued *transition* relation, defined over V and an isomorphic copy, V'. For each variable x in V, its copy, x', denotes the value of x in a successor state.

A program defines a *transition system*, represented by the tuple (S, S^0, R), as follows. The set of *states*, S, is the set of all (type-consistent) valuations to the variables V; the subset of *initial states*, S^0, is

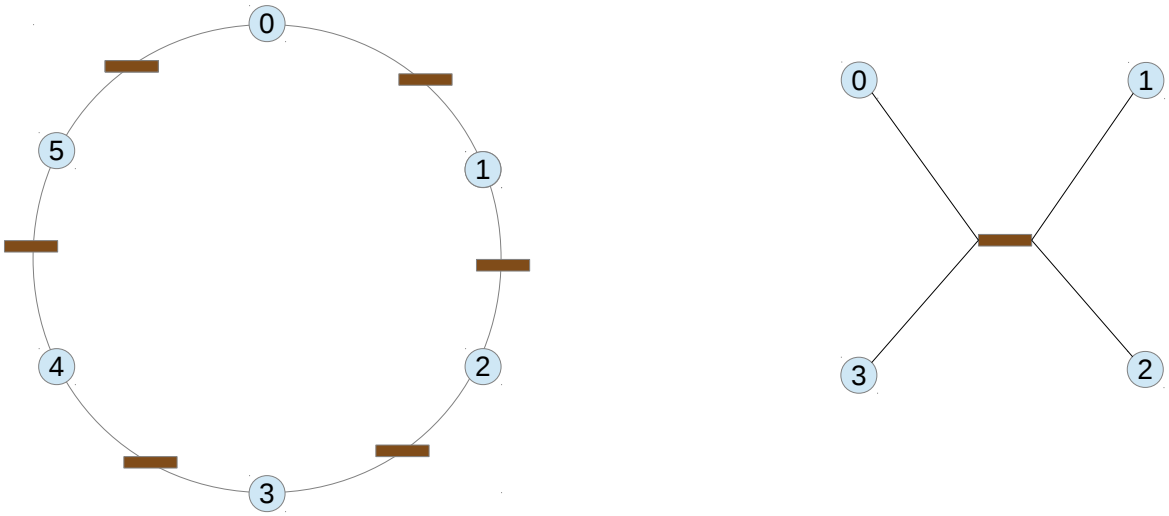

Figure 1: Network Structure: circles represent nodes and processes, rectangles represent edges and shared state. In these networks, connectivity is bidirectional.

those states which satisfy the initial condition I; and a pair of states, (s,t), is in the *transition relation* R if $T(s,t)$ holds.

To make the role of locality clear, we work with programs that are structured as a *process network*. The network is a graph structure where nodes are labeled with programs (also called processes) and edges are labeled with shared state. Formally, the graph underlying the network is a tuple (N,E,C), where N is a set of *nodes*, E is a set of *edges*, and C is a *connectivity* relation, a subset of $(N \times E) \cup (E \times N)$. The structure of a ring network and that of a star network is shown in Figure 1.

The *neighborhood* of a node is the set of edges that are connected to it. I.e., for a node n, the set of edges $\{e \mid (e,n) \in C \vee (n,e) \in C\}$ forms its neighborhood. Two nodes are *adjacent* if their neighborhoods have an edge in common. A node m *points-to* a node n if there is an edge e in the neighborhood of n such that (m,e) is in C.

An *assignment* is a mapping of programs to the nodes of the network and of state variables to the edges. This must be done so that the program which is mapped to node n only accesses, other than its internal variables, only those external variables which are mapped to edges in the neighborhood of n. We also require that the process only reads those variables on edges e such that (e,n) is a connection, and only writes those variables on edges e such that (n,e) is a connection. The semantics of the process network is formally defined as a program $P = (V,I,T)$, where

- V is the union of all program variables $V = (\cup i : V_i)$. The variables in V_i which are not mapped to network edges are the *internal variables* of process P_i, and are denoted by L_i.
- I is any initial condition over the program state, whose projection on V_i is I_i. Notationally, $[(\exists V \backslash V_i : I) \equiv I_i]$.
- T is the transition condition which enforces asynchronous interleaving. Formally, $[T \equiv (\vee i : T_i \wedge unch(V \backslash V_i)]$. I.e., T is the disjunction of the individual process transitions, under the constraint that the transition of process P_i leaves all variables other than those of V_i unchanged. For a simpler

notation, we adopt the convention that T_i is defined so that it leaves other variables unchanged.Then T can be written as $(\vee i : T_i)$.

