
A FAITHFUL SEMANTICS FOR GENERALISED SYMBOLIC TRAJECTORY EVALUATION

KOEN CLAESSEN^a AND JAN-WILLEM ROORDA^b

^a Chalmers University of Technology, Sweden
e-mail address: koen@chalmers.se

^b Fenix Design Automation, the Netherlands
e-mail address: janwillem@fenix-da.com

ABSTRACT. Generalised Symbolic Trajectory Evaluation (GSTE) is a high-capacity formal verification technique for hardware. GSTE is an extension of Symbolic Trajectory Evaluation (STE). The difference is that STE is limited to properties ranging over finite time-intervals whereas GSTE can deal with properties over unbounded time.

GSTE uses *abstraction*, meaning that details of the circuit behaviour are removed from the circuit model. This improves the capacity of the method, but has as down-side that certain properties cannot be proven if the wrong abstraction is chosen.

A semantics for GSTE can be used to predict and understand why certain circuit properties can or cannot be proven by GSTE. Several semantics have been described for GSTE by Yang and Seger. These semantics, however, are not *faithful* to the proving power of GSTE-algorithms, that is, the GSTE-algorithms are *incomplete* with respect to the semantics. The reason is that these semantics do not capture the abstraction used in GSTE precisely.

The abstraction used in GSTE makes it hard to understand why a specific property can, or cannot, be proven by GSTE. The semantics mentioned above cannot help the user in doing so. So, in the current situation, users of GSTE often have to revert to the GSTE algorithm to understand why a property can or cannot be proven by GSTE.

The contribution of this paper is a *faithful semantics* for GSTE. That is, we give a simple formal theory that deems a property to be true if-and-only-if the property can be proven by a GSTE-model checker. We prove that the GSTE algorithm is sound and complete with respect to this semantics. Furthermore, we show that our semantics for GSTE is a generalisation of the semantics for STE and give a number of additional properties relating the two semantics.

1998 ACM Subject Classification: B.6.3, F.3.2, F.4.3.

Key words and phrases: Formal Verification, Formal Specification, Model Checking, Symbolic Simulation, Generalized Symbolic Trajectory Evaluation, Semantics.

lsuper bThis work was carried out while employed at Chalmers University.

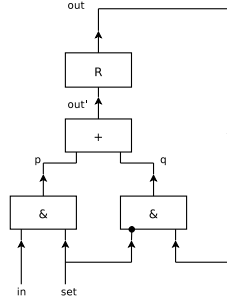


Figure 1: A memory cell

1. INTRODUCTION

The rapid growth in hardware complexity has led to a need for *formal verification* of hardware designs to prevent bugs from entering the final silicon. *Model checking* is a verification method in which a model of a system is checked against a *property*, describing the desired behaviour of the system over time. Today, all major hardware companies use model checkers in order to reduce the number of bugs in their designs.

1.1. Symbolic Trajectory Evaluation. *Symbolic Trajectory Evaluation* (STE) [12] is a high-performance model checking technique based on *simulation*. STE combines three-valued simulation (using the standard values 0 and 1 together with the extra value X, “don’t know”) with symbolic simulation (using symbolic expressions to drive inputs). STE has been extremely successful in verifying properties of circuits containing large data paths (such as memories, FIFOs, and floating point units) that are beyond the reach of traditional symbolic model checking [1, 10, 12].

Consider the circuit in Figure 1. The circuit consists of two AND-gates, an OR-gate, a register (depicted by the letter R), and an inverter (depicted by the black dot). The register has output node `reg` and input node `reg'`. The value of the output of the register at time $t + 1$ is the value of its input at time t . The memory cell can be written with the value at node `in` by making node `set` high.

In STE, circuit specifications are *assertions* of the form $A \implies C$. Here, A is called the *antecedent* and C the *consequent*. For example, an STE-assertion for the memory cell is:

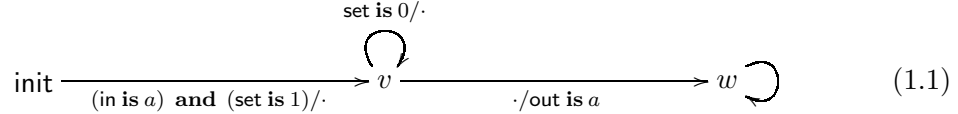
$$(\text{in is } a) \text{ and } (\text{set is } 1) \implies \mathbf{N}(\text{out is } a)$$

Here a is a *symbolic constant*¹, which can take on the value 0 or 1, and `in`, `set` and `out` are *node names*. \mathbf{N} is the next-time operator. The assertion states that when node `in` has value a , and node `set` has value 1, then at the next point in time, node `out` must have value a .

¹ The name *symbolic constant* is used to indicate that the variable keeps a constant value over different points in time. In plain STE, such variables are called *symbolic variables*. As this paper deals with GSTE, we will use the GSTE terminology even when we discuss plain STE.

1.2. Generalised Symbolic Trajectory Evaluation. One of the main disadvantages of STE is that it can only deal with properties ranging over a finite number of time-steps. *Generalised Symbolic Trajectory Evaluation* (GSTE) [16, 15, 18, 17] is an extension of STE that can deal with properties ranging over unbounded time.

In GSTE, circuit properties are given by *assertion graphs*. For example, an assertion graph for the memory cell is:



In the assertion graph, each edge is labelled with a pair A/C . As in STE, A is called the antecedent and C is called the consequent. The syntax of A and C is like the syntax of the antecedent and consequent in STE without the next-time operator \mathbf{N} . The \mathbf{N} operator can not be used because each edge only represents a single time-point. A dot (\cdot) means an empty antecedent or consequent.

The assertion graph above states that if we write value a to the memory cell, and then for arbitrary many time-steps we do not write, the memory cell still contains value a .

