A Dash of Fairness for Compositional Reasoning*

Ariel Cohen[1], Kedar S. Namjoshi[2], and Yaniv Sa'ar[3]

[1] New York University, New York, NY
arielc@cs.nyu.edu
[2] Bell Labs, Alcatel-Lucent, Murray Hill, NJ
kedar@research.bell-labs.com
[3] Weizmann Institute of Science, Rehovot, Israel
yaniv.saar@weizmann.ac.il

Abstract. Proofs of progress properties often require fairness assumptions. Directly incorporating global fairness assumptions in a compositional method is difficult, given the local flavor of such reasoning. We present a fully automated local reasoning algorithm which handles fairness assumptions through a process of iterative refinement. Refinement strengthens local proofs by the addition of auxiliary shared variables which expose internal process state; it is needed as local reasoning is inherently incomplete. Experiments demonstrate that the new algorithm shows significant improvement over standard model checking.

1 Introduction

Model checking is fundamentally constrained by state explosion [6]: for concurrent programs, the state space can grow exponentially with the number of processes. A promising approach to ameliorating state explosion is to decompose a verification task so that the reasoning is as localized as possible. In this work, we propose and evaluate a new algorithm which carries out compositional reasoning for temporal properties which hold only under global fairness assumptions.

Fairness assumptions are often needed for proofs of progress properties. It has long been understood how to incorporate fairness in standard model checking [5,14], but doing so is a challenge for compositional methods. The difficulty is that fairness assumptions commonly refer to local state from a number of processes. For example, a common (strong) fairness constraint is that "*for every process: if the process is enabled infinitely often, it is infinitely often executed*". As "enabledness" depends on local state, this assumption refers to the local state of every process. Since compositional reasoning is based on a per-process view, the presence of such global assumptions can be problematic.

This work develops a new algorithm for compositional model checking with fairness assumptions, which tackles this problem with a successive refinement method. It also

* This research was supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant awarded to David Harel from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 543–557, 2010.

presents a new compositional proof rule for verification under fairness. Moreover, the model checking algorithm can be instrumented to generate a valid instantiation of the proof rule upon success.

The new algorithm is the continuation of a line of research on mechanizing assertional (i.e., state-predicate based) compositional verification. The starting point is an algorithm from [22] which computes the strongest *split invariant*. A split invariant is a vector of interference-free, per-process invariants. (A set of per-process invariants is free of interference [24,19] if the action of one process does not invalidate the invariant of another process.) The term "split invariant" is used as the conjunction of the local invariants forms a inductive invariant for the program as a whole. The strongest split invariant may be weaker than the set of reachable states, and therefore not strong enough to prove a safety property. In [8], we solve this problem by formulating an complete verification procedure which strengthens the split invariant by discovering and adding auxiliary shared variables to track local predicates. In [9], we use split invariance as the basis for a new compositional algorithm for checking LTL properties. Experiments reported in these papers show that assertional local reasoning can be significantly faster than monolithic (i.e., non-compositional) model checking.

The local liveness method of [9] does not directly apply to fairness constraints. This is because the method is sound only for properties expressed over shared variables. Incorporating fairness into the specification, through the identity $M \models \mathsf{A}_{Fair}(Spec) \equiv M \models \mathsf{A}(Fair \Rightarrow Spec)$, results in a new specification which names a number of local variables (due to $Fair$). One can, of course, turn all the local variables in $Fair$ into shared variables, but this defeats the purpose of local reasoning.

The new algorithm gets around this difficulty by a process of iterative refinement. The fairness constraint is replaced with a weaker form, which depends monotonically on the current split invariant, and is expressed over only the shared variables. This allows using the compositional algorithm from [9], with slight modifications. If verification succeeds with the weaker fairness assumption, the property is proved. If not, a bogus counter-example is produced, and analyzed to discover new local predicates which are then exposed as auxiliary shared variables. Exposing local state strengthens the split invariant in the next round of computation, which strengthens the abstracted fairness assumption by monotonicity. This is repeated until a decisive result (either success or a real counter-example) is obtained. The iterative process terminates—and is thus complete—for finite-state programs: eventually, enough of the local state is exposed to either prove a correct property or to disprove an incorrect one. Moreover, it is possible to disprove a property *without* building up the entire state space.

The algorithm, being predicate-based, has a simple implementation using BDDs. We carry out an evaluation with several parameterized protocols, where each instance of the protocol is finite-state. The experimental results show promise: the compositional verification is faster in almost all cases, sometimes by one or two orders of magnitude. Exposing a limited amount of local state suffices for both proofs and disproofs of properties, validating the basic premise behind compositional reasoning.

2 Related Work

The question of handling fairness in compositional verification is a natural and important one. The comprehensive book on compositional methods by de Roever et. al. [11], however, does not mention a compositional proof rule directly incorporating fairness. Compositional proof rules for general LTL properties (e.g., [1,20,21,23]) can handle fairness only by compiling it into the specification. To the best of our knowledge, this is the first compositional algorithm and proof rule to directly incorporate fairness.