The set of *reachable states* of a program $P = (V, I, T)$ is denoted by $Reach(P)$, and is defined as the least fixpoint expression $(\mu Z : I \vee SP(T, Z))$. The fixpoint expression denotes the least (smallest, strongest) set Z which satisfies the fixpoint constraint $[Z \equiv I \vee SP(T, Z)]$. The *strongest post-condition* operator is denoted SP; for a transition relation T and a set of states Z, the expression $SP(T, Z)$ denotes the immediate successors of states in Z due to transitions in T. Formally, $SP(T, Z) = \{t \mid (\exists s : Z(s) \wedge T(s, t)\}$.

A set of states (or assertion) φ is an *invariant* for the program if it is true for all reachable states, i.e., $[Reach(P) \Rightarrow \varphi]$. An assertion φ is *inductively invariant* if it is invariant, and also closed under program transitions. These conditions can be succinctly expressed by (1) (initiality) $[I \Rightarrow \varphi]$, and (2) (step) $[SP(T, \varphi) \Rightarrow \varphi]$. Inductive invariance forms the basis of proof rules for program correctness.

2.2 Generalized Dining Philosophers' Protocol

We will use a Dining Philosophers' protocol as a running example. The protocol consists of a number of similar processes operating on an arbitrary network. Every edge on the network models a shared "fork". The edge between nodes i and j is called f_{ij}. Its value can be one of $\{i, j, \perp\}$. Node i is said to *own* the fork f_{ij} if $f_{ij} = i$; node j owns this fork if $f_{ij} = j$; and the fork is available if $f_{ij} = \perp$.

The process at node i goes through the following internal states: T (thinking); H (hungry); E (eating); and R (release), which are the values of its internal variable, L. The local state of a node also includes the state of each of its adjacent edges (i.e., forks). Let $nbr(i, j)$ be a predicate true for nodes i, j if they share an edge. The transitions for a process are defined in guarded command notation as follows.

- A transition from T to H is always enabled. I.e., $(L = T) \longrightarrow L := H$
- In state H, the process acquires forks, but may also choose to release them
  - (acquire fork) $(L = H) \wedge nbr(i, j) \wedge f_{ij} = \perp \longrightarrow f_{ij} := i$,
  - (release fork) $(L = H) \wedge nbr(i, j) \wedge f_{ij} = i \longrightarrow f_{ij} := \perp$, and
  - (to-eat) $(L = H) \wedge (\forall j : nbr(i, j) : f_{ij} = i) \longrightarrow L := E$.
- A transition from E to R is always enabled. I.e., $(L = E) \longrightarrow L := R$.
- In state R, the process releases its owned forks.
  - (release fork) $(L = R) \wedge nbr(i, j) \wedge f_{ij} = i \longrightarrow f_{ij} := \perp$
  - (to-think) $(L = R) \wedge (\forall j : nbr(i, j) : f_{ij} \neq i) \longrightarrow L := T$

The initial state of the system is one where all processes are in internal state T and all forks are available (i.e., have value $\perp$). The desired safety property is that there is no reachable global state where two neighboring processes are in the eating state E.

2.3 Split Invariance

An inductive invariant, in general, depends on all program variables; i.e., it can express arbitrary constraints among the program variables. The divide and conquer principle suggests that one should break up an invariance assertion into a number of assertions which are limited in scope, each depends only on the variables of a single process. Hence, we define a *split assertion* θ to be a conjunction, written $(\wedge i : \theta_i)$, of a number of local assertions $\{\theta_i\}$. The i'th assertion, $\theta_i(V_i)$, is a function only of the variables of process P_i; i.e., its internal variables, and those assigned to the neighborhood of node i.