Each finite path, starting in the initial vertex init of the graph, represents an STE property. For instance, the finite paths through the assertion graph above represent the following STE properties:

$$\begin{array}{ll}
 \text{(in is } a) \text{ and (set is } 1) & \implies \mathbf{N}(\text{out is } a) \\
 \text{(in is } a) \text{ and (set is } 1) \text{ and } \mathbf{N}(\text{set is } 0) & \implies \mathbf{NN}(\text{out is } a) \\
 \text{(in is } a) \text{ and (set is } 1) \text{ and } \mathbf{N}(\text{set is } 0) \text{ and } \mathbf{NN}(\text{set is } 0) & \implies \mathbf{NNN}(\text{out is } a) \\
 \dots &
 \end{array}$$

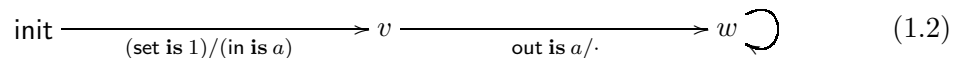
Each of these assertions can be proven by an STE model checker. But, as the set of assertions is infinite, we cannot use plain STE to prove all of them. However, if we use GSTE to prove that the circuit satisfies the above assertion graph, it follows that all STE-assertions represented by the assertion graph hold as well.

Note that in GSTE, just like in STE, the initial values of registers are ignored.

1.3. Earlier work on semantics for GSTE. A *semantics* for GSTE can be used to predict and understand why certain circuit properties can or cannot be proven by GSTE. In [17, 18] three semantics for GSTE are distinguished: (1) the *strong semantics*, (2) the *normal semantics*, and (3) the *fair semantics*. The semantics have in common that a circuit satisfies an assertion-graph if it satisfies all *appropriate* paths in the assertion graph. The meaning of appropriate differs over the three semantics, as we explain in the following paragraphs. As in [5], we refer to this class of semantics as the \forall -*semantics*, because these semantics really consider *all* concrete paths, rather than approximating this quantification by applying abstraction.

In the strong semantics, a circuit satisfies a GSTE assertion graph if-and-only-if the circuit satisfies all STE-assertions corresponding to *finite* paths in the assertion graph. For instance, as the memory cell satisfies the set of finite assertions above, it also satisfies assertion graph (1.1).

Consider the following assertion graph:



Intuitively, we might want the above assertion graph to state that if at some time-point node `out` has value a , and just before that, node `set` was high, then at this time-point node `in` should have value a . This is an example of a *backwards property*, that is, a property in which a consequent depends on an antecedent at a later time-point.

The strong semantics cannot deal with such backwards properties. For instance, for the above property, the path starting in vertex `init` and ending in vertex v corresponds to the assertion

$$(\text{set is } 1) \implies (\text{in is } a)$$

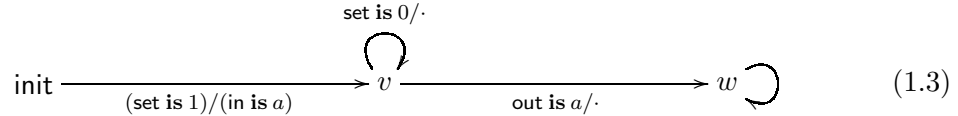
This assertion is, of course, not true for the memory cell. But, any run of the circuit that makes `in is a` fail, makes $\mathbf{N}(\text{out is } a)$ fail as well. So, intuitively, the assertion is not satisfied because a consequent failed before the antecedent it depended on could fail.

In the normal semantics, a circuit satisfies a GSTE assertion graph if-and-only-if the circuit satisfies the STE-assertions corresponding to all *infinite* paths in the assertion graph. Therefore, the normal semantics can deal with backwards properties as well. For instance, in assertion graph (1.2), there is only one infinite path. This path corresponds to the following assertion:

$$(\text{set is } 1) \text{ and } \mathbf{N}(\text{out is } a) \implies (\text{in is } a)$$

As any circuit trace that satisfies the antecedent satisfies the consequent as well, this assertion is satisfied by the circuit. Thus, in the normal semantics, the GSTE assertion graph is satisfied.

Finally, the need for the fair semantics is illustrated by the following example. Consider the assertion graph:



The assertion graph above states that if at some time-point node `out` has value a , and before that, for a period of time no values were written to the memory-cell, and before that, `set` was high, then at this time-point `in` should have value a .

In the normal semantics, the memory cell circuit does not satisfy this assertion graph. Consider the infinite path starting in `init` and then cycling at the self-loop at v for ever. This path corresponds to the infinite assertion:

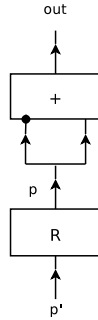
$$(\text{set is } 1) \text{ and } \mathbf{N}(\text{set is } 0) \text{ and } \mathbf{NN}(\text{set is } 0) \text{ and } \dots \implies \text{in is } a$$

For a given a , this assertion can be falsified by the trace in which value $\neg a$ is written at time 0, and is kept in memory since then.

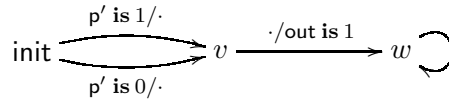
In the fair semantics for GSTE, this problem is solved by selecting a set of *fair edges*. The semantics only considers paths that visit every fair edge infinitely often. For instance, if in the above assertion graph the edge from vertex w to itself is made fair, the assertion graph holds in the fair semantics.

1.4. GSTE model checking. In the same papers [17, 18], model checking algorithms for normal, strong and fair GSTE are described. It is proven that the model checking algorithms are sound with respect to their corresponding semantics. However, the algorithms are not *complete*. The reason is that the \forall -semantics do not precisely capture the information loss due to the three-valued abstraction in GSTE.

For example, consider the following circuit



and the following assertion graph



The assertion graph represents the following STE-assertions:

$$\begin{aligned}
 p' \text{ is } 1 &\implies \mathbf{N}(\text{out is } 1) \\
 p' \text{ is } 0 &\implies \mathbf{N}(\text{out is } 1)
 \end{aligned}$$

Both assertions hold. So, the semantics described above predict that the circuit satisfies the assertion graph.

However, it turns out that the GSTE-algorithm cannot prove the assertion graph! The reason is that GSTE algorithms only compute one three-valued assertion for each edge in the assertion graph. This is in general not enough to take account for all STE assertions corresponding to all paths through the assertion graph, so a certain *information loss* happens. In this particular case, the state calculated on the edge from v to w gives value \mathbf{X} to node out . This can be explained as follows. The antecedent at the top edge between vertices init and v requires node p' to have value 1. The antecedent at the bottom edge requires node p' to have value 0. Node p' is the input to a register with node p as output. So, when the edge from v to w is reached via the top edge between init and v node p will receive value 1. When the edge from v to w is reached via the bottom edge between init and v , node p will receive value 0. As the value of node p should comply with both paths, the algorithm chooses value \mathbf{X} for node p , and thus node out receives value \mathbf{X} as well.