The methods used here are assertional; i.e., they are based on computing state predicates. The "thread-modular" reasoning method [16] computes a split invariant using explicit-state representations, but is limited to safety properties. An alternative line of work on automated compositional reasoning is based on representing interface behavior, and is thus behavioral in nature. One instance of this method uses the following complete proof rule: to show $M_1//M_2 \models Spec$, find an interface automaton A such that $M_1 \models A$ and $M_2//A \models Spec$. (Here, $\models$ is read as language inclusion.) The procedures developed in [17,27] employ a combination of model checking and finite-automaton learning via variants of the L^* method [2] to construct an appropriate automaton A. Standard learning algorithms compute automata on finite words, and hence can be used only for proofs of safety properties. An algorithm is developed in [15] for learning a Büchi automaton, but it has not yet been applied to verification of progress properties. Although an automaton is a powerful and compact representation object, current implementations of behavioral methods have difficulty showing a significant improvement over non-compositional model checking [7].

3 Background: Local Reasoning and Liveness Properties

This section defines the system model and split invariance, and gives a short summary of the method for local liveness checking. Part of this material is taken from [22,8,9], and is repeated here for convenience.

A Note on Notation. Throughout the paper, we use notation based on that of Dijkstra and Scholten [12]. Sets of program states are represented by first-order formula on program variables. Existential quantification of formula ξ by a set of variables X is written as $(\exists X : \xi)$. The notation $[\xi]$ stands for "ξ is valid". The successor operation is denoted by sp (for strongest post-condition): $sp(\tau, \xi)$ represents the set of states reachable from states satisfying ξ in one τ-transition. The notation $sp_i(\tau, \xi)$ is used for successors computed within the state space of process P_i.

3.1 Model: Asynchronous, Shared-Memory Composition

A *process* is given by a tuple (V, I, T), where V is a set of (typed) variables, $I(V)$ is a predicate over V defining an initial condition, and $T(V, V')$ is a predicate defining a transition condition, where V' is a fresh set of variables in 1-1 correspondence with V. The semantics of a process is given by a *transition system* in the standard way.

The *asynchronous composition* of processes $\{P_i\}$ is written as $\|_i P_i$. For convenience, we suppose that there is a set of variables, X, called the *shared variables*, and

sets of variables, $\{L_i\}$, called the *local variables*, such that $V_i = X \cup L_i$ for each i, and L_i is disjoint from L_j, for $i \neq j$, and L_i is also disjoint from X. The components of the composition are defined as follows. Let $V = \bigcup_i V_i$ and $I = \bigwedge_i I_i$. The set of local variables is $L = \bigcup_i L_i$. Let $\hat{T}_i = T_i(V_i, V_i') \wedge (\forall j : j \neq i \;\Rightarrow\; unch(L_j))$, where $unch(W)$ is short for $\bigwedge_{w \in W}(w' = w)$. Thus, $\hat{T}_i$ behaves like T_i, but leaves local variables of other processes unchanged. The transition relation of the composition, T, is defined as $\bigvee_i \hat{T}_i$.

3.2 Split-Invariance: Definition and Calculation

Let $P = ||_k P_k$ be an N-process composition. For localized reasoning about invariance, the shape of invariance assertions is restricted to a conjunction of local (i.e., per-process) assertions. A *local* assertion is one that is based on the variables of a single process. A *split assertion* is a vector of local assertions, $\theta = (\theta_1, \theta_2, \ldots, \theta_N)$, one for each process, so that θ_i is defined on V_i (equivalently, on X and L_i). Split assertion θ is a *split invariant* if the conjunction of its components, i.e., $\bigwedge_k \theta_k$, is an inductive invariant for the full program P. Split-invariance can equivalently be defined as in Figure 1.

Definition 1. *The notation $T_k^\theta(X, X')$ denotes $(\exists L_k, L_k' : T_k \wedge \theta_k)$. This is a "summary transition", representing the effect that a move of P_k from a state satisfying its local invariant has on the shared variables.*

For each process index i:

1. **[initiality]** θ_i includes all initial states of process P_i. I.e., $[(\exists L \setminus L_i : I) \;\Rightarrow\; \theta_i]$
2. **[step]** θ_i is closed under transitions of P_i. I.e., $[sp_i(T_i, \theta_i) \;\Rightarrow\; \theta_i]$
3. **[non-interference]** θ_i is closed under transitions (interference) by processes other than P_i. I.e., for all k different from i, $[sp_i(T_k^\theta \wedge unch(L_i), \theta_i) \;\Rightarrow\; \theta_i]$

Fig. 1. Split Invariance Conditions

These conditions are a simple instance of (syntactically circular) *assume-guarantee reasoning*: θ_i is the invariance guarantee provided by process i, based on assumptions $\{\theta_j : j \neq i\}$ about the other processes. The constraints can be gathered into the set of simultaneous implications: for each i,

$$[(\exists L \setminus L_i : I) \vee sp_i(T_i, \theta_i) \vee (\vee k : k \neq i : sp_i(T_k^\theta \wedge unch(L_i), \theta_i)) \;\Rightarrow\; \theta_i] \quad (1)$$

