Reasoning about GSTE Assertion Graphs

Alan J. Hu[1], Jeremy Casas[2], and Jin Yang[2]

[1] Department of Computer Science, University of British Columbia, ajh@cs.ubc.ca,
2366 Main Mall, Vancouver, BC V6T 1Z4, Canada, +1-604-822-6667, FAX +1-604-822-5485
[2] Strategic CAD Labs, Intel Corporation

Abstract. *Generalized symbolic trajectory evaluation* (GSTE) is a new model-checking approach that combines the industrially-proven scalability and capacity of classical symbolic trajectory evaluation with the expressive power of temporal-logic model checking. GSTE was originally developed at Intel and has been used successfully on Intel's next-generation microprocessors. However, the supporting theory and algorithms for GSTE are still immature. In particular, GSTE specifications are given as *assertion graphs*, a variety of $\forall$-automata, and although an efficient model-checking algorithm exists to verify whether a circuit model obeys a specification assertion graph, there is no work on reasoning about assertion graphs themselves. This paper presents new algorithms to leverage GSTE model checking to efficiently decide whether one assertion graph implies another, and to model check one assertion graph under the assumption that another is true (under regular GSTE acceptance conditions). These two operations — deciding whether one specification implies another and verifying under an assumption — are the fundamental building blocks of compositional verification and any higher-level reasoning about model-checking results, so the algorithms presented here are key steps to using GSTE in a broader verification framework. Preliminary experimental results applying our algorithms to real, industrial circuits and specifications show that our algorithms are useful in practice.

1 Introduction

Generalized symbolic trajectory evaluation (GSTE) is a powerful, new model-checking approach [20]. GSTE is based on classical symbolic trajectory evaluation [16], which has proven itself able to handle large, industrial designs and has been in active use at Compaq (now HP), IBM, Intel, and Motorola (e.g., [12, 10, 1, 4]). Classical symbolic trajectory evaluation, although efficient, is very limited in the types of properties that it can specify and verify. GSTE extends classical symbolic trajectory evaluation to handle ω-regular properties, giving it comparable expressive power to more established model-checking approaches [5, 13, 18, 8, 6], while still maintaining the efficiency and capacity of classical symbolic trajectory evaluation. GSTE was originally developed at Intel and has been used successfully on Intel's next-generation microprocessors (e.g., [3]).

Key to the efficiency and usability of GSTE is the manner in which properties are specified, in a variety of automata called an *assertion graph*. Existing GSTE theory provides an efficient procedure for model checking that a circuit obeys an assertion graph, as well as techniques based on abstract interpretation to combat state explosion [21].

What is missing, however, is all the supporting theory and algorithms that have developed around more established formalisms like CTL [5] or LTL [18]. In particular, there has been no published research on how to reason about assertion graphs.

This paper presents the foundational pieces for reasoning about specifications given as assertion graphs. Specifically, we give new algorithms to decide whether one assertion graph implies another, and to model check one assertion graph under the assumption that another is true. These two operations — deciding whether one specification implies another and verifying under an assumption — are the fundamental building blocks for decomposing a verification task, composing verification results, and any other higher-level reasoning about specifications. Our current verification system is a mixed deductive-algorithmic system, with an efficient GSTE model-checking procedure built into a lightweight theorem prover. Our new algorithms exploit the existing GSTE model-checking procedure, creating an efficient, algorithmic means to discharge basic deductive reasoning steps about assertion graphs. Preliminary experimental results on real, industrial circuits and specifications show that the algorithms are efficient in practice.

2 Background

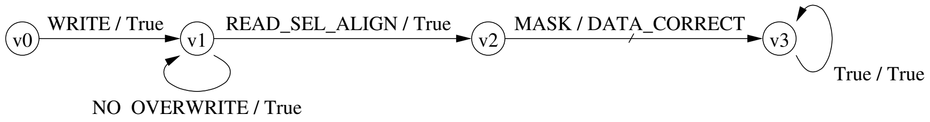
2.1 GSTE and Assertion Graphs

GSTE is explained in several sources (e.g., [20, 21, 19], etc.). Here, we concentrate on the specification style used by GSTE and highlight its characteristics.

GSTE is basically a linear-time model-checking method, i.e., the possible behaviors of the system being verified is considered to be the set of all possible execution traces, and verification consists of checking that all of these traces obey the specification. The specification in GSTE is called an *assertion graph*, and is basically a variety of automaton. One can think of the assertion graph as defining the set of execution traces that it accepts, so the verification problem is basically language containment. Figure 1 gives a simple example and intuitive explanation of an assertion graph.

In general, an assertion graph is a directed graph with distinguished initial vertex $v0$, and the restriction that all vertices must have non-zero out-degree. Each edge e is labeled with an antecedent $ant(e)$ and a consequent $cons(e)$. The antecedents and consequents are simply propositional formulas over some set of atomic propositions AP. Traditionally, the atomic propositions correspond exactly to the state variables of the system being verified, so the antecedents and consequents are formulas over the state of the system at some point in time. The assertion graph also has *acceptance conditions*, described below.