1.5. The problem. The previous example illustrates that the \forall -semantics for GSTE discussed previously cannot be used to explain how the three-valued abstraction causes certain properties to be not provable with GSTE. This can lead to situations where seemingly trivial changes to either the circuit or the assertion can suddenly make an assertion not provable anymore.

This is an undesirable situation. We believe that a faithful semantics for GSTE is needed.

A faithful semantics deems a property to be true if-and-only-if the property can be proven by a GSTE-model checker. Without a faithful semantics, a GSTE verification engineer is left to the particular internals of the model checker at hand to understand what can and cannot be proved. Also, a faithful semantics can be used to understand differences between different GSTE model checkers. For example, the GSTE semantics of satGSTE [14]

is expressed using successive unrollings of the assertion graph as STE assertions. However, the abstraction obtained in that way does not correspond to the abstraction in standard GSTE model checkers. This means that there are assertion graphs for which satGSTE and standard GSTE model checkers give different answers.

To further clarify the importance of a faithful GSTE semantics, we would like to point out that there is a difference between the use of abstraction in (G)STE, and the application of abstraction as a performance enhancer in model checkers for standard temporal logics like LTL and CTL. In the latter case, a model checker might simply give up when it happens to choose an abstraction that is too weak to prove a property, but it is still clear to the verification engineer what the specification means. In (G)STE, what abstraction to use in the model checker is an *artefact of the specification*, not an artefact of the model checker. So, in (G)STE it is vital to understand what a specification means, separate from a particular model checker, including the abstraction that is specified.

In previous work [9], we have described a faithful semantics for STE. However, up till now, no faithful semantics for GSTE has been described.

1.6. Our contribution. In this paper, we present a semantics for GSTE that is faithful to the proving power of the GSTE model checking algorithm. Compared to the semantics described in [17, 18], our semantics corresponds to the strong semantics of GSTE. That is, in this paper, we do not consider backwards properties or fairness constraints, which remains future work. One difference with the strong semantics in [17, 18] is that our semantics captures the three-valued abstraction of GSTE precisely, and thus can be used to explain the information loss caused by the three-valued abstraction in GSTE.

Another difference is that our semantics for GSTE follows the same structure as the semantics for STE [9, 12, 6]. For instance, where STE deals with *sequences* to represent abstract circuit behaviour, our GSTE semantics uses *sequence graphs*. Here, a sequence graph is a mapping from edges in an assertion graph to abstract circuit states. We show that our GSTE semantics is a generalisation of the STE semantics. That is, given a linear assertion graph, the STE-semantics and GSTE-semantics are equivalent. Finally, we state a number of additional properties relating the two semantics.

We believe that our faithful semantics for STE is an important contribution to the research on GSTE for at least two reasons.

First of all, a faithful semantics makes GSTE more accessible to novice users: a faithful semantics enables users to understand the abstraction used in GSTE, without having to understand the details of the model checking algorithm. Additionally, in this paper, we aim at increasing the understanding for GSTE users of subtle cases of information-loss due to abstraction by providing enlightening examples.

Furthermore, a faithful semantics for GSTE can be used as basis for research on new GSTE model checking algorithms and other GSTE tools. To illustrate this, in previous work [8], we described a new SAT-based model checking algorithm for STE and proven that it is sound and complete with respect to our faithful semantics for STE presented in [9]. Without a faithful semantics for STE, we would have been forced to prove the correctness of our algorithm by relating it to other model checking algorithms for STE. This is clearly a more involved and less elegant approach. In fact, we believe that without constructing a faithful semantics for STE first, we would not have obtained the level of understanding of STE needed to develop the new SAT-based model checking algorithm.

In the same way, we expect that the faithful semantics for GSTE presented in this paper will open the door for new research on GSTE model checking algorithms and other GSTE tools.

1.7. Other related work. The following papers are based on the \forall -semantics for GSTE.

GSTE as partitioned model checking. In [11], the relation between GSTE and classic symbolic model checking is studied. It is explained how GSTE can be seen as a *partitioned* form of classic symbolic model checking. However, the abstraction of GSTE is not taken into account. Therefore, this paper, focussing on the abstraction in GSTE, is complementary to [11].

Using SAT for debugging of GSTE assertion graphs. In [14], the tool *satGSTE* is presented. The tool considers a finite subset of all finite paths in an assertion graphs, for instance, all paths up to a certain length. For each path in this subset, the tool model checks the corresponding STE assertion. The authors explain how the tool can be used to debug and refine GSTE assertion graphs. However, their tool does not follow the same semantics as standard GSTE model checking algorithms. Thus, certain counter examples that would occur in a standard GSTE model checker due to the use of abstraction cannot be found with their algorithm.

Monitor circuits for GSTE assertion graphs. In (conventional, non-symbolic) simulation, a model of a circuit is fed with a large number of inputs. For every input it is checked whether the output is as expected. Typically, a *monitor circuit* is used to make this check. The monitor circuit observes the system under verification without interfering. During each step of the simulation, it indicates whether the system has obeyed the formal specification thus far.

In [4, 7] methods for automatic construction of monitor circuits for GSTE assertion graphs are described. The method in [4] requires the use of a symbolic simulator if the assertion graph contains symbolic constants. In [7] it is explained how, for the class of so-called *simulation friendly assertion graphs*, the method of [4] can be extended to deal with symbolic constants even in conventional non-symbolic simulation.

The papers explain how monitor circuits can be used to make a bridge between GSTE model checking and conventional simulation. For instance, monitor circuits can be used to quickly debug and refine GSTE specifications before trying to use more labour intensive GSTE model checking.

Reasoning about GSTE assertion graphs. Using the construction of monitor circuits for GSTE assertion graphs, [5] describes two algorithms that can be used in compositional verification using GSTE. The first algorithm decides whether one assertion graph implies another. The second algorithm can be used to model check an assertion graph under the assumption that another assertion graph is true.

1.7.1. *Relation to this paper.* Each of the papers above is based on the \forall -semantics for GSTE. As explained above, the \forall -semantics are not faithful to the proving power of the GSTE model checking algorithms. So, it can occur that a tool described in the papers deems a GSTE assertion to be true, while the GSTE model checking algorithm cannot prove it.