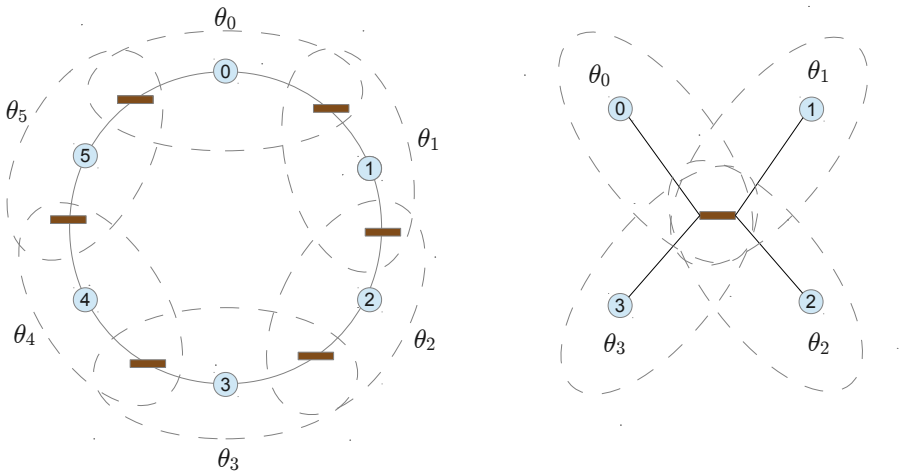

Figure 2: Split Invariance. A dotted ellipse shows the scope of a term of the split invariant.

Figure 2 gives a pictorial view of a split invariant for the ring and star networks, showing the scope of each of its terms. The terms for adjacent nodes have the shared variables in common; this sets up (weak) constraints between the invariant states of those processes.

We now consider the conditions for a split assertion to be a global inductive invariant. Examining the initiality and step conditions, one notices that, as the split assertion is conjunctive and *SP* distributes over the disjunction of transition relations, those conditions simplify to the equivalent set of constraints given below.

$$[I \Rightarrow \theta_i] \tag{1}$$
$$[SP(T_i, \theta) \Rightarrow \theta_i] \tag{2}$$
$$[SP(T_j, \theta) \Rightarrow \theta_i], \text{ for all } j \text{ which point to } i \tag{3}$$

In the last constraint, nodes j which do not point to i are not considered, as any action in T_j must leave the state of V_i unchanged since there are no variables in common.

The form of these constraints is remarkably similar to the "assume-guarantee" or Owicki-Gries rules for compositional reasoning, which can be stated as follows.

$$[I \Rightarrow \theta_i] \tag{4}$$
$$[SP(T_i, \theta_i) \Rightarrow \theta_i] \tag{5}$$
$$[SP(T_j, \theta_i \wedge \theta_j) \Rightarrow \theta_i], \text{ for } j \text{ points-to } i \tag{6}$$

The first two constraints show that θ_i is an invariant of process P_i by itself. The third constraint is Owicki and Gries' *non-interference* condition: a transition by any other process from a state satisfying both process' invariants, preserves θ_i. The closeness of the connection between the two formulations is shown by the following theorem.

```
var theta, new_theta: prediate array
// initialize
for i := 1 to N do new_theta[i] := emptyset done;
// compute until fixpoint
repeat
  theta := new_theta;
  for i := 1 to N do new_theta[i] := F(i,theta) done;
until (theta = new_theta)
```

Figure 3: Computing the Strongest Split Invariant.

Theorem 1 *Every solution to the assume-guarantee constraints is a split inductive invariant. Moreover, in a network where all processes refer to common shared state (such as the star network in Figure 1), the strongest solutions of the two sets of constraints are identical.*

For computational purposes, we are interested in the strongest solutions, as they correspond to least fixpoints. We will use the assume-guarantee form from now on, as it is is simpler to manipulate. As θ_i is defined only in terms of V_i, projecting the left-hand sides of the implications (4)-(6) on V_i gives an equivalent set of constraints:

$$[(\exists V \backslash V_i : I) \Rightarrow \theta_i] \tag{7}$$
$$[(\exists V \backslash V_i : SP(T_i, \theta_i)) \Rightarrow \theta_i] \tag{8}$$
$$[(\exists V \backslash V_i : SP(T_j, \theta_i \wedge \theta_j)) \Rightarrow \theta_i], \text{ for } j \text{ points-to } i \tag{9}$$

These constraints can be reworked into the simultaneous pre-fixpoint form (cf. [15, 24])

$$[F_i(\theta) \Rightarrow \theta_i] \tag{10}$$

where F_i is the disjunction of the left-hand-sides of equations (7)-(9). By the monotonicity of SP, the function F_i is monotonic in θ, considered now as a vector of local assertions, $(\theta_1, \ldots, \theta_N)$, ordered by point-wise implication. By the Knaster-Tarski theorem, there is a least fixpoint, which defines the strongest compositional invariant. It can be computed by the standard iteration shown in Figure 3.