Theorem 1. *(Namjoshi [22]) The simultaneous least fixpoint of equations* (1) *exists by the Knaster-Tarski fixpoint theorem. This defines the strongest split invariant.*

3.3 Incompleteness and Auxiliary Variables

Local reasoning is inherently incomplete. This is illustrated by the mutual exclusion protocol from Figure 2. The strongest split invariant for 2 processes is $(true, true)$, which is too weak to prove mutual exclusion. A general mechanism for overcoming

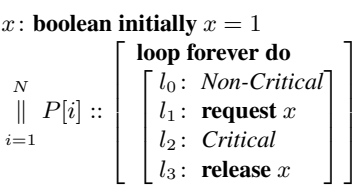

Fig. 2. MUXSEM

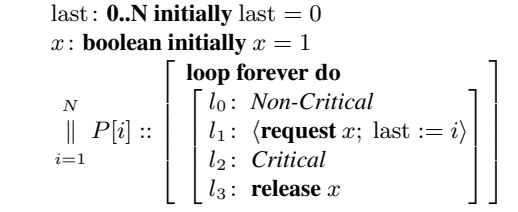

Fig. 3. MUXSEM with auxiliary variable

Local Liveness (LL) Algorithm

1. Compute the strongest split invariant, θ.
2. For each i: build an abstract form of process i, called P_i^θ, with initial states given by $(\exists L \setminus L_i : I)$, and two kinds of transitions:
   - the transition T_i of process i, and
   - summary transitions T_j^θ (see Defn. 1) for all other processes P_j $(j \neq i)$
3. Form a Büchi automaton for the *negated* specification. For each i, form the synchronous product of this automaton with P_i^θ and check that *there is no computation where infinitely often there is a process i transition from a Büchi accepting state*
4. Declare success if the check succeeds **for each** abstract process

Fig. 4. Local Liveness (LL) Algorithm

incompleteness, proposed by Owicki-Gries and Lamport [19], is to add auxiliary shared variables which expose portions of the local state or execution history. In Figure 3, an auxiliary variable records the last process to enter the critical section. The strongest split invariant for the augmented protocol is given by $\theta_i \equiv (l_2(i) \equiv (x = 0) \wedge (last = i))$, which suffices to prove mutual exclusion as $[\theta_i \wedge \theta_j \wedge (i \neq j) \Rightarrow \neg(l_2(i) \wedge l_2(j))]$. The discovery of auxiliary predicates can be effectively automated [8].

3.4 Local Verification of Liveness Properties

Owicki and Gries also developed compositional proof rules for termination. In [9], a related proof rule is turned into a compositional algorithm for checking general linear-time temporal properties. This "local liveness" method, referred to subsequently as the LL algorithm, is shown in Figure 4. We give a sketch of its soundness proof, as this is important for the extension to fairness. The LL algorithm requires that the LTL property is expressed by shared variables. With this method, one can show that the property "infinitely often $(x = 0)$" holds for the protocol in Figure 2—i.e., that some process is in the critical section infinitely often. Starvation freedom, however, holds only under a strong fairness assumption, and its compositional proof requires the new method.

Theorem 2. *(Cohen-Namjoshi [9]) The LL method is sound.*

Proof Sketch. The soundness proof shows the following: if a property does not hold, any global counter-example can be projected to a counter-example for **some** abstract process. Let σ be a global counter-example. Then (1) each state of σ must satisfy the

split invariant and (2) the Büchi automaton must accept infinitely often along σ. As there is a fixed number of processes, by (2), there is a process, say P_i, whose transition is executed infinitely often from a Büchi accepting state along σ. Consider the abstract process P_i^θ formed out of P_i. The computation σ can be projected, transition-by-transition, to an execution of P_i^θ. A transition by process P_i is kept as is; a transition by another process, say P_k, is replaced by its summary transition, T_k^θ (detailed proof is in [9]). Any summary transition preserves the change to shared variables made by the original; hence, the sequence of shared-variable values is identical in the original and the projected computations. As the automaton checks properties defined only over shared variables, its accepting run carries over to the projected computation. In the projected computation, there are infinitely many positions where there is a transition by P_i from an accepting automaton state. Hence, the check in Step 3 fails for process P_i^θ. □

4 Fairness

We describe the modifications necessary to incorporate fairness assumptions into the local liveness method. We begin with a simple but useful kind of fairness, called *unconditional fairness*.

4.1 Unconditional Fairness

This fairness notion is a foundational concept in the UNITY programming language and proof system [3], and it suffices for many interesting distributed protocols. Under unconditional fairness, every process is scheduled infinitely often in an infinite computation. The statement uses "scheduled" rather than "executed"—a process may be scheduled but do nothing (i.e., behave as *skip*) because its transition is not enabled. To analyze a protocol under unconditional fairness, Step 3 of the local liveness method is modified to check that, for each P_i^θ, *there is no unconditionally fair computation where infinitely often there is a process i transition from a Büchi accepting state*.