For instance, the monitor circuits described above cannot be used to debug and refine assertions graphs that are true in the \forall -semantics but yield a spurious counter-example when trying to prove them with a GSTE model checker. The satGSTE tool is limited in the same way. We elaborate further on this in the future work section of this paper.

1.8. **Structure of this paper.** In the next section, we revisit the semantics of STE assertions. Then, in Section 3, we present our semantics of GSTE assertion graphs. In Section 4, we compare the STE semantics with the GSTE semantics by giving a number of properties describing their relation. In Section 5, we describe the GSTE model checking algorithm and show that it is sound and complete with respect to our semantics. Finally, in Section 6, we conclude and give suggestions for future work.

2. STE PRELIMINARIES

A semantics for STE was first described by Seger and Bryant [12]. Later, a simplified and easier to understand semantics was given by Melham and Jones [6]. Both of these semantics are expressed in terms of a *next state function*, expressing the relationship between two consecutive states in the circuit. Unfortunately, neither of these semantics matches the proving power of currently available STE model checkers. The problem is that they cannot deal with *combinational* properties (properties ranging over one single point in time). All such properties are deemed to be false by the semantics. Therefore standard next state semantics does not seem to be a good starting point for finding a faithful semantics for GSTE.

In previous work [9], we have described an alternative semantics for STE that actually is faithful to the proving power of STE model checkers. The semantics is called the *closure semantics*. Informally, the closure semantics only differs from the traditional STE semantics for combinational properties.

A main ingredient of the closure semantics for STE is the concept of a *closure function*. The idea is that a closure function takes as input a state of the circuit, and calculates all information about the circuit state at the same point in time that can be derived by propagating the information in the input state in a forwards fashion. In the next section, we give an alternative semantics for GSTE also based on closure functions.

In this section we briefly describe the closure semantics for STE. For more examples and a discussion on the differences with the semantics given in [12, 6], we refer the reader to [9].

Readers familiar with [9] can skip most of this section; compared to [9] we slightly changed notation in the definition of the closure function on sequences, and we introduced an extra variant of a closure semantics called the simple semantics. Furthermore, we adapted the terminology to GSTE: we call the variables in STE-assertions *symbolic constants* to indicate that they keep a constant value over time. Finally, we use *finite* sequences to represent circuit behaviour, as opposed to the standard use of infinite sequences. Notice that this is a very superficial change on the notational level; it does not change the semantics

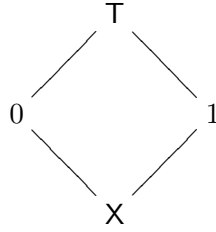


Figure 2: The STE lattice

$v \mid \neg v$	$\& \mid 0 \ 1 \ X \ T$	$+ \mid 0 \ 1 \ X \ T$	$\sqcup \mid 0 \ 1 \ X \ T$	$\sqcap \mid 0 \ 1 \ X \ T$
0 1	0 0 0 0 T	0 0 1 X T	0 0 T 0 T	0 0 X X 0
1 0	1 0 1 X T	1 1 1 1 T	1 T 1 1 T	1 X 1 X 1
X X	X 0 X X T	X X 1 X T	X 0 1 X T	X X X X X
T T	T T T T T	T T T T T	T T T T T	T 0 1 X T

Figure 3: Four-valued extensions of the logical operators, least upper bound and greatest lower bound operators.

itself. The reason for making the change is that it enables us to considerably simplify the proof of Proposition 4.1 on page 24.

2.1. Values and Circuits States. Values In STE, we can abstract away from specific Boolean values of a node, by using the value X , which stands for *unknown*. The value T stands for *over constrained*. A node takes on the value T when is required to have both value 0 and value 1.

On this set an *information-ordering* \leq is introduced, see Figure 2. The unknown value X contains the least information, so $X \leq 0$ and $X \leq 1$, while 0 and 1 are incomparable. The over-constrained value contains the most information, so $0 \leq T$ and $1 \leq T$. If $v \leq w$ it is said that v is *weaker* than w .

The set V together with the ordering \leq forms a *lattice*. The *least upper bound operator* is written \sqcup , the *greatest lower bound operator* is written \sqcap , see Figure 3.

The logical operators for conjunction, written $\&$, disjunction, written $+$, and negation, written \neg , are extended to the four-valued domain as in Figure 3.

States A *circuit state*, written s : **State**, is a function from the set of nodes of a circuit to the values $\{0, 1, X, T\}$ ².

2.2. Closure functions. In our semantics for STE, *closure functions* are used as circuit models. The idea is that a closure function, written $F : \mathbf{State} \rightarrow \mathbf{State}$ takes as input a state of the circuit, and calculates all information about the circuit state at the *same* point

²Such an STE circuit state can be thought of as representing a *set* of regular states, commonly used in set-based abstractions, where X represents the set $\{0, 1\}$ and T represents the empty set. This view induces a natural set-theoretic lattice, with set inclusion as its ordering. It is perhaps confusing that the standard STE lattice ordering (also used here) goes exactly the other way around; i.e. the STE \sqcup corresponds to \cap and \sqcap corresponds to \cup .

in time that can be derived by propagating the information in the input state in a *forwards* fashion.

Example 2.1. The closure function for a circuit consisting of a single AND-gate with inputs \mathbf{p} and \mathbf{q} , and output \mathbf{r} is given by the table below. Here, s is a state and n is a node.

$$\begin{array}{c|c} n & F(s)(n) \\ \hline \mathbf{p} & s(\mathbf{p}) \\ \mathbf{q} & s(\mathbf{q}) \\ \mathbf{r} & (s(\mathbf{p}) \& s(\mathbf{q})) \sqcup s(\mathbf{r}) \end{array}$$

The least upper bound operator in the expression for $F(s)(\mathbf{r})$ combines the value of \mathbf{r} in the given state s , and the value for \mathbf{r} that can be derived from the values of \mathbf{p} and \mathbf{q} , being $s(\mathbf{p}) \& s(\mathbf{q})$.