A *path* in the assertion graph is a directed path (defined in the usual manner for directed graphs) starting from the initial vertex $v0$. Every path in the assertion graph specifies a temporal if-then assertion: if the antecedents hold, then the consequents must hold as well. More precisely, a path of length n (i.e., with n edges) is an assertion about the system's behavior over a period of n clock cycles. If all of the antecedents along the path hold at the corresponding points in the system's behavior, then all of the consequents must also hold at the corresponding points, in order for the assertion to be



$$
\begin{aligned}
\text{WRITE} &:= (\text{we} = 1) \wedge (\text{addr} = A) \wedge (\text{datawr} = D) \\
\text{NO\_OVERWRITE} &:= (\text{we} = 0) \vee (\text{addr} \neq A) \\
\text{READ\_SEL\_ALIGN} &:= (\text{ck} = 0) \wedge (\text{we} = 0) \wedge (\text{addr} = A) \wedge (\text{sel} = S) \wedge (\text{align} = R) \\
\text{MASK} &:= (\text{ck} = 1) \wedge (\text{maskbegin} = B) \wedge (\text{maskend} = E) \\
\text{DATA\_CORRECT} &:= (\text{dataout} = \text{mask}(\text{align}(\text{select}(D,S),R),B,E))
\end{aligned}
$$

Fig. 1. GSTE Assertion Graph Example. This assertion graph, adapted from [20], was used in the verification of an industrial memory design, which reads and writes data with a large variety of selection and alignment options. The property being verified is that, if data value D is written to address A, followed by an arbitrary number of clock cycles that don't overwrite the same address, followed by a read of the address, then the value returned is the value that was written, appropriately aligned and masked. The edge labels are of the form "*antecedent / consequent*", where the antecedents and consequents are simply propositional formulas over the state of the system at a given clock cycle. For example, the antecedent WRITE specifies that the value of the write-enable input `we` is high, that the address input `addr` is equal to some value A, etc. The capital letters denoting values, like A, D, etc., are *symbolic constants*, which are essentially skolem constants that can be equal to any value, making the verification result hold for all possible values of the symbolic constants. A path is a sequence of edges that start from the initial vertex $v0$. A terminal path is a path that ends with a terminal edge (shown in the figure by a tic-mark on the edge, e.g., the edge from $v2$ to $v3$). A path accepts an execution trace if at least one antecedent on that path fails (is false on the state of the system at that clock cycle) or if all antecedents and all consequents on the path succeed (are true on that clock cycle). Intuitively, a path is an if-then assertion: the antecedents say when the assertion is relevant; the consequents say what must hold whenever the assertion is relevant. If any antecedent fails, the assertion is vacuously true; if all antecedents are satisfied, then all consequents must be satisfied as well. The assertion graph as a whole accepts an execution trace if **every** terminal path in the assertion graph accepts that trace. Intuitively, the assertion graph takes a potentially infinite set of assertions about the system and rolls them up into a graph; therefore, every trace must satisfy every assertion (vacuously or otherwise).

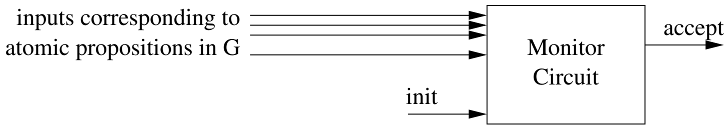


Fig. 2. Monitor Circuit. Our algorithms rely on a linear-space, linear-time construction for a monitor circuit from an assertion graph G. The generated circuit has inputs corresponding to the atomic propositions in G and an output that is true iff the sequence of states presented at the input would have been accepted by G. The `init` input initializes the internal state of the circuit.

satisfied. If any antecedent doesn't hold, then the assertion is vacuously true. Formally, if ρ is a path of length n, with $\rho[i]$ denoting the ith edge in ρ, and if σ is a trace consisting of n system states, with $\sigma[i]$ denoting the ith state, then σ *satisfies* or *is accepted* by ρ iff

$$(\forall i_{1 \le i \le n}.\ \sigma_i \models ant(\rho[i])) \ \Rightarrow\ (\forall i_{1 \le i \le n}.\ \sigma_i \models cons(\rho[i]))\,.$$

For convenience, we will say that "σ satisfies the antecedents of ρ" if $\forall i_{1 \le i \le n}.\ \sigma_i \models ant(\rho[i])$, and that "$\sigma$ fails at least one of the consequents of ρ" if $\exists i_{1 \le i \le n}.\ \sigma_i \not\models cons(\rho[i])$.

An assertion graph as a whole accepts a given trace iff **all** "appropriate" paths in the assertion graph are satisfied. Appropriate is defined by the four different kinds of acceptance in GSTE:

- In *strong satisfiability*, a finite-length trace is accepted iff it satisfies all paths of the same length in the assertion graph.
- In *terminal satisfiability*, some edges are marked as *terminal edges*, and a *terminal path* is a path that starts from $v0$ and ends with a terminal edge. A finite-length trace is accepted iff it satisfies all terminal paths of the same length.
- In *normal satisfiability*, an infinite trace is accepted iff it satisfies all infinite paths.
- In *fair satisfiability*, there is a finite set of *fair edge sets*. A path is fair iff it visits each fair edge set infinitely often (generalized Büchi fairness). An infinite trace is accepted iff it satisfies all fair paths.

The different kinds of acceptance are listed in (roughly) increasing order of model-checking complexity.