The computation takes time polynomial in N, the number of processes in the network – a rough bound is $O(N^2 * L^3 * D)$, where L is the size of the local state space of a process, and D is the maximum degree of the network. This computation produces the split invariant for the Dining Philosophers on a 3000 node ring in about 1 second. A number of experimental results can be found in [6] and in [9].

2.4 Completeness

The split invariance formulation is, in general, incomplete. That is, it is not always possible to prove a program invariant by exhibiting a stronger split invariant. In part, this is indicated by the complexity bounds: the split invariance calculation is polynomial in N, while the invariance checking problem is PSPACE-complete in N. However, this reasoning depends on whether PSPACE=P. An direct, unconditional proof is given by the simple, shared-memory mutual exclusion program below. Every process of the program goes through states T (thinking), H (hungry), and E (eating). The desired invariance property is that no two processes are in state E together.

```
var x: boolean; initially true

process P(i):
var l: {T,H,E}; initially T

while (true) {
  T: skip;
  H: <x -> x := false>  // atomic test-and-set
  E: x := true
}
```

The fixpoint calculation produces the split invariant θ where $\theta_i = true$, for all i. This invariant clearly does not suffice to show mutual exclusion. As recognized by Owicki-Gries and Lamport, one can strengthen a split invariant by introducing auxiliary global variables which record part of the history of the computation. Intuitively, the auxiliary state helps to tighten the constraints between the θ components, as a pair of local invariant states must agree on the shared auxiliary state. For this example, it suffices to introduce an auxiliary global variable, `last`, which records the last process to enter its E state.

```
var x: boolean; initially true
var last: 0..N; initially 0

process P(i):
var l: {T,H,E}; initially T

while (true) {
  T: skip;
  H: <x -> x := false; last := i>  // atomic test-and-set
  E: x := true
}
```

In the fixpoint, the i'th component θ_i is given by $(E_i \equiv (\neg x \wedge last = i))$. This suffices for mutual exclusion – if distinct processes P_m and P_n are both in state E, then $last$ must simultaneously be equal to m and to n, which is impossible.

An important question in automated compositional model checking is, therefore, the development of heuristics for discovering appropriate auxiliary variables. For split invariance, one such heuristic is developed in [6]. It is based on a method which analyzes counter-examples to expose aspects of the internal state of a process as an auxiliary global predicate. The method is complete for finite-state processes: in the worst case, all of the internal process state is exposed as shared state, which implies that the fixpoint computation turns into reachability on the global state space. The intuition is that for many protocols, it is unnecessary to go to this extreme in order to obtain a strong enough invariant. This intuition can be corroborated by experiments such as those in [6]. In the automaton-learning approach to compositional verification [5], the auxiliary state is represented by the states of the learned automata.

3 Local Symmetries

Many concurrent programs have inherent symmetries. For instance, the mutual exclusion protocol of the previous section has a fully symmetric state space, while the Dining Philosopher's protocol when

run on a ring network has state space which is invariant under ring rotations. Symmetries can be used to reduce the state space that must be explored for model checking, as shown in the pioneering work in [3, 14, 20]. Global symmetry reduction, however, works well only for fully symmetric state spaces, where it can result in an exponential reduction. For a number of other regular networks, such as the ring, torus, hypercube, and mesh networks, there is not enough global symmetry: the state space reductions are usually at most linear.

This earlier work on symmetry is connected to model checking on the full state space. What is the corresponding notion for compositional methods? It is the nature of compositional reasoning that the invariant of a process depends only on that of its neighbors. Thus, intuition suggests that it should suffice for the network to have enough *local* symmetry. For example, any two nodes on a ring network are locally symmetric: each has a single left neighbor and a single right neighbor. Torus, mesh and hypercube networks also have similar local symmetries.

Technically, the notion of local symmetry is best described by a *groupoid* [31]. A groupoid is a weaker object than a group (which is used to describe global symmetries), but has many similar properties. We use a specific groupoid, developed in [17], which defines the local symmetries of a network. The elements of the network groupoid are triples of the form (m,β,n), where m and n are nodes of the network, and β is an isomorphism on their neighborhoods which preserves the direction of connectivity. (I.e., (m,e) is a connection if, and only if $(n,\beta(e))$ is a connection and, similarly, (e,m) is a connection if, and only if, $(\beta(e),n)$ is a connection.) We call such a triple a *local symmetry*. Local symmetries have group-like properties:

- The composition of local symmetries is a symmetry: if (m,β,n) and (n,δ,k) are symmetries, so is $(m,\delta\beta,k)$
- The symmetry (m,id,m) is the identity of the composition
- If (m,β,n) is a symmetry, the symmetry (n,β^{-1},m) is its inverse.