A state $s : \{\mathbf{p}, \mathbf{q}, \mathbf{r}\} \rightarrow \mathbb{V}$ can be written as a vector $s(\mathbf{p}), s(\mathbf{q}), s(\mathbf{r})$. For example, the state that assigns the value 1 to nodes \mathbf{p} and \mathbf{q} and the value X to node \mathbf{r} is written as 11X. Applying the closure function to the state 11X yields 111. The reason is that when both inputs to the AND-gate have value 1, then by forwards propagation of information, also the output has value 1. Applying the closure function to state 1XX yields 1XX. The reason is that the output of the AND-gate is unknown when one input has value 1 and the other value X. The *forwards* nature of simulation becomes clear when the closure function is applied to state XX1, resulting in XX1. Although the inputs to the AND-gate must have value 1 when the output of the gate has value 1, this cannot be derived by forwards propagation.

A final example shows how the over-constrained value T can arise. Applying the closure function to state 0X1 yields 0XT. The reason is that the input state gives node \mathbf{r} value 1 and node \mathbf{p} value 0. From \mathbf{p} having value 0 it can be derived by forwards propagation that \mathbf{r} has value 0, therefore \mathbf{r} receives the over-constrained value T. \square

A closure function is a function $F : \mathbf{State} \rightarrow \mathbf{State}$ satisfying the following three conditions:

- F is *monotonic*, that is, for all states s_1, s_2 : $s_1 \leq s_2$ implies $F(s_1) \leq F(s_2)$. This means that a more specified input state cannot lead to a less specified result. The reason is that given a more specified input state, more information about the state of the circuit can be derived.
- F is *idempotent*, that is, for every state s : $F(F(s)) = F(s)$. This means that repeated application of the closure function has the same result as applying the function once. The reason is that the closure function should derive all information about the circuit state in one go.
- F is *extensive*, that is, for every state s : $s \leq F(s)$. This means that the application of a closure function to a circuit state should yield a state at least as specified as the input state. The reason is that the closure function is required not to lose any information.

Netlists Here, a netlist is an acyclic list of definitions describing the relations between the values of the nodes. Inverters are not modelled explicitly in our netlists, instead they occur implicitly for each mention of the negation operator \neg on the inputs of the gates. Registers are not mentioned explicitly in the netlist either. Instead, for a register with output node n in the circuit, the input of the register is node n' which is mentioned in the netlist. For simplicity, we only allow AND-gates and OR-gates in netlists. It is, however, straightforward to extend this notion of netlists to include more operations.

Induced Closure Function Given the netlist of a circuit c , the *induced closure function* for the circuit, written F_c , can easily be constructed by interpreting each definition in the netlist as a four-valued gate (see Figure 3). Each

Given a state s , a circuit c , and a circuit node n , we calculate $F_c(s)(n)$ as follows:

- If n is a circuit input or the output of a register, then we define $F_c(s)(n) = s(n)$.
- If n is the output of an AND-gate with input nodes p and q , then we define

$$F_c(s)(n) = (F_c(s)(p) \& F_c(s)(q)) \sqcup s(n).$$

- If n is the output of an OR-gate with input nodes p and q , then we define

$$F_c(s)(n) = (F_c(s)(p) + F_c(s)(q)) \sqcup s(n).$$

- If n is the output of an inverter with input node p , then we define

$$F_c(s)(n) = \neg F_c(s)(p) \sqcup s(n).$$

This definition is well-defined because netlists are acyclic by definition.

Proposition 2.2. *The induced closure function for a circuit is by construction monotonic, idempotent and extensive.*

Proof. The closure function F_c is a composition of the monotonic functions of four-valued negation, four-valued conjunction and least upper bound, therefore it is monotonic itself.

As netlists are acyclic by definition, we can prove properties by induction over the definition of a node. We prove idempotency by proving $F_c(F_c(s))(n) = F_c(s)(n)$ by induction on the definition of n . Assume n is in the set of input- and state-holding nodes $\mathcal{I} \cup \mathcal{S}$, then $F_c(F_c(s))(n) = F_c(s)(n)$ by definition. If n is defined by $n = p$ AND q , then:

$$\begin{aligned} & F_c(F_c(s))(n) \\ &= (F_c(F_c(s))(p) \& F_c(F_c(s))(q)) \sqcup F_c(s)(n) && \text{(definition)} \\ &= (F_c(s)(p) \& F_c(s)(q)) \sqcup F_c(s)(n) && \text{(ind. hyp.)} \\ &= (F_c(s)(p) \& F_c(s)(q)) \sqcup (F_c(s)(p) \& F_c(s)(q)) \sqcup s(n) && \text{(definition)} \\ &= (F_c(s)(p) \& F_c(s)(q)) \sqcup s(n) && \text{(property } \sqcup \text{)} \\ &= F_c(s)(n) && \text{(definition)} \end{aligned}$$

A similar argument holds when n is defined by a different gate definition.

The extensivity of F_c follows directly from its definition: If n is an input or state holding node then $F_c(s)(n) = s(n)$, otherwise $F_c(s)(n)$ is defined as the least upper bound of $s(n)$ and another expression, so $s(n) \leq F_c(s)(n)$. \square

2.3. A closure function for sequences. Sequences A *sequence of depth d* , written $\sigma : \{0, 1, \dots, d\} \rightarrow \mathbf{State}$, is a function from a point in time to a circuit state, describing the behaviour of a circuit over time. The set of all sequences is written **Seq**. A *three-valued sequence* is a sequence that does not assign the value \top to any node at any time.

The order \leq and the operators \sqcup and \sqcap are extended to sequences in a point-wise fashion. That is, the order \leq on sequences is defined by $\sigma_1 \leq \sigma_2$ iff for all n , $\sigma_1(n) \leq \sigma_2(n)$. Furthermore, $(\sigma_1 \sqcup \sigma_2)(n) = (\sigma_1(n) \sqcup \sigma_2(n))$, and $(\sigma_1 \sqcap \sigma_2)(n) = (\sigma_1(n) \sqcap \sigma_2(n))$.

Closure for sequences In STE, a circuit is simulated over multiple time steps. During simulation, information is propagated forwards through the circuit and through time, from each time step t to time step $t + 1$. Note that the initial values of registers are ignored.

To model this forwards propagation of information through time, a *closure function for sequences*, notation $F^\rightarrow : \mathbf{Seq} \rightarrow \mathbf{Seq}$, is used. Given a sequence, the closure function for

sequences calculates all information that can be derived from that sequence by forwards propagation. The closure function for sequences preserves the depth of the given sequence.

Recall that for every register with output n , the input to the register is node n' . Therefore, the value of node n' at time t is propagated to node n at time $t + 1$ in the forwards closure for sequences.