An assertion graph G defines the set of traces that it accepts. Call that set the language of G, denoted $L(G)$. Similarly, a system M defines the set of traces that it can produce, denoted $L(M)$. Verification consists of proving that $L(M) \subseteq L(G)$. In subsequent sections of this paper, unless otherwise stated, we will restrict ourselves to terminal satisfiability, which includes strong satisfiability as a special case, because the finite-trace satisfiabilities are currently the most commonly used in practice.

At first glance, assertion graphs may appear somewhat bizarre: the antecedent/consequent edge labels are unusual, as is acceptance based on **all** paths accepting. However, assertion graphs are actually the natural combination of symbolic trajectory evaluation and automata-theoretic specification. The antecedent/consequent style comes from classical symbolic trajectory evaluation [16] and is a natural way to specify temporal properties. For example, timing diagrams, one of the most widely used hardware specifications in practice, are typically interpreted this way (e.g., if some sequence of events happens, then some other events must happen) [2]. In addition, the explicit identification of antecedents and consequents provides an efficiency benefit, because the model-checking algorithm can limit its search on-the-fly to paths that satisfy the antecedents. The "for all paths" acceptance criteria makes assertion graphs a variety of $\forall$-automata [9], which are less familiar than the usual existential acceptance of nondeterministic automata (where a trace is accepted if there exists a corresponding path through the automata), but the $\forall$ semantics also provides both usability and efficiency benefits. The usability arises because an assertion graph defines a set of **assertions**, and one typically wants all assertions to be true; in contrast, usually with automata as specifications, the automata directly defines a set of possible **behaviors**, so verification consists of determining if the system's behavior exists in the set provided by the specification. The efficiency advantage of the $\forall$ semantics — as in other works that use $\forall$-automata as specifications [9, 8, 2] — is that a $\forall$-automaton is essentially pre-complemented, so checking language containment can bypass the expensive step of complementing a non-deterministic automaton. Indeed, GSTE model-checking is very efficient in practice, and the correctness of the algorithm relies on the $\forall$ semantics.

We emphasize that assertion graphs take their present form as the direct result of practical considerations. The natural theoretical question is what relationship they have to more established formalisms. Assertion graphs with fairness can express all ω-regular properties: an easy construction is to start with a non-deterministic, generalized Büchi automata and then to note that the almost-isomorphic assertion graph (with the same structure, the same fairness constraints, the Büchi automaton's edge labels moved to the antecedents, and all consequents labeled with *False*) accepts the complement language. ω-regular expressiveness follows because ω-regular languages are closed under complementation. The same construction also shows that non-deterministic Büchi automata can be simulated with a single-exponential blow-up (to pre-complement the Büchi automaton), and that LTL model checking can be translated to GSTE with at worst the same complexity as the translation to generalized Büchi automata, for which efficient tools exist (e.g., [17]). In the other direction, assertion graphs can be simulated by more conventional automata.[3] Analogous results hold for assertion graphs with terminal satisfiability and ordinary regular automata. In theory, therefore, assertion graphs are no more expressive.

In our case, we have an existing user community with practical experience using GSTE assertion graphs as well as an industrially-proven, efficient GSTE model-checking tool. The short-term need was for algorithms for rudimentary reasoning with assertion graphs — implication and model-checking under assumptions — so we sought to develop efficient algorithms to perform these operations directly on assertion graphs (with terminal satisfiability), exploiting the existing GSTE model-checking engine as much as possible.

2.2 Monitor Circuits from Assertion Graphs

Our algorithms for reasoning about assertion graphs rely on an efficient (linear space and time) algorithm for constructing circuits from assertion graphs, which was inspired by efficient methods for generating circuits from regular expressions [15, 14, 11]. The construction is rather intricate and is described elsewhere [7]. Here, we give a brief overview.

Given an assertion graph G, we construct a monitor circuit for G. A monitor circuit is simply a small circuit that watches, without interfering, the system being verified and

[3] Simulation by a conventionally labeled $\forall$-automata can be done with twice as many states; simulation by a normal $\exists$-automata requires an exponential blow-up. We would like to thank the anonymous reviewers for suggesting the construction for simulation via conventional $\forall$-automata, and for pointing out that there cannot be a general sub-exponential construction to simulate assertion graphs via normal $\exists$-automata or vice-versa.

flags whether or not the system is obeying some user-specified correctness property. In this case, the monitor circuit has inputs corresponding to the atomic propositions AP that are used in G. The monitor circuit has a single output `accept`, which is true iff the trace that has been observed on the inputs would be accepted by G. The circuit is a Mealy machine, so the value at the inputs is immediately reflected at the `accept` output. The circuit also has an `init` input, which initializes the internal state of the circuit; `init` is asserted at the same time that the first state of the execution trace is presented at the inputs, and then de-asserted from then on. See Figure 2.