Theorem 3. *The* LL *method modified for unconditional fairness is sound.*

Proof Sketch. The proof sketch for Theorem 2 shows that the sequence of process identifiers associated with the transitions is identical in the original and the projected computations. As the original error computation is unconditionally fair by assumption, the projected error computation must also be unconditionally fair. This argument shows that the modified check is sound. □

4.2 Strong Fairness

The strong fairness algorithm is based on iterated refinement. The idea is to start with a weakened form of the strong fairness assumption, and use the refinement mechanism which adds auxiliary variables to strengthen this assumption with each iteration, until a conclusive result is obtained. To keep the notation simple, we consider a common form of strong fairness, given as $\Phi \equiv (\bigwedge i : \mathsf{FG}(p_i) \vee \mathsf{GF}(q_i))$, where p_i and q_i are assertions over the variables of process P_i. Recall that the proof of soundness of the

local liveness method projects a global counter-example, σ, on to a local computation of abstract process P_k^θ, for some k. In the presence of fairness, there are two key properties of σ:

1. Every state on σ satisfies $\bigwedge_i \theta_i$, as θ is a split invariant, and
2. σ satisfies the fairness assumption Φ

Taken together, this implies—crucially—that σ must also satisfy the *stronger* fairness assertion, Φ^*, given by $(\bigwedge i : \mathsf{FG}(\theta_i \wedge p_i) \vee \mathsf{GF}(\theta_i \wedge q_i))$. The fact that Φ^* is stronger than Φ for any θ follows from the monotonicity of G and F. The fact that Φ^* holds for σ follows by the first property: as every state on σ satisfies $(\bigwedge j : \theta_j)$, assertions $\mathsf{FG}(\theta_i \wedge p_i)$ and $\mathsf{FG}(p_i)$ are equivalent on σ, as are assertions $\mathsf{GF}(\theta_i \wedge q_i)$ and $\mathsf{GF}(q_i)$.

The abstract fairness property is formed by quantifying out local variables from Φ^*, as follows.

$$\Phi^\theta = (\bigwedge i : \mathsf{FG}((\exists L_i : \theta_i \wedge p_i)) \vee \mathsf{GF}((\exists L_i : \theta_i \wedge q_i)))$$

Subsequently, we refer to the term $(\exists L_i : \theta_i \wedge p_i)$ as p_i^θ and to $(\exists L_i : \theta_i \wedge q_i)$ as q_i^θ. The transformed fairness property is weaker than Φ^*, but not necessarily weaker than Φ, and it is defined over the shared variables only.

It is important that Φ^θ depends on θ, and does so in a monotonic manner. This enables refinement: as the split invariant is strengthened by adding auxiliary variables, the abstract fairness assumption also becomes stronger. The new method is shown in Figure 5; other than a modified check at Step 3, it is identical to the LL method from Figure 4.

Theorem 4. *The FLL method is sound.*

Proof. This proof is an extension of the proof of Theorem 2. Consider a global counter-example σ which is fair according to Φ. By the proof of Theorem 2, the projection of σ on P_i^θ satisfies the second part of the condition of Step 3: i.e., infinitely often there is a process i transition from a Büchi accepting state. It remains to be shown that the projected computation also satisfies Φ^θ.

As σ is a counter-example based on the fairness assumption, it satisfies Φ; as it is a program computation, it satisfies the split invariant, θ. Hence, by the reasoning above, it satisfies Φ^* and therefore the weaker property Φ^θ. As Φ^θ is a property over shared state only, and the sequence of values for shared variables is preserved by the projection, Φ^θ holds also of the projected computation. □

Fair Local Liveness (FLL) Algorithm

1. Compute the strongest split invariant, θ.
2. For each i: build an abstract form of process i, P_i^θ, as defined in Figure 4
3. Form a Büchi automaton for the *negation* of the specification. For each i, form the synchronous product of this automaton with P_i^θ and check that *there is no computation which is strongly fair according to Φ^θ and on which infinitely often there is a process i transition from a Büchi accepting state*
4. Declare success if the check succeeds **for each** abstract process

Fig. 5. Fair Local Liveness (FLL) Algorithm

Refinement for Fair Local Liveness

1. Check if every summary transition in the abstract counter-example σ is a MUST transition for the process which makes it. If not, expose a local predicate for the MUST condition, as defined in [9] for the LL method, and REPEAT the full verification.
2. Inductively construct a global computation δ which matches σ
3. Check if δ satisfies the original fairness condition, Φ. If so, HALT with δ as the valid global counter-example.
4. Use a fairness term $(\mathsf{FG}(p_j) \vee \mathsf{GF}(q_j))$ which is *not* satisfied by δ to discover and expose a local predicate, and REPEAT full verification.

Fig. 6. Refinement for the FLL method, given a counter-example σ in the abstract process P_i^θ

Remark 1. Our implementation uses a stronger abstraction of the fairness property. In Step 3 of the FLL algorithm, instead of the uniform assumption Φ^θ, the implementation uses a fairness assumption for P_i^θ where all terms from Φ are abstracted relative to θ as described above, *except* the term $(\mathsf{FG}(p_i) \vee \mathsf{GF}(q_i))$, which is used as is, since it refers only to variables of process P_i.