Given a circuit state s , the function next calculates the information that is propagated by the registers, and is defined by:

$$\text{next}(s)(n) = \begin{cases} s(n'), & n \in \mathcal{S} \\ \mathbf{X}, & \text{otherwise} \end{cases}$$

The closure function for sequences F^\rightarrow is defined in terms of a closure function F . Given a closure function F for a circuit with a set of outputs of registers \mathcal{S} , the *closure function for sequences*, written $F^\rightarrow : \mathbf{Seq} \rightarrow \mathbf{Seq}$, is inductively defined by:

$$\begin{aligned} F^\rightarrow(\sigma)(0) &= F(\sigma(0)) \\ F^\rightarrow(\sigma)(t+1) &= F(\sigma(t+1) \sqcup \text{next}(F^\rightarrow(\sigma)(t))) \quad (0 \leq t \leq d-1) \end{aligned}$$

Proposition 2.3. *The function F^\rightarrow inherits the properties of being monotonic, idempotent and extensive from F .*

Proof. The closure function F^\rightarrow is a composition of the monotonic functions, F and least upper bound, therefore it is monotonic itself.

We prove the idempotency of F^\rightarrow by proving $F^\rightarrow(F^\rightarrow(\sigma))(t) = F^\rightarrow(\sigma)(t)$ by induction on t .

Suppose $t = 0$, then

$$\begin{aligned} &F^\rightarrow(F^\rightarrow(\sigma))(0) \\ &= F(F^\rightarrow(\sigma)(0)) \quad (\text{definition of } F^\rightarrow) \\ &= F(F(\sigma(0))) \quad (\text{definition of } F^\rightarrow) \\ &= F(\sigma(0)) \quad (\text{idempotency of } F) \\ &= F^\rightarrow(\sigma)(0) \quad (\text{definition of } F^\rightarrow) \end{aligned}$$

The induction hypothesis is: $F^\rightarrow(F^\rightarrow(\sigma))(t) = F^\rightarrow(\sigma)(t)$ for a fixed t . Suppose that the induction hypothesis holds, then:

$$\begin{aligned} &F^\rightarrow(F^\rightarrow(\sigma))(t+1) \\ &= F(F^\rightarrow(\sigma)(t+1) \sqcup \text{next}(F^\rightarrow(F^\rightarrow(\sigma))(t))) \quad (\text{definition of } F^\rightarrow) \\ &= F(F^\rightarrow(\sigma)(t+1) \sqcup \text{next}(F^\rightarrow(\sigma)(t))) \quad (\text{ind. hyp.}) \end{aligned}$$

Now we reduce the term $F^\rightarrow(\sigma)(t+1) \sqcup \text{next}(F^\rightarrow(\sigma)(t))$ further.

$$\begin{aligned} &F^\rightarrow(\sigma)(t+1) \sqcup \text{next}(F^\rightarrow(\sigma)(t)) \\ &= F(\sigma(t+1) \sqcup \text{next}(F^\rightarrow(\sigma)(t))) \sqcup \text{next}(F^\rightarrow(\sigma)(t)) \quad (\text{def. } F^\rightarrow) \\ &= F(\sigma(t+1) \sqcup \text{next}(F^\rightarrow(\sigma)(t))) \quad (F \text{ extensive, prop. } \sqcup) \end{aligned}$$

Thus:

$$\begin{aligned} &F^\rightarrow(F^\rightarrow(\sigma))(t+1) \\ &= F(F^\rightarrow(\sigma)(t+1) \sqcup \text{next}(F^\rightarrow(\sigma)(t))) \quad (\text{see above}) \\ &= F(F(\sigma(t+1) \sqcup \text{next}(F^\rightarrow(\sigma)(t)))) \quad (\text{see above}) \\ &= F(\sigma(t+1) \sqcup \text{next}(F^\rightarrow(\sigma)(t))) \quad F \text{ idempotent} \\ &= F^\rightarrow(\sigma)(t+1) \quad (\text{def. } F^\rightarrow) \end{aligned}$$

Finally, F^\rightarrow being extensive follows directly from the definition of F^\rightarrow and the properties of \sqcup . \square

2.4. Semantics for STE. Before giving our semantics for STE we first introduce the concept of *trajectories*:

Trajectories A trajectory is defined as a sequence in which no more information can be derived by forwards propagation. That is, a sequence τ is a trajectory of a closure function when it is a fixed-point of the closure function for sequences. So, a sequence τ is a *trajectory* of F iff $\tau = F^\rightarrow(\tau)$.

STE-assertions have the form $A \implies C$. Here A and C are formulas in *Trajectory Evaluation Logic* (TEL). The only variables in the logic are time-independent Boolean variables taken from the set V of *symbolic constants*. The language is given by the following grammar:

$$f ::= n \text{ is } 0 \mid n \text{ is } 1 \mid f_1 \text{ and } f_2 \mid P \rightarrow f \mid \mathbf{N}f$$

where n is a circuit node and P is a Boolean propositional formula over the set of symbolic constants W . The operator **is** is used to make a statement about the Boolean value of a particular node in the circuit, **and** is conjunction, \rightarrow is used to make conditional statements, and **N** is the next time operator. Note that symbolic constants only occur in the Boolean propositional expressions on the left-hand side of an implication. The notation $n \text{ is } P$, where P is a Boolean symbolic expression over the set of symbolic constants V , is used to abbreviate the formula: $(\neg P \rightarrow n \text{ is } 0) \text{ and } (P \rightarrow n \text{ is } 1)$.

The *depth* of a TEL-formula f is the maximal degree of nestings of **N** in f . The depth of an STE-assertion $A \implies C$ is the maximum of the depth of A and the depth of C .