Intuitively, the monitor circuit has an internal copy of the assertion graph and keeps track of paths by placing tokens on the edges in its copy. In theory, each token represents a path that ends on that edge at that clock cycle, and the token remembers the history of which antecedents and consequents were true during preceding clock cycles. At each clock cycle, tokens can update their histories and advance to the next edge, possibly splitting into multiple tokens if there are multiple out-going edges. The monitor accepts a trace iff all tokens represent accepting paths. The key insight to making this construction efficient is that the tokens can actually be almost memoryless. The only history necessary is to distinguish between three different kinds of pasts: (1) if an antecedent has failed already, this path and its continuations will always accept, so they need not be tracked any further, (2) if all antecedents and all consequents so far have succeeded, then this path currently accepts, but its continuations might not, and (3) if all antecedents have succeeded, but at least one consequent has failed, then this path currently rejects, but its continuations might eventually accept if an antecedent fails in the future. All paths with the same history that arrive at the same edge at the same time will share the same future, so their tokens can be merged. Hence, the constructed monitor circuit has a structure that exactly corresponds to the assertion graph, with two state bits per edge to track the two kinds of tokens, and a constant amount of circuitry per edge and per vertex to update the tokens appropriately. The constructed circuit is clearly linear-size compared to G.

3 Assertion Graph Implication

We now consider determining whether one assertion graph G_1 implies another assertion graph G_2, or, equivalently, whether $L(G_1) \subseteq L(G_2)$.

3.1 Implication via Product Construction

The monitor circuit construction immediately yields an obvious way to determine whether $L(G_1) \subseteq L(G_2)$:

1. Build circuits C_1 and C_2 for the assertion graphs G_1 and G_2.
2. Tie the inputs together.
3. Verify on the combined machine, using GSTE or any other model checking method, whether $\texttt{accept}_1 \Rightarrow \texttt{accept}_2$ in all reachable states.

The disadvantage of this approach is that we are building circuits for both G_1 and G_2, rather than using G_2 as a specification, potentially increasing the possibility of state explosion. Instead, we would like to harness the efficiency of GSTE and avoid adding G_2 to the state space.

3.2 Implication via GSTE

Given a circuit M and an assertion graph G, GSTE model checking provides an efficient way to determine whether $L(M) \subseteq L(G)$, or equivalently, whether M is a model of G, notated $M \models_T G$. (The "T" is for terminal satisfiability.) With our construction of a circuit from an assertion graph, one might consider generating a circuit C_1 from assertion graph G_1, and then determining whether $G_1 \Rightarrow G_2$ by model checking whether $C_1 \models_T G_2$. Unfortunately, this approach does not work: C_1 is a monitor circuit that indicates whether or not an input stream was an accepting trace; it is not a circuit whose behaviors are exactly the accepting traces. A more subtle approach is needed.

The idea behind our algorithm is to modify G_2 so that it ignores traces that are not accepted by C_1. More precisely, given assertion graphs G_1 and G_2, we determine whether $G_1 \Rightarrow G_2$ as follows:

1. Without loss of generality, we assume that the initial vertex $v0$ of G_2 has in-degree of 0. (If this is not the case, we can modify G_2 by creating a duplicate initial vertex $v0'$, which has the same incoming and outgoing edges as $v0$, and then we delete the incoming edges to the true initial vertex $v0$.)
2. Apply the monitor circuit construction to G_1, resulting in circuit C_1.
3. Modify G_2 to work with C_1, creating a new assertion graph G_2':
   (a) The new graph G_2' has all of the same vertices as G_2.
   (b) For every edge e in G_2 from vertex v_i to vertex v_j, create two edges e' and e'', both from vertex v_i to vertex v_j. Set

$$ant(e') = \begin{cases} ant(e) \wedge \texttt{accept} \wedge \texttt{init} & \text{if } v_i = v0 \\ ant(e) \wedge \texttt{accept} \wedge \neg\texttt{init} & \text{otherwise} \end{cases}$$

$$ant(e'') = \begin{cases} ant(e) \wedge \neg\texttt{accept} \wedge \texttt{init} & \text{if } v_i = v0 \\ ant(e) \wedge \neg\texttt{accept} \wedge \neg\texttt{init} & \text{otherwise} \end{cases}$$

   The consequents do not change: $cons(e) = cons(e') = cons(e'')$. Edge e' is a terminal edge in G_2' iff edge e is a terminal edge in G_2. Edge e'' is not a terminal edge.
   (c) Add `init` and `accept` to the atomic proposition set.

   Figure 3 shows this construction applied to the assertion graph from Figure 1.
4. Use GSTE to model check whether $C_1 \models_T G_2'$. The result is true iff $G_1 \Rightarrow G_2$.

Proof that $G_1 \Rightarrow G_2$ implies $C_1 \models_T G_2'$:
Suppose $C_1 \not\models_T G_2'$. Then, there exists a trace σ' of C_1 and a terminal path ρ' of G_2', of the same length, where σ' satisfies all the antecedents in ρ', but fails at least one consequent. Define the trace σ by projecting out the `accept` and `init` signals from each state of σ'. Define path ρ in G_2 formed from ρ' by mapping back through the edge doubling. We prove that σ is a witness that $G_1 \not\Rightarrow G_2$ by showing that:

1. $\sigma \models_T G_1$.
2. ρ is a terminal path in G_2.
3. σ satisfies the antecedents along ρ.
4. σ fails at least one consequent along ρ.