4.3 FLL Algorithm Variant

The basic FLL algorithm can be varied by changing Steps (2)-(4) as follows. The new combination checks whether for *some* i, *the abstract process* P_i^θ *satisfies the specification, assuming strong fairness according to* Φ^θ. We call this algorithm the B-variant of the FLL algorithm; the original is called the A-variant. Note that the correctness condition in FLL (B) is stricter than that for FLL (A); on the other hand, it suffices that one of the abstract processes satisfies the test. The justification is based on a proof similar to that of Theorem 4: if a global counter-example exists, its projection in P_i^θ fails the FLL (B) requirement, for *every* i. The contra-positive shows that it suffices for some i to satisfy the FLL (B) requirement for the program to be correct.

The two algorithms offer a trade-off. Due to the weaker correctness condition of FLL (A), this algorithm may prove correctness while FLL (B) does not, leading to extra refinements in the B-variant. On the other hand, for FLL (B), it suffices to check a single, fixed process (say, P_0^θ); this is potentially faster for programs with a large number of components.

4.4 Refinement for Fairness

As local reasoning is approximate, it is possible for the FLL method to fail even though the property is true of the whole program. One can analyze the failure, though, to suggest auxiliary Boolean variables which expose local state predicates, as shown in Figure 6, which extends the refinement procedure used for the LL method.

Step 1 is the refinement step for LL. Recall that a transition of σ in P_i^θ by a process P_k other than P_i can modify only the shared variables. A change of shared state from $X = a$ to $X' = b$ is considered a MUST transition if this change is possible no matter what the local state of process P_k may be, so long as it is consistent with θ_k. The

predicate $m(L_k) \equiv \theta_k(a, L_k) \wedge \neg(\exists L'_k : T_k(a, L_k, b, L'_k))$ expresses this succinctly: the transition from $X = a$ to $X' = b$ is a MUST transition if, and only if, m is unsatisfiable. If m is satisfiable, it is "exposed" by adding an auxiliary shared variable x_m. The constraint $x'_m \equiv m(L'_k)$ is added to the transition relation of P_k, and the constraint $x'_m \equiv x_m$ to that for all other processes. Together with the initialization of x_m to $m(L_k)$, these constraints maintain the global invariant $(x_m \equiv m)$.

Regarding Step 2, if each summary transition in σ is a MUST transition, it is possible to inductively construct a global computation δ which matches σ. The initial values for the local variables for processes other than P_i can be chosen arbitrarily, consistent with the initial condition. Inductively, the MUST property guarantees that a concrete transition can be found for each process making a summary move such that the change to the shared state is preserved. Although σ satisfies Φ^θ, it need not be the case that δ satisfies Φ. If Φ fails to hold on the computed δ (Step 3), the proof of Theorem 6 shows how a new predicate can be derived by analyzing this failure.

Theorem 5. *(Soundness for failures) If the FLL refinement procedure halts with failure, the trace is a valid counter-example under strong fairness.*

Proof. Follows from the reasoning given for Steps 2 and 3. □

Theorem 6. *(Finitary Completeness) The FLL procedure with refinement terminates for finite-state programs.*

Proof. It suffices to show that a new predicate—one that is not a Boolean combination of existing predicates—is added at each refinement step. Termination follows, as there is a finite number of distinct predicates. Theorems 4 and 5 show that each termination outcome is correct; thus, the method is complete. For Step 1, the fact that a new predicate is added was shown for the LL method in [9]. For the predicate added at Step 4, it can be shown as follows.

If the check at Step 3 fails, there is a term, $(\mathsf{FG}(p_j) \vee \mathsf{GF}(q_j))$, for some j, which fails to hold for δ. Thus, from some point on, all states on δ fail q_j, and infinitely often, there is a state failing p_j. Depending on which sub-term is used to satisfy $(\mathsf{FG}(p_j^\theta) \vee \mathsf{GF}(q_j^\theta))$ on σ, there is a state s that is on σ and its corresponding state t on δ such that either (i) s satisfies p_j^θ and t does not satisfy p_j or (ii) s satisfies q_j^θ while t does not satisfy q_j.

Consider the first case, the proof of the second is similar. By the definition of p_j^θ as $(\exists L_j : \theta_j \wedge p_j)$, there is a valuation c for L_j such that for $u = (s(X), c)$ it is the case that $\theta_j(u)$ and $p_j(u)$ both hold. On the other hand, while $\theta_j(t)$ holds by the invariance of θ for the concrete computation δ, $p_j(t)$ does not hold by the assumption. By the correspondence of s and t, states u and t differ only on the valuation of L_j. Let q be a predicate expressing this difference (e.g., $q(L_j) = (L_j = c)$). We have to show that q is a new predicate; i.e., it cannot be expressed as a function of the already exposed predicates.