The set of all local symmetries forms the *network groupoid*.

One may reasonably conjecture that nodes which are locally symmetric have isomorphic compositional invariants. I.e., if (m,β,n) is a symmetry, then θ_m and θ_n are isomorphic up to β. This is, however, not true in general. The reason is that the compositional invariant computed at nodes m and n depends on the invariants computed at adjacent nodes, and those must be symmetric as well. Thus, one is led to a notion of recursive similarity, called *balance* [17]. This has a co-inductive form like that of bisimulation.

A balance relation B is a sub-groupoid of the network groupoid, with the following property: if (m,β,n) is in B, and node k points to m, there is a node l which points to n and an isomorphism δ, such that (k,δ,l) is in B. Moreover, β and δ must agree on the mapping of edges which are common to the neighborhoods of m and k.

The utility of the balance relation is given by the following theorems.

Theorem 2 *(From [25])*

1. *If G is a group of automorphisms for the network, then the set $\{(m,\beta,n) \mid \beta \in G \wedge \beta(m) = n\}$ is a balance relation.*
2. *Let θ be the strongest compositional invariant for a network. If (m,β,n) is in a balance relation, then $[\theta_n \equiv \langle\beta\rangle\theta_m]$.*

Informally, the first result shows that the global symmetry group induces balanced local symmetries; this is a quick way of determining a balance relation for a network. The second shows that the local invariants for a pair of balanced nodes are isomorphic. Here, $\langle\beta\rangle$ is a pre-image operator that maps states over V_m to states over V_n using β to relate the values of corresponding edges.

Local Symmetry Reduction. This theorem points the way to symmetry reduction for compositional methods. The idea is to compute fixpoint components only for representatives of local symmetry classes. The group-like properties ensure that for any groupoid, its *orbit relation*, defined as $m \sim n$ if there is β such that (m,β,n) is in the groupoid, is an equivalence. For a ring network, it suffices to compute a single component, rather than all N components! The calculation is thus independent of the size of the network. This has interesting consequences for parametric proofs, as explained in the next section.

4 Local Abstractions

The symmetry reductions described in the previous section are applicable to several networks which have only a small amount of global symmetry. Still, protocols have other local symmetries which cannot be captured by this definition. For instance, consider the Dining Philosophers' protocol on an arbitrary network. Every node operates in a roughly similar fashion, attempting to own all of its forks before entering the eating state. However, nodes with differing numbers of adjacent edges cannot be locally symmetric – there can be no isomorphism between their neighborhoods. In fact, a network may be so irregular as to have only the trivial symmetry groupoid.

In order to be able to represent these other symmetries, we must abstract away from the structural differences between nodes. It suffices to define an abstraction function over the local state of a node. As explained in more detail in [26], a *local abstraction* is formally specified by defining for each node m an abstract domain, D_m, and a total abstraction function, α_m, which maps local states of P_m to elements of D_m. This induces a Galois connection on subsets, which we also refer to as (α_m, γ_m): $\alpha_m(X) = \{\alpha_m(x) \mid x \in X\}$, and $\gamma_m(A) = \{x \mid \alpha_m(x) \in A\}$.

We must adjust the fixpoint computation to operate at the abstract state level. The abstract set of initial states, $\overline{I}_m$ is given by $\alpha_m(I_m)$. The abstract step transition, $\overline{T}_m$, is obtained by standard existential abstraction: there is a transition from (abstract) state a to (abstract) state b if there exist local states x, y such that $\alpha_m(x) = a$, $\alpha_m(y) = b$, and $T_m(x,y)$ holds. An abstract transition (a,b) for node m is the result of interference by a transition of node k from θ_k if the following holds.

$$(\exists s,t : \alpha_m(s[m]) = a \ \wedge \ \alpha_m(t[m]) = b \ \wedge \ T_k(s,t) \ \wedge \ \alpha_k(s[k]) \in \theta_k) \tag{11}$$

For the Dining Philosophers' protocol, such a function can be defined through a predicate, A, which is true at a local state if, and only if, the node owns all forks in that state. The abstract state of a node is now a pair (l,a) where l is its internal state (one of T,H,E,R) and a is the value of the predicate A. With this definition, the abstract fixpoint calculation produces the local invariant shown as a transition graph in Figure 4. This graph shows that the abstract invariant for node m implies that $(E_m \ \Rightarrow \ \mathsf{A}_m)$. Concretizing this term, one obtains that the concrete invariant implies that if E_m is true, then node m owns all of its forks. This, in turn, implies the exclusion property, as adjacent nodes m and n cannot both own the common fork f_{mn} in the same global state.