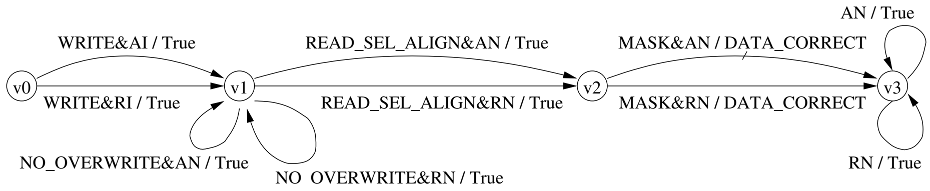


Fig. 3. Assertion Graph Modified to Consider Only Accepting Paths. This figure shows the result of modifying the assertion graph in Figure 1 using the construction from Section 3.2. Edge labels are defined as in Figure 1, with AI := $\texttt{accept} \wedge \texttt{init}$, AN := $\texttt{accept} \wedge \neg\texttt{init}$, RI := $\neg\texttt{accept} \wedge \texttt{init}$, and RN := $\neg\texttt{accept} \wedge \neg\texttt{init}$. The implication construction modifies an assertion graph so that it considers only the accepting paths of the other assertion graph. The basic idea is to double all edges, with one edge guessing that the path is accepting and the other edge guessing that the path is rejecting. Because these guesses are in the antecedents, paths that guess wrong are disregarded. The modification also ensures that the monitor circuit is initialized properly, via the `init` signal.

Claim 1: We know that σ' satisfies the antecedents of ρ'. Therefore, the circuit C_1 is initialized properly, because the antecedent constrain the `init` signal. Also, the `accept` signal is true in the last state of σ', because ρ' ends on a terminal edge, so σ is an input sequence that would end up with C_1 accepting. Therefore, $\sigma \models_T G_1$, by the construction of C_1.

Claim 2: G_2' is created by doubling the edges of G_2. Undoing the doubling maps the path back to a path on G_2. Since ρ' ended on a terminal edge in G_2', the corresponding edge in G_2 must also be a terminal edge, so ρ is a terminal path.

Claim 3: Recall that σ' satisfies the antecedents of ρ'. The path ρ has antecedents that are strictly weaker than the corresponding antecedents in ρ', because they are missing the conjuncts about `accept` and `init`. Therefore, σ satisfies the antecedents of ρ.

Claim 4: We are given that σ' fails at least one consequent along ρ'. The consequents are the same in ρ and ρ', so σ must fail the corresponding consequent along ρ. ∎

Proof that $G_1 \Rightarrow G_2$ is implied by $C_1 \models_T G_2'$:
Suppose $G_1 \not\Rightarrow G_2$. Then, there exists a trace σ (in the state space defined by the atomic propositions) such that $\sigma \models_T G_1$, but $\sigma \not\models_T G_2$. We will construct a trace σ' of C_1 that is not accepted by G_2', witnessing that $C_1 \not\models_T G_2'$.

We construct σ' by augmenting the state space of σ with values for `init` and `accept`. For the initial state of σ', set `init` to be 1. In all other states of σ', set `init` to be 0. Because C_1 is a Mealy machine, we can always compute the value of `accept` by feeding σ as input to C_1. Thus, σ' is a trace of C_1 by construction. The resulting trace σ' has atomic proposition values that are the same as σ and has `accept` true in the last state (because σ is accepted by G_1).

Since $\sigma \not\models_T G_2$, we know there exists a terminal path ρ in G_2, of the same length as σ, such that σ satisfies all the antecedents in ρ but fails at least one consequent. Construct path ρ' in G_2' as follows: Match ρ edge-for-edge, picking the `accept` or $\neg$`accept` version of the edge in G_2' depending on the value of the `accept` signal in σ'. Since σ' ends with `accept` true, the constructed path ρ' ends at a terminal edge in G_2'.

Now, we see that σ' satisfies the antecedents in ρ' because the states/antecedents are the same as in σ and ρ (with the `accept`$'$ or $\neg$`accept`$'$ edge chosen correctly by the construction of ρ'). On the other hand, σ fails at least one consequent of ρ, so σ' must fail the corresponding consequent of ρ', since the consequents are the same in both paths. Therefore, σ' witnesses that $C_1' \not\models_T G_2'$. ∎

4 Model Checking under an Assumption

Besides assertion graph implication, the other main reasoning tool we wanted was how to perform GSTE model checking under an assumption. We notate this problem $C_0 \models_T (G_1 \Rightarrow G_2)$, meaning that all behaviors of a circuit C_0 that satisfy an assertion graph G_1 (the assumptions) also satisfy the assertion graph G_2. This construction is closely related to the preceding one.

The basic idea is that we build a monitor circuit C_1 for G_1 and augment C_0 with this monitor, in a non-interfering manner. Then, we modify G_2 so that it ignores traces that are not accepted by the monitor, resulting in verifying only the behaviors of C_0 that satisfy the assumptions of G_1. An alternative intuition is to consider the implication construction in Section 3.2 as the special case of model checking a completely unconstrained machine under the assumption of G_1; here, we constrain the inputs of C_1 to be the behaviors of C_0.

1. Without loss of generality, we assume that the initial vertex $v0$ of G_2 has in-degree of 0.
2. Build the monitor circuit C_1 from G_1.
3. Connect the inputs of C_1 to the state variables of C_0. In this way, C_1 will watch C_0 and indicate accept/reject depending on whether or not C_0's behavior obeys the assertion graph G_1. Call this combined circuit C_{01}.
4. Build G_2' from G_2 by edge-doubling and modifying the antecedents, exactly as in the implication construction.
5. $C_{01} \models_T G_2'$ iff $C_0 \models_T (G_1 \Rightarrow G_2)$.