A property of the split invariant, which can be shown by induction, is that $[\theta_j \Rightarrow (x_m \equiv m)]$ for each shared refinement variable x_m that is added for a predicate m exposed for process P_j. As u and t agree on all shared variables, including refinement variables, and as both satisfy θ_j, it follows that all prior predicates exposed for P_j have identical values on u and t. As this is not true for q, it cannot be expressed as a function of the already exposed predicates. □

4.5 Weak and Generalized Fairness

Weak Fairness, also called "justice", has the normal form $\mathsf{GF}(p)$ ("infinitely often p"). It is often used to express the constraint that a continuously enabled transition cannot be forever ignored; i.e., $\mathsf{FG}(enabled) \Rightarrow \mathsf{GF}(executed)$. As its normal form is a special case of strong fairness, the algorithm developed for strong fairness can be applied to it. Thus, the common weak fairness specification $\Phi \equiv (\bigwedge i : \mathsf{GF}(p_i))$, where p_i is an assertion over the variables of process P_i, is abstracted to $\Phi^\theta \equiv (\bigwedge i : \mathsf{GF}(\exists L_i : \theta_i \wedge p_i))$ for use in the FLL algorithm.

Emerson and Lei consider a general fairness criterion in [13], which is a disjunction of strong fairness conditions. This can be handled by abstracting each disjunct separately and re-forming the disjunction.

For simplicity, the development of the algorithm considered fairness assertions $(\bigwedge i : \mathsf{FG}(p_i) \vee \mathsf{GF}(q_i))$ where p_i and q_i are expressed in terms of the variables of process P_i. In a more general setting, these predicates may be expressed over the local state of more than one process. The analysis method extends easily, with each predicate being abstracted by quantifying out the relevant local variables. Thus, the general abstraction function is $p^\theta \equiv (\exists L : (\bigwedge i : \theta_i) \wedge p)$.

5 Experimental Results

We implemented our method as part of SPLIT [10] – a compositional LTL verifier, and tested it on several parameterized examples which require fairness assumptions. We also compared it with the LTL model checker implemented on top of JTLV [26], and with the model checker NUSMV [4]. The latter, however, is optimized for verifying synchronous systems and even after disabling the conjunctive partitioning the results obtained by it were considerably inferior to those obtained by JTLV and SPLIT. We therefore do not include in this paper the results obtained by NUSMV. The experiments were conducted on a Intel Core 2 Duo 2.4 GHz with 4 GB RAM running 64-bit Linux. Both SPLIT and JTLV were configured to use the CUDD BDD library. We set a timeout of 20 minutes for the experiments.

The experiments test the method on a number of well-known parameterized protocols. These protocols form a good set of benchmarks: they represent succinct models of standard synchronization patterns found in concurrent software; their characteristics (e.g., proof structure and complexity) are well known, making comparisons with other methods easy for the reader. While the descriptions are short, standard model checking

Table 1. Experimental results when assuming only unconditional fairness

	Example	Property	N	JTLV		SPLIT (A)			SPLIT (B)		
				Nodes	Time	Ref.	Nodes	Time	Ref.	Nodes	Time
1	BAKERY	*no-starvation*	3	300K	0.3	2	1.2M	2.5	2	0.9M	1.5
		– Valid –	4	11.6M	93	2	14.6M	52	2	7.4M	17.4
2	MUXSEM	*no-starvation*	5	58K	0.2	1	48K	0.3	1	44K	0.3
		– Invalid –	10	21M	24	2	371K	1.1	2	330K	1
			20	over 20 minutes		2	2.1M	9	2	1.9M	8.3

Table 2. Experimental results when assuming only strong fairness only over P[0]

	Example	Property	N	JTLV		SPLIT (A)			SPLIT (B)		
				Nodes	Time	Ref.	Nodes	Time	Ref.	Nodes	Time
3	MUXSEM	*no-starvation*	5	24K	0	1	61K	0.2	1	38K	0.2
		– Valid –	10	1.2M	3.8	1	259K	0.7	1	142K	0.5
			20	over 20 minutes		1	1.2M	3	1	697K	1.5

f: **array** $[0..N]$ **of boolean initially** $f = 1$

$$P[1] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} l_0\colon \textit{Non-Critical} \\ l_1\colon \textbf{request}\ f[2] \\ l_2\colon \textbf{request}\ f[1] \\ l_3\colon \textit{Critical} \\ l_4\colon \textbf{release}\ f[1] \\ l_5\colon \textbf{release}\ f[2] \end{array}\right] \end{array}\right] \quad \underset{i=2}{\overset{N}{\|}} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} l_0\colon \textit{Non-Critical} \\ l_1\colon \textbf{request}\ f[i] \\ l_2\colon \textbf{request}\ f[i \oplus_N 1] \\ l_3\colon \textit{Critical} \\ l_4\colon \textbf{release}\ f[i \oplus_N 1] \\ l_5\colon \textbf{release}\ f[i] \end{array}\right] \end{array}\right]$$

Fig. 7. Program DINING-PHIL: the dining philosophers

is by no means proportionally easy, as shown by the time-outs in experiments. Both variants (A and B) of the FLL compositional algorithm are examined. In our experiments, variant B has the better performance.