There are two features to note of this transition graph. First, all interference transitions are self-loops – i.e., the actions of neighboring processes do not change the abstract state of a process. This is due to the protocol: the action of a neighbor cannot cause a process to own a fork, or to give up one that it owns. Second, all nodes in any network fall into one of the two classes which are shown, in terms of their abstract compositional invariant. It follows that the concretized compositional invariant holds in a parametric sense: i.e., over *all* nodes of *all* networks.

This connection between compositional reasoning and parametric proofs is not entirely unexpected. Parametric invariants for protocols often have the universally quantified form "for every node n of an in-

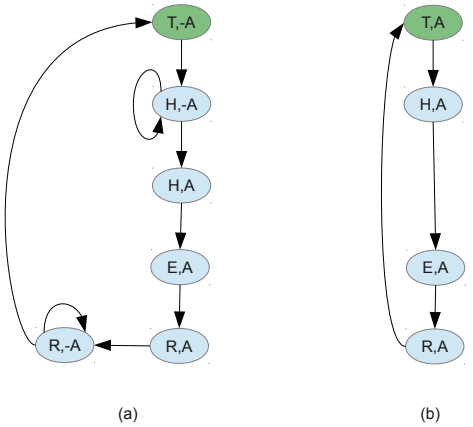

Figure 4: (From [26]) Abstract State Transitions (a) for non-isolated nodes and (b) for an isolated node. The notation "−A" indicates the negation of A. Green/dark states are initial.

stance, $\theta(n)$ holds". If the property $\theta(n)$ is restricted to the neighborhood of n and holds compositionally, which is often the case, then the property is a split invariant for every fixed-size instance. The application of abstraction and symmetry serves to turn this connection around: computing a compositional invariant on an (abstract) instance induces a parametric invariant.

The following theorem shows that this is a complete method – but it is not automatic, as it requires the choice of a proper abstraction. The abstraction in the theorem can be chosen so that every pair of nodes is locally symmetric in terms of its abstract state space, and the cross-node interference is benign, as in the illustration above.

Theorem 3 *(From [26]) For a parameterized family of process networks, any compositional invariant of the form* $(\forall i : \theta_i)$*, where each* θ_i *is local to process* P_i*, can be established by compositional reasoning over a small abstract network.*

5 Related Work

The book [29] has an excellent description of the Owicki-Gries method and other compositional methods. The fixpoint formulation for computing the strongest split invariant is implicit in the deduction system of [15] and is explicitly formulated in [24]. Other closely related work on compositional verification has been referenced in the previous sections.

Local symmetry and balance are originally defined in [17] in a slightly different form. That paper analyzes the role of local symmetry to prove *existential* path properties of *continuous* systems; it is remarkable that those definitions also serve to analyze universal properties of discrete systems. it is explicitly stated

Parametric verification is undecidable in general, even if each process has a small, fixed number of states [2]. A common thread running through the various approaches to parametric verification is the

intuition that for a correct protocol, behaviors of very large instances are already present in some form in smaller instances. Decidability results [16, 13, 12] are based on "cutoff" theorems which establish that it suffices to check all instances up to the cutoff size, or on well-quasi-ordering of the transition structure [1]. The method of invisible invariants [28] generalizes an invariant computed automatically for a small instance to an inductive invariant for all instances. In [24], it was shown that the success of generalization is closely related to the invariant being a split invariant; the parametric analysis based on abstractions and local symmetry that is carried out in this paper is a further extension of those results. The "environmental abstraction" procedure [4] analyzes a single process in the context of an approximation of the rest of the system. Although the approximation is developed starting with the full state space, there is a close similarity between the final method and compositional reasoning. Related procedures include [23] and [30].