Proof:
The constraints on `init` in the antecedents of G_2' guarantee that we only consider traces in which C_1 is properly initialized.

The monitor circuit C_1 has no effect on C_0. Therefore, C_{01} has the same traces as C_0, except for some additional state bits that determine whether or not G_1 would have accepted the trace.

Any path in G_2' that guesses accept/reject incorrectly on any edge will have its antecedent fail and will be ignored. For any path in G_2, there will always exist a corresponding path in G_2' that guesses accept/reject correctly for every edge. The only paths that are checked are the ones that are terminal in G_2', which means that they were terminal in G_2 as well, and also that the `accept` signal is true, which means that G_1 would have accepted the path. Thus, we check only the traces of C_0 that satisfy G_1. ∎

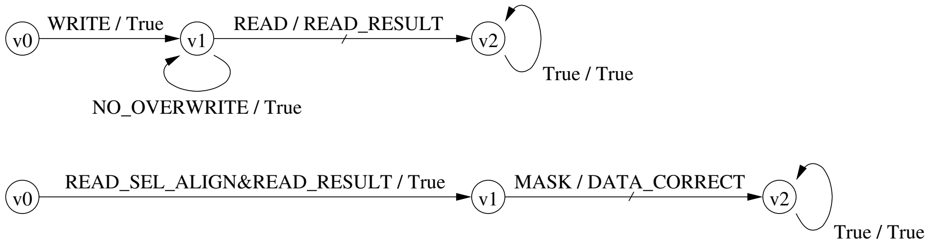


Fig. 4. Decomposing a Property. We have manually decomposed the assertion graph from Figure 1 into two smaller ones. The edge labels are as before, except READ $:= (\mathrm{ck} = 0) \wedge (\mathrm{we} = 0) \wedge (\mathrm{addr} = A)$ and READ_RESULT $:= (\mathrm{memout} = D)$, where memout is the internal data output of the memory array. We model check that the memory unit obeys the smaller assertion graphs, and then use our implication construction to verify that the two smaller assertion graphs imply the original specification. This process took less than 2/3 the time of verifying the original property directly.

5 Experimental Results

We have implemented the above algorithms into Intel's Forte verification system[4] and report their effectiveness on two verification tasks taken from real, industrial problems.

5.1 Decomposing a Verification Property: Verifying a Memory Unit

The first example is the verification of an industrial memory unit, using the assertion graph from Figure 1. Verifying this assertion graph on the memory unit by directly applying GSTE model checking required 56 seconds.

Alternatively, we manually decomposed the assertion graph into two smaller assertion graphs G_1 and G_2, which separates the memory behavior from the selection and alignment specifications. See Figure 4. GSTE model checking these two specifications on the memory unit took 28 seconds and 7 seconds, respectively. Note that, because of the $\forall$ semantics, we can produce the assertion graph for $G_1 \wedge G_2$ simply by having the two graphs share a single initial vertex. Accordingly, we verified that $(G_1 \wedge G_2)$ implies the original assertion graph, using the implication construction from Section 3.2. This step took 0.3 seconds, and the generated monitor circuit for $(G_1 \wedge G_2)$ had 5338 gates and 44 latches — far smaller than the memory unit. The total verification runtime was, therefore, less than 36 seconds, compared to the original 56 seconds.

Obviously, for such a small property, the time savings are not enough to repay the effort of decomposing the property. Nevertheless, we see that the decomposition does reduce the overall model-checking complexity, and our new algorithm does enable verifying automatically that a combination of sub-properties implies a more complex one.

[4] Forte is available for download at
http://www.intel.com/software/products/opensource/tools1/verification/
but our new algorithms are not yet part of the the standard distribution.

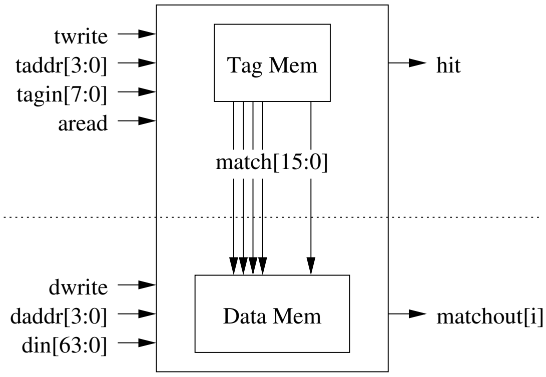


Fig. 5. Content-Addressable Memory (CAM). A CAM allows finding data by matching the value of a tag. In this CAM, a 64-bit data value is written at the same time as an 8-bit tag. Values can be read by supplying the correct tag. The $\mathtt{match}[i]$ signals indicate which of the 16 tags matches a supplied tag. The "outputs" on the right are for verification only: $\mathtt{hit} = \bigvee_i \mathtt{match}[i]$, and $\mathtt{matchout}[i] = \mathtt{datamem}[i]$ if $\mathtt{match}[i]$ is true, otherwise $\mathtt{matchout}[i] = 0$. The overall CAM has 1152 latches. Our verification will cut the circuit at the dotted line. We first verify the tag portion of the circuit, then use that assertion graph as an assumption to verify the data portion of the circuit.