As mentioned in Subsection 4.1, unconditional fairness is sufficient to guarantee various properties in selected protocols. For example, in algorithm BAKERY [18] ensuring individual starvation-freedom, i.e., $\forall i : \mathsf{G}(wait(i) \Rightarrow \mathsf{F}(crit(i)))$, does not require to assume any weak or strong fairness conditions. For other protocols, such as MUXSEM, the same property is not valid when assuming only unconditional fairness, and both JTLV and SPLIT generate valid counter examples when attempting to verify it. The results for checking the eventual access property of $P[1]$ for the two protocols are provided in Table 1. Note that since the property should be over global variables, the location variable of $P[1]$ was exposed to all processes. "N" is the number of processes, "Nodes" is the peak number of BDD nodes generated, "Time" is the run time in seconds, and "Ref." is number of refinements had to be executed by SPLIT. For both examples the run-times are better for SPLIT; for MUXSEM, where counter examples had to be constructed, they are better by several orders of magnitude. Both SPLIT and JTLV required more than 20 minutes for verifying BAKERY for $N = 5$.

Assuming the strong fairness $\mathsf{GF}(P[1].at\_loc_1 \wedge x) \Rightarrow \mathsf{GF}(P[1].at\_loc_2)$ *only* for $P[1]$ is sufficient to prove the correctness of $\mathsf{G}(wait(1) \Rightarrow \mathsf{F}(crit(1)))$ for MUXSEM. Both model checkers indeed validated the property under this condition and the results are provided in Table 2; they are again in favor of our method by a few orders of magnitude.

Most interesting and challenging test cases with respect to fairness are those that require to assume weak or strong fairness conditions for *all* the processes. The first such example is DINING-PHIL (a simple solution to the dining philosophers problem using semaphores), presented in Fig. 7. The eventual access property is valid only when

Table 3. Results for properties that require to assume general fairness over *all* processes

	Example	Property	N	JTLV		SPLIT (A)			SPLIT (B)		
				Nodes	Time	Ref.	Nodes	Time	Ref.	Nodes	Time
4	DINING-PHIL	*no-starvation* – Valid –	8	3M	13	0	1.9M	4	0	1.2M	1.8
			9	9.1M	63	0	4.1M	8.6	0	2.4M	4.3
			10	25M	421	0	8.6M	18	0	5.3M	9.9
5	COND-TERM	*termination* – Valid –	4	91K	0.4	3	389K	1	2	299K	0.8
			6	537K	1.6	3	2.1M	6.7	2	1.6M	5.1
			8	4M	10	3	19M	101	2	11M	75.4
6	MUXSEM-NON-DET	*no-starvation* – Valid –	8	262K	0.6	1	172K	0.5	1	96K	0.4
			12	5.3M	32.6	1	393K	1	1	210K	0.5
			16	over 20 minutes		1	720K	1.8	1	385K	0.9

assuming that $\mathsf{GF}(P[i].at\_loc_1 \wedge f[i]) \Rightarrow \mathsf{GF}(P[i].at\_loc_2)$ and $\mathsf{GF}(P[i].at\_loc_2 \wedge f[i \oplus_N 1]) \Rightarrow \mathsf{GF}(P[i].at\_loc_3)$ for $1 < i \leq N$ and assuming the symmetric conditions for $i = 1$. Namely, for each philosopher, if she can (enabled) infinitely often pick the first fork and subsequently pick the second fork then she should eat Spaghetti infinitely often. Example 4 in Table 3 presents the run-time results for verifying this example, that are again in favor of our method.

$$y\colon \textbf{0..M where } y = M$$
$$\Big\|_{i=1}^{N} P[i] :: \left[\begin{array}{l} l_0\colon \textbf{while } y \neq 0 \textbf{ do} \\ \quad \left[\begin{array}{l} l_2\colon y := y - 1 \\ \quad \textbf{or} \\ l_3\colon y := min(y+1, M) \end{array}\right] \\ l_1\colon \\ : \ l_4 \end{array}\right]$$

Fig. 8. COND-TERM

The second example is proving termination of COND-TERM. The protocol is presented in Fig. 8. A process terminates only if the strong fairness condition $\mathsf{GF}(P[i].at\_l_3) \Rightarrow \mathsf{GF}(false)$ is assumed over *all* processes. This condition permits only computations where y is increased a finite number of times. We verified the termination of COND-TERM for $M = 15$. The results are provided as example 5 in Table 3. This time they are in favor of the monolithic model checking as SPLIT requires a number of refinements to prove the property.