The meaning of a TEL formula is defined by a satisfaction relation that relates valuations of the symbolic constants and sequences to TEL formulas. Here, the following notation is used: The time shifting operator σ^1 is defined by $\sigma^1(t)(n) = \sigma(t+1)(n)$. Standard propositional satisfiability is denoted by \models_{PROP} . Satisfaction of a trajectory evaluation logic formula f of depth d , by a sequence σ of at least depth d , and a valuation $\phi : W \rightarrow \{0, 1\}$ (written $\phi, \sigma \models f$) is defined by

$$\begin{aligned} \phi, \sigma \models n \text{ is } b &\equiv \sigma(0)(n) = b, \quad b \in \{0, 1\} \\ \phi, \sigma \models f_1 \text{ and } f_2 &\equiv \phi, \sigma \models f_1 \text{ and } \phi, \sigma \models f_2 \\ \phi, \sigma \models P \rightarrow f &\equiv \phi \models_{\text{PROP}} P \text{ implies } \phi, \sigma \models f \\ \phi, \sigma \models \mathbf{N}f &\equiv \phi, \sigma^1 \models f \end{aligned}$$

Semantics for STE We introduce three semantics for STE. They differ in the way that is dealt with the over-constrained value **T**. There are several ways of dealing with this value in a semantics for STE.

First of all, we can treat **T** as a global contradiction. That is, a sequence that gives value **T** to any node, satisfies any antecedent and consequent. So, in order to check whether an STE-assertion holds we need only consider three-valued sequences.

Definition 2.4. A circuit with closure function F satisfies a trajectory assertion $A \implies C$ of depth d , written $F \models A \implies C$, iff for every valuation $\phi : W \rightarrow \{0, 1\}$ of the symbolic constants, and for every three-valued trajectory τ of F of depth d , it holds that:

$$\phi, \tau \models A \Rightarrow \phi, \tau \models C.$$

Secondly, we can treat **T** as a local contradiction. For example, the requirement that a node should have value 1 is fulfilled if the node has value **T**. But other, unrelated requirements are unaffected. We introduce the *simple semantics for STE* using this approach.

Definition 2.5. A circuit with closure function F *simply satisfies* a trajectory assertion $A \implies C$ of depth d , written $F \models_{\text{Simple}} A \implies C$, iff for every valuation $\phi : W \rightarrow \{0, 1\}$ of the symbolic constants, and for every trajectory τ of F of depth d , it holds that:

$$\phi, \tau \models A \implies \phi, \tau \models C.$$

The simple semantics turns out to be useful when we compare the proving power of STE and GSTE in a precise way in Sect. 4. In the simple semantics, it is for example meaningful to talk about what happens in a sequence *before* certain nodes get a value \top forced by an antecedent.

Finally, we can treat \top as an error. That is, if a node is required to have value \top by the antecedent of an STE-assertion, the STE-assertion is not true. This is the default approach taken in Intel’s in-house verification toolkit Forte [3]: it raises an *antecedent failure* if a node is required to have value \top by the antecedent. We call this semantics, the *cautious semantics for STE*.

Definition 2.6. A circuit with closure function F *cautiously satisfies* a trajectory assertion $A \implies C$ of depth d , written $F \models_{\text{Cautious}} A \implies C$, if *both* $F \models A \implies C$ *and* for every valuation ϕ of the symbolic constants there exists a three-valued trajectory τ of depth d such that $\phi, \tau \models A$.

Example 2.7. For an AND-gate with inputs in_1 and in_2 , and output out , the assertion

$$(\text{out is } 1) \text{ and } (\text{in}_1 \text{ is } a) \text{ and } (\text{in}_2 \text{ is } b) \implies (\text{in}_1 \text{ is } 1) \text{ and } (\text{in}_2 \text{ is } 1)$$

is true in the normal semantics but not in the cautious semantics.

For valuations that give at least one of the symbolic constants a and b the value 0, there are no three-valued trajectories that meet the antecedent: there are no three-valued trajectories in which at least one of the inputs of the AND-gate (nodes in_1 and in_2) has value 0, while the output (node out) has value 1. Only for the valuation that gives both the symbolic constants value 1, there exists a three-valued trajectory that satisfies the antecedent. As this trajectory satisfies the consequent as well, the assertion is true in the normal semantics. \square

3. A FAITHFUL SEMANTICS FOR GSTE

In this section, we present an alternative semantics for GSTE. As stated in the introduction, there are two reasons for doing so. First of all, the existing semantics for GSTE [17] are not faithful to the proving power of GSTE algorithms. Therefore, they cannot be used to understand or predict whether certain properties can be proven by GSTE model checkers. Secondly, a faithful semantics for GSTE can be used as basis for research on new GSTE model checking algorithms and other GSTE tools.

The semantics presented in this section is built up in the same way as the semantics for STE in the previous section. First, we introduce the concept of *sequence graphs*. Like sequences in STE, sequence graphs represent circuit behaviour over time.

Then, we define a *closure function for sequence graphs*. Comparable to the closure function for sequences in STE, the closure function for sequence graphs, given a sequence graph, calculates all information that can be derived by forwards propagation of information.

After that, we introduce the concept of *trajectory graphs*. A trajectory graph is a sequence graph in which no more information can be derived by forwards propagation of

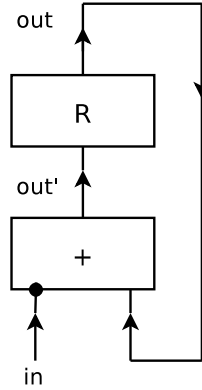


Figure 4: A simple circuit

information. Thus, a sequence graph is a trajectory graph precisely when it is a fixpoint of the closure function for sequence graphs.

Then, we formally define the concept of *assertion graphs*. Examples of assertion graphs are given in the introduction of this paper. An assertion graph describes a property of the behaviour of the circuit, possibly ranging over unbounded time.

Finally, by combining all these concepts, we introduce a faithful semantics for GSTE.

3.1. Sequence Graphs. We introduce the concept of *sequence graphs* to represent circuit behaviours over time. They are comparable to the concept of sequences in STE. Sequence graphs, however, are more expressive: each sequence graph represents a (possibly infinite) number of sequences.

Example 3.1. Consider the circuit given in Figure 4. The following picture represents a sequence graph of the circuit.

$$\text{init} \begin{array}{c} \xrightarrow{11X} \\ \xleftarrow{0XX} \end{array} v \xrightarrow{XXX} w \curvearrowright 000 \tag{3.1}$$

The sequence graph has vertices init, v and w , two edges from init to v , an edge from v to w , and an edge from w to itself. In the picture, states are represented by vectors of truth-values, in the order $\text{in}, \text{out}', \text{out}$. For instance, in the state represented by $11X$, node in has value 1, node out' has value 1, and node out has value X .