6 Conclusions

In this article, I have attempted to show that compositional reasoning is a topic with a rich theory and practically relevant application. There are pleasing new connections to the new concept (in verification) of local symmetry, and to long-established ones such as abstraction and parametric reasoning. In this article I have chosen to focus on the simplest form of compositional reasoning, that used to construct inductive invariants, but the methods extend to general (i.e., possibly non-inductive) invariance, as well as to proofs of temporal properties under fairness assumptions [7, 8, 9]. The simultaneous fixpoint calculation lends itself to parallelization, as the individual components can be computed asynchronously so long as the computation schedule is fair [10]. It is worth noting that the theory applies to arbitrary state spaces under appropriate abstractions, as shown by the work in [18], which applies compositional reasoning to C programs.

There are many open questions. Among the major ones are the following: Why are certain protocols more amenable to compositional methods than others? ("Loose coupling" is sometimes offered as an answer, but that term does not have a precise definition.) Can one create better methods which compute only as much auxiliary state as is necessary for a proof? What sorts of abstractions are useful for parametric proofs?

Closing. I would like to thank the referees for helpful comments on the initial draft of this paper. The work described here would not have been possible without the varied and immensely enjoyable collaborations with my co-authors: Ariel Cohen, Yaniv Sa'ar, Lenore Zuck, and Richard Trefler. My co-authors on a survey of compositional verification, Corina Păsăreanu and Dimitra Giannakopoulou, contributed many insights into the methods. I am very glad to be able to offer this as my contribution to David Schmidt's Festschrift. It is a small return for the respect I have for his research work, for the help and advice I received from him when co-organizing VMCAI in 2006, and for many enjoyable conversations!

My initial work on compositional reasoning was supported in part by the NSF, under award CCR 0341658. The writing of this paper was done while I was supported in part by DARPA, under agreement number FA8750-12-C-0166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson & Yih-Kuen Tsay (1996): *General Decidability Theorems for Infinite-State Systems*. In: *LICS*, IEEE Computer Society, pp. 313–321, doi:10.1109/LICS.1996.561359.

[2] Krzysztof R. Apt & Dexter Kozen (1986): *Limits for Automatic Verification of Finite-State Concurrent Systems*. *Inf. Process. Lett.* 22(6), pp. 307–309, doi:10.1016/0020-0190(86)90071-2.

[3] E. M. Clarke, T. Filkorn & S. Jha (1993): *Exploiting Symmetry in Temporal Logic Model Checking*. In: *CAV*, *LNCS* 697, pp. 450–462, doi:10.1007/3-540-56922-7_37.

[4] Edmund M. Clarke, Muralidhar Talupur & Helmut Veith (2006): *Environment Abstraction for Parameterized Verification*. In: *VMCAI*, *LNCS* 3855, pp. 126–141, doi:10.1007/11609773_9.

[5] Jamieson M. Cobleigh, Dimitra Giannakopoulou & Corina S. Pasareanu (2003): *Learning Assumptions for Compositional Verification*. In: *TACAS*, *LNCS* 2619, Springer, pp. 331–346, doi:10.1007/3-540-36577-X_24.

[6] A. Cohen & K. S. Namjoshi (2007): *Local Proofs for Global Safety Properties*. In: *CAV*, *LNCS* 4590, Springer, pp. 55–67, doi:10.1007/978-3-540-73368-3_9.

[7] A. Cohen & K. S. Namjoshi (2008): *Local Proofs for Linear-Time Properties of Concurrent Programs*. In: *CAV*, *LNCS* 5123, Springer, pp. 149–161, doi:10.1007/978-3-540-70545-1_15.

[8] A. Cohen, K. S. Namjoshi & Y. Sa'ar (2010): *A Dash of Fairness for Compositional Reasoning*. In: *CAV*, pp. 543–557, doi:10.1007/978-3-642-14295-6_46.

[9] A. Cohen, K. S. Namjoshi & Y. Sa'ar (2010): *SPLIT: A Compositional LTL Verifier*. In: *CAV*, pp. 558–561, doi:10.1007/978-3-642-14295-6_47.

[10] Ariel Cohen, Kedar S. Namjoshi, Yaniv Sa'ar, Lenore D. Zuck & Katya I. Kisyova (2010): *Parallelizing A Symbolic Compositional Model-Checking Algorithm*. In: *HVC*, *LNCS* 6504, pp. 46–59, doi:10.1007/978-3-642-19583-9_9.