The last example that requires to assume general fairness over all the processes is MUX-SEM-NON-DET, presented in figure Fig. 9. This example is a variation of MUXSEM that allows each of the processes to stay non-deterministically, possibly forever, in the critical section. Thus, $\mathsf{G}(wait(1) \Rightarrow \mathsf{F}(crit(1)))$ is valid only when assuming $\bigwedge i : \mathsf{GF}(P[i].at\_l_2) \Rightarrow \mathsf{GF}(P[i].at\_l_3)$. Namely, for each process, if it can (enabled) infinitely often leave the critical section then it should leave it infinitely often. Example 6 in Table 3 presents the run-time results for verifying this example, that are again in favor of our method.

$$x\colon \textbf{boolean where } x = 1$$
$$\Big\|_{i=1}^{N} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} l_0\colon \textit{Non-Critical} \\ l_1\colon \textbf{request } x \\ l_2\colon \langle \textit{Critical};\ \textbf{await } (\textit{false}) \textbf{ or skip}\rangle \\ l_3\colon \textbf{release } x \end{array}\right] \end{array}\right]$$

Fig. 9. MUX-SEM-NON-DET: mutual exclusion with a non-deterministic stay in critical section

1. Find a vector of local assertions, $\theta = (\theta_1, \ldots, \theta_N)$, which meets the split invariance conditions from Figure 1
2. Form a fairness assertion, Ξ, out of the abstract assertions in Φ^θ and the acceptance condition of the Büchi automaton for the negated property. For each i, instantiate the strong fairness proof rule of [25] for the synchronous composition of the automaton and the abstract process P_i^θ, with the fairness assertion Ξ and specification $\mathsf{G}(\mathit{true} \Rightarrow \mathsf{F}\mathit{false})$.

Fig. 10. Local Proof Rule for LTL properties

6 Deductive Compositional Proofs under Fairness

The LL method was derived in [9] from a proof rule for verifying linear-time properties expressed by a Büchi automaton for their negation. That proof rule has two parts: the first part expresses that θ is a split invariant, while the second part shows that a Büchi accepting state occurs only finitely often on any joint computation of the program and the automaton, using rank functions which are local to each process.

This structure can be modified to accommodate fairness, as shown in Figure 10. The proof rule of [25] is used with the conclusion being false. A valid proof shows the absence of any joint computation which is fair and is an accepting Büchi automaton run. All assertions and rank functions are local by definition. Moreover, as shown in [25], one can generate these components by instrumenting the model checking algorithms used in FLL.

7 Conclusions and Future Work

The algorithm presented here enables fully automated and compositional verification of progress properties under fairness and is, we believe, the first algorithm to do so. It deals with the main difficulty, that of handling the global nature of fairness, by a process of refinement: the fairness assumption is initially weakened relative to a split invariant, and is then strengthened in subsequent iterations until a decisive result is obtained. The algorithm has a simple implementation. Experiments with several parameterized protocols show a clear advantage for the compositional method over the standard non-compositional one.

One aspect that merits further exploration is the choice of counter-example trace for refinement; currently, the algorithm uses whichever trace is provided by the model checking procedure. It would help, for instance, if the trace generation is biased to generate a trace which satisfies as many MUST requirements as possible.

References

1. Alur, R., Henzinger, T.: Reactive modules. In: IEEE LICS (1996)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
3. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
4. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
5. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic. ACM Transactions on Programming Languages and Systems (TOPLAS) 8(2) (1986)
6. Clarke, E.M., Grumberg, O.: Avoiding the state explosion problem in temporal logic model checking. In: PODC, pp. 294–303 (1987)
7. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In: ISSTA, pp. 97–108 (2006)
8. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 55–67. Springer, Heidelberg (2007)
9. Cohen, A., Namjoshi, K.S.: Local proofs for linear-time properties of concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 149–161. Springer, Heidelberg (2008)
10. Cohen, A., Namjoshi, K.S., Sa'ar, Y.: SPLIT: A Compositional LTL Verifier. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 558–561. Springer, Heidelberg (2010)
11. de Roever, W.-P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods. Cambridge University Press, Cambridge (2001)
12. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, Heidelberg (1990)
13. Emerson, E.A., Lei, C.-L.: Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In: LICS (1986)
14. Emerson, E.A., Lei, C.-L.: Modalities for model checking: Branching time logic strikes back. Sci. of Comp. Programming 8(3) (1987)
15. Farzan, A., Chen, Y., Clarke, E.M., Tsan, Y., Wang, B.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008)
16. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
17. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: ASE, pp. 3–12 (2002)
18. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. ACM Commun. 17(8), 453–455 (1974)
19. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. 3(2) (1977)
20. McMillan, K.L.: A compositional rule for hardware design refinement. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 24–35. Springer, Heidelberg (1997)
21. McMillan, K.L.: Circular compositional reasoning about liveness. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 342–346. Springer, Heidelberg (1999)

22. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 299–313. Springer, Heidelberg (2007)
23. Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 139–153. Springer, Heidelberg (2000)
24. Owicki, S.S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. ACM Commun. 19(5), 279–285 (1976)
25. Pnueli, A., Sa'ar, Y.: All you need is compassion. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 233–247. Springer, Heidelberg (2008)
26. Pnueli, A., Sa'ar, Y., Zuck, L.D.: JTLV: A framework for developing verification algorithms. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 171–174. Springer, Heidelberg (2010), http://jtlv.ysaar.net/
27. Tkachuk, O., Dwyer, M.B., Pasareanu, C.S.: Automated environment generation for software model checking. In: ASE, pp. 116–129 (2003)