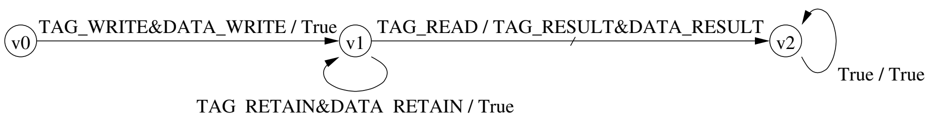
For larger, more challenging verification tasks, being able to decompose a difficult property into smaller ones, verify the smaller properties, and then conclude that the original property holds, is extremely useful.

5.2 GSTE with an Assumption: Content-Addressable Memory

The second example is from the verification of a content-addressable memory (CAM). This example illustrates GSTE model checking under an assumption.

A CAM allows finding data in its memory by matching a given tag value in an array of stored tags, i.e., by matching a value to the content of storage locations, rather than by address. CAMs are ubiquitous in modern microprocessors, where they are used to cache small amounts of frequently accessed data (e.g., in caches, TLBs, and assorted other buffers). Figure 5 shows the CAM for this example.

We wish to verify that the CAM as a whole satisfies the assertion graph G_2 in Figure 6. Verifying this assertion graph on the CAM by directly applying GSTE model checking required 15 seconds. Alternatively, to evaluate our algorithm for model checking under an assumption, we first verified the correct operation of the tag portion, in isolation, against the tag-correctness assertion graph G_1 in Figure 7. This verification took 0.8 seconds. Then, we abstracted away the tag portion of the CAM and used our algorithm for verification under an assumption to verify that G_2 holds, assuming that G_1 does: (data portion of CAM) $\models_T (G_1 \Rightarrow G_2)$. This verification took 7 seconds. Altogether, the decomposed verification was roughly twice as fast as the direct approach, and the monitor circuit for G_1 had only 12 latches, an order of magnitude less than the tag memory that was abstracted away.



$$
\begin{aligned}
\text{TAG\_WRITE} &:= (\text{twrite} = 1) \wedge (\text{taddr} = A) \wedge (\text{tagin} = T) \\
\text{DATA\_WRITE} &:= (\text{dwrite} = 1) \wedge (\text{daddr} = A) \wedge (\text{din} = D) \\
\text{TAG\_RETAIN} &:= (\text{twrite} = 0) \vee (\text{taddr} \neq A) \\
\text{DATA\_RETAIN} &:= (\text{dwrite} = 0) \vee (\text{daddr} \neq A) \\
\text{TAG\_READ} &:= (\text{aread} = 1) \wedge (\text{tagin} = T) \\
\text{TAG\_RESULT} &:= (\text{hit} = 1) \wedge \forall i[(i = A) \Rightarrow (\text{match}[i] = 1)] \\
\text{DATA\_RESULT} &:= \forall i[(i = A) \Rightarrow (\text{matchout}[i] = D)]
\end{aligned}
$$

Fig. 6. CAM Correctness Specification. This assertion graph specifies that if a tag and data values are written, followed by an arbitrary number of cycles in which they are not overwritten, followed by a read by the same tag, then the CAM must indicate a hit, and the `matchout` signal must give the correct data value at any matching locations.

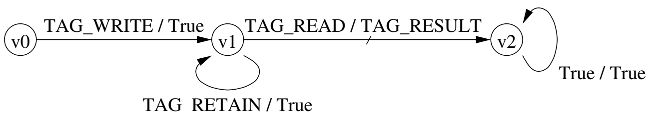


Fig. 7. Tag Correctness Specification. This assertion graph specifies that if a tag is written, not overwritten for an arbitrary number of cycles, and then the same tag is presented, the `hit` signal and the correct `match` signal must be asserted. We first verify this property on the tag portion of the circuit. Then, we use this assertion graph as an assumption to abstract away the tag portion of the circuit when verifying the whole CAM.

As in the previous example, the time savings on a small verification task are not enough to repay the time to manually decompose the problem. Nevertheless, this example does demonstrate how our new algorithm runs efficiently and enables decomposing a harder verification problem into smaller, easier ones. In general, we envision using this style of proof for simplifying complex verification tasks, and also for verification with IP cores (portions of a circuit supplied by third-parties, for which functionality is specified, but internal details are not visible) as well as the verification of partial or incomplete circuits.

6 Conclusion and Future Work

We have presented new algorithms for reasoning about GSTE assertion graphs. These algorithms appear efficient in theory, and preliminary experiments indicate that they are efficient in practice as well. Given the increasing practical importance of GSTE model checking, the need for (practically efficient) supporting theory and algorithms is great. This work is a first step.

The practical success of GSTE is the justification for studying assertion graphs. In theory, assertion graphs are simply a new variety of automata, with equivalent expressive power to established varieties of automata, so an obvious, fundamental question is to elucidate whether and how GSTE is gaining efficiency advantages over older techniques. Do assertion graphs facilitate writing specifications in a manner that enables more efficient model checking? Are other aspects of GSTE, completely separable from assertion graphs, more important for efficiency? Can we leverage these ideas with other verification methods? On the other hand, perhaps the practical successes have been primarily the result of the overall verification methodology, the types of verification tasks undertaken, or the skill of the verification engineers. Assertion graphs and GSTE give symbolic-trajectory-evaluation-based approaches comparable expressive power to other model-checking approaches, so it is now possible to make direct comparisons.