[11] E.W. Dijkstra & C.S. Scholten (1990): *Predicate Calculus and Program Semantics*. Springer Verlag, doi:10.1007/978-1-4612-3228-5.

[12] E. Allen Emerson & Vineet Kahlon (2000): *Reducing Model Checking of the Many to the Few*. In: *CADE*, *LNCS* 1831, pp. 236–254, doi:10.1007/10721959_19.

[13] E.A. Emerson & K.S. Namjoshi (1995): *Reasoning about Rings*. In: *ACM Symposium on Principles of Programming Languages*, doi:10.1145/199448.199468.

[14] E.A. Emerson & A.P. Sistla (1993): *Symmetry and Model Checking*. In: *CAV*, *LNCS* 697, pp. 463–478, doi:10.1007/3-540-56922-7_38.

[15] C. Flanagan & S. Qadeer (2003): *Thread-Modular Model Checking*. In: *SPIN*, *LNCS* 2648, pp. 213–224, doi:10.1007/3-540-44829-2_14.

[16] S. German & A.P. Sistla (1992): *Reasoning about Systems with Many Processes*. *Journal of the ACM*, doi:10.1145/146637.146681.

[17] Martin Golubitsky & Ian Stewart (2006): *Nonlinear dynamics of networks: the groupoid formalism*. *Bull. Amer. Math. Soc.* 43, pp. 305–364, doi:10.1090/S0273-0979-06-01108-6.

[18] Ashutosh Gupta, Corneliu Popeea & Andrey Rybalchenko (2011): *Predicate abstraction and refinement for verifying multi-threaded programs*. In: *POPL*, ACM, pp. 331–344, doi:10.1145/1926385.1926424.

[19] Gerard J. Holzmann (2004): *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

[20] C.N. Ip & D. Dill (1996): *Better Verification Through Symmetry*. *Formal Methods in System Design* 9(1/2), pp. 41–75, doi:10.1007/BF00625968.

[21] L. Lamport (1977): *Proving the Correctness of Multiprocess Programs*. *IEEE Trans. Software Eng.* 3(2), doi:10.1109/TSE.1977.229904.

[22] Leslie Lamport (1997): *Composition: A Way to Make Proofs Harder*. In: *COMPOS*, pp. 402–423, doi:10.1007/3-540-49213-5_15.

[23] Kenneth L. McMillan & Lenore D. Zuck (2011): *Invisible Invariants and Abstract Interpretation*. In Eran Yahav, editor: *SAS*, *Lecture Notes in Computer Science* 6887, Springer, pp. 249–262, doi:10.1007/978-3-642-23702-7_20.

[24] K. S. Namjoshi (2007): *Symmetry and Completeness in the Analysis of Parameterized Systems*. In: *VMCAI*, *LNCS* 4349, pp. 299–313, doi:10.1007/978-3-540-69738-1_22.

[25] Kedar S. Namjoshi & Richard J. Trefler (2012): *Local Symmetry and Compositional Verification*. In: *VMCAI*, *LNCS* 7148, pp. 348–362, doi:10.1007/978-3-642-27940-9_23.

[26] Kedar S. Namjoshi & Richard J. Trefler (2013): *Uncovering Symmetries in Irregular Process Networks*. In: *VMCAI*, LNCS 7737, pp. 496–514, doi:10.1007/978-3-642-35873-9_29.

[27] S. S. Owicki & D. Gries (1976): *Verifying Properties of Parallel Programs: An Axiomatic Approach. Commun. ACM* 19(5), pp. 279–285, doi:10.1145/360051.360224.

[28] A. Pnueli, S. Ruah & L. D. Zuck (2001): *Automatic Deductive Verification with Invisible Invariants.* In: *TACAS*, *LNCS* 2031, pp. 82–97, doi:10.1007/3-540-45319-9_7.

[29] W-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel & J. Zwiers (2001): *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods*. Cambridge University Press.

[30] Alejandro Sánchez, Sriram Sankaranarayanan, César Sánchez & Bor-Yuh Evan Chang (2012): *Invariant Generation for Parametrized Systems Using Self-reflection - (Extended Version)*. In Antoine Miné & David Schmidt, editors: *SAS*, *Lecture Notes in Computer Science* 7460, Springer, pp. 146–163, doi:10.1007/978-3-642-33125-1_12.

[31] Alan Weinstein (1996): *Groupoids: Unifying Internal and External Symmetry-A Tour through Some Examples*. *Notices of the AMS*.