Focusing on assertion graphs, research is needed on composing and decomposing assertion graphs. For example, given the $\forall$ semantics, it should be possible to decompose a large assertion graph into the conjunction of smaller ones, as is possible in formalizations of timing graphs [2]. Such a decomposition could reduce the complexity of model checking.

A related, and perhaps more immediately applicable, direction for research is to look for transformations and inference rules for assertion graphs. For example, it is easy to see that adding edges, weakening antecedents, or strengthening consequents are all operations that cannot enlarge the set of traces accepted by an assertion graph. Perhaps it is possible to develop a powerful set of inference rules to reason about assertion graphs, without having to perform model checking.

The work presented here are fundamental building blocks for reasoning about assertion graphs. An important next step is to develop compositional verification theorems, so that we can automate the process of stitching together partial verification results.

Finally, although assertion graphs are interesting to consider in isolation as a variety of automata, in practice their use is intimately tied to GSTE model checking. This connection suggests that it may be interesting to consider weaker notions of implication (and equivalence). For example, rather than defining $G_1 \Rightarrow G_2$ to mean $L(G_1) \subseteq L(G_2)$, we could use the weaker definition: $\forall$ circuits $M.(M \models G_1) \Rightarrow (M \models G_2)$. Under all the different acceptance conditions, we have constructed small assertion graphs G_1 and G_2 such that $L(G_1) \neq L(G_2)$, but that are equivalent under the weaker definition because no circuit satisfies either one. (The intuition is that real circuits cannot generate arbitrary sets of strings, e.g., a circuit can always be run for one more clock cycle, generating a longer string.) We do not know whether the difference between these definitions is theoretically interesting or practically important.

In general, increasing evidence demonstrates the practical value of GSTE and assertion graphs, but the supporting infrastructure is underdeveloped. Much work remains to be done.

References

1. Mark Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *35th Design Automation Conference*, pages 538–541. ACM/IEEE, 1998.
2. Nina Amla, E. Allen Emerson, and Kedar S. Namjoshi. Efficient decompositional model-checking for regular timing diagrams. In *Correct Hardware Design and Verification: 10th IFIP WG 10.5 Advanced Research Working Conference (CHARME'99)*, pages 67–81. Springer, 1999. Lecture Notes in Computer Science Number 1703.
3. Bob Bentley. High level validation of next generation microprocessors. In *International Workshop on High-Level Design, Validation, and Test*. IEEE, 2002.
4. Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Computer-Aided Verification: 13th International Conference*, pages 454–464. Springer, 2001. Lecture Notes in Computer Science Number 2102.
5. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981. Published 1982 as Lecture Notes in Computer Science Number 131.
6. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
7. Alan J. Hu, Jeremy Casas, and Jin Yang. Efficient generation of monitor circuits for GSTE assertion graphs. In *International Conference on Computer-Aided Design*. IEEE/ACM, 2003. To appear.
8. Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.
9. Zohar Manna and Amir Pnueli. Specification and verification of concurrent programs by $\forall$-automata. In *Symposium on Principles of Programming Languages*, pages 1–12. ACM, 1987.
10. Kyle L. Nelson, Alok Jain, and Randal E. Bryant. Formal verification of a superscalar execution unit. In *34th Design Automation Conference*, pages 161–166. ACM/IEEE, 1997.
11. Márcio T. Oliveira and Alan J. Hu. High-level specification and automatic generation of IP interface monitors. In *39th Design Automation Conference*, pages 129–134. ACM/IEEE, 2002.
12. Manish Pandey, Richard Raimi, Derek L. Beatty, and Randal E. Bryant. Formal verification of PowerPC arrays using symbolic trajecory evaluation. In *33rd Design Automation Conference*, pages 649–654. ACM/IEEE, 1996.

13. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in Cesar. In *5th International Symposium on Programming*, pages 337–351. Springer, 1981. Lecture Notes in Computer Science Number 137.
14. Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In *23rd International Colloquium on Automata, Languages, and Programming*, pages 336–347. Springer, 1996. Lecture Notes in Computer Science Number 1099.
15. Andrew Seawright and Forrest Brewer. High-level symbolic construction techniques for high performance sequential synthesis. In *30th Design Automation Conference*, pages 424–428. ACM/IEEE, 1993.
16. Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, 1995.
17. Fabio Somenzi and Roderick Bloem. Efficient Büchi automata from LTL formulae. In *Computer-Aided Verification: 12th International Conference*, pages 248–263. Springer, 2000. Lecture Notes in Computer Science Number 1855.
18. Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society, 1986.
19. Jin Yang and Amit Goel. GSTE through a case study. In *International Conference on Computer-Aided Design*, pages 534–541. IEEE/ACM, 2002.
20. Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. In *International Conference on Computer Design*, pages 360–365. IEEE, 2001.
21. Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation — abstraction in action. In *Formal Methods in Computer-Aided Design: Fourth International Conference*, pages 70–87. Springer, 2002. Lecture Notes in Computer Science Number 2517.