Each *path* in the graph starting in initial vertex init , represents a possible behaviour of the circuit over time. For instance, consider the path starting in init , going through the top edge between init and v , and then cycles twice through the looping edge at vertex w . This path represents the sequence

$$[11X, XXX, 000, 000] \quad \square$$

The reader should note the difference between *sequence graphs* and *assertion graphs* (see page 3 for an example of an assertion graph). Sequence graphs represent *circuit behaviour* (corresponding to *sequences* in STE), whereas assertion graphs describe *desired properties* of circuit behaviour (corresponding to *assertions* in STE).

Definition 3.2. A *sequence graph* is a triple (V, E, Σ) , where:

- V is a finite set of vertices containing the *initial vertex* init .
- E is a finite set of directed edges between vertices. Each edge e has a *start vertex* $\text{start}(e)$ and an *end vertex* $\text{end}(e)$. Multiple edges between two vertices are allowed.
- $\Sigma : E \rightarrow \mathbf{State}$ is a function from edges to circuit states.

We say that sequence graphs (V_1, E_1, Σ_1) and (V_2, E_2, Σ_2) are of the *same shape* iff $V_1 = V_2$ and $E_1 = E_2$. The set of all sequence graphs is denoted SeqGraph .

Usually, a sequence graph is identified by the function Σ only.

The order \leq and the operators \sqcup and \sqcap on the domain $\{0, 1, X, \top\}$ are extended in a point-wise fashion to pairs of sequence graphs of the same shape. That is, the order \leq on sequence graphs is defined by $\Sigma_1 \leq \Sigma_2$ iff for all edges e and nodes n , $\Sigma_1(e)(n) \leq \Sigma_2(e)(n)$. Furthermore, $(\Sigma_1 \sqcup \Sigma_2)(e)(n) = (\sigma_1(e)(n) \sqcup \sigma_2(e)(n))$ and $(\Sigma_1 \sqcap \Sigma_2)(n) = (\Sigma_1(e)(n) \sqcap \Sigma_2(e)(n))$.

An edge is *initial* if it starts in the initial vertex init . We define the set of incoming edges of an edge e , written $\text{in}(e)$ by:

$$\text{in}(e) = \{e' \in E \mid \text{start}(e) = \text{end}(e')\}$$

A *path of depth d* is a list of edges $\rho = (e_0, e_1, \dots, e_d)$ such that for each i , $\text{start}(e_{i+1}) = \text{end}(e_i)$. An *initial path* is a path whose first edge is initial.

A finite initial path ρ of depth d in a sequence graph Σ represents the sequence $\mathbf{seq}(\Sigma, \rho)$ of depth d defined by

$$\mathbf{seq}(\Sigma, \rho)(t) = \Sigma(\rho(t)).$$

A sequence graph Σ represents the set of sequences $\mathbf{seq}(\Sigma)$ defined by

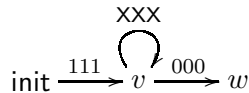
$$\mathbf{seq}(\Sigma) = \{\mathbf{seq}(\Sigma, \rho) \mid \rho \text{ is a finite initial path in } \Sigma\}$$

We will only consider sequence graphs in which each edge and each vertex is reachable from the initial vertex init . That is, we require that for each edge there exists an initial path containing the edge, and for each vertex there exists an initial path containing the vertex. The reason is that states at unreachable edges cannot appear in the sequences represented by the sequence graph.

Example 3.3. The sequence graph (3.1) represents the following infinite set of sequences:

[11X]
 [11X, XXX]
 [11X, XXX, 000]
 [11X, XXX, 000, 000]
 ...
 [0XX]
 [0XX, XXX]
 [0XX, XXX, 000]
 [0XX, XXX, 000, 000]
 ...

The sequence graph



represents the following infinite set of sequences:

- [111]
 - [111, 000]
 - [111, XXX]
 - [111, XXX, 000]
 - [111, XXX, XXX]
 - [111, XXX, XXX, 000]
 - ...
-

3.2. Trajectory Graphs and Closure Functions. We introduce the concept of *trajectory graphs* to represent sequence graphs in which no more information can be derived by forwards propagation of information. Trajectory graphs are comparable to the concept of trajectories in STE.

In order to define trajectory graphs, we define a *closure function for sequence graphs*, written $F^\circ : \text{SeqGraph} \rightarrow \text{SeqGraph}$. The idea is that such a closure function, given a sequence graph, derives all information that can be derived by forwards propagation. Then, a trajectory graph is defined as a fixpoint of this closure function.

Before doing so, let us first get some more intuition on desired properties for a closure function for sequence graphs. Recall that a sequence graph represents a (possibly infinite) collection of sequences. Each initial path ρ in the graph represents a sequence $\text{seq}(\Sigma, \rho)$ as defined before.

Furthermore, recall, from the introduction, when a circuit satisfies a GSTE assertion graph, the circuit should also satisfy all STE-assertion corresponding to finite initial paths in the assertion graphs.

Therefore, given a sequence graph and an initial path ρ in the graph, we expect that the closure function on sequence graphs F° for the edges in ρ derives at most the information as the closure function for sequences F^\rightarrow does for the sequence $\text{seq}(\Sigma, \rho)$. The reason for requiring this is that if the closure function on sequence graphs were to derive more information for a particular sequence in the sequence graph than the closure function on sequences, then we could construct a GSTE assertion graph that is satisfied by the circuit, but that contains a path corresponding to an untrue STE-assertion.

So, we require the following property:

Property 3.4. A closure function F° for sequence graphs *derives no more information* than a closure function on sequences F^\rightarrow , if for all sequence graphs Σ and initial paths ρ ,

$$\text{seq}(F^\circ(\Sigma), \rho) \leq F^\rightarrow(\text{seq}(\Sigma, \rho))$$

The closure function for sequence graphs is allowed to derive *less* information for a particular path than the closure function for sequences does. The reason is that an edge might be reached via different initial paths. If, for these paths, the closure function for sequences derives conflicting values for a circuit node at that edge, the above property forces the circuit node to take on value X. We elaborate on this in Example 3.6 on page 19.

Defining a closure function for sequence graphs is a greater challenge than defining a closure function for sequences. There are two reasons for this.

First of all, in STE, for a state at time $t + 1$, there is precisely one “previous” state, namely the state at time t . So, it is clear how the information from previous points in time

should be propagated. In GSTE, however, a state at an edge can have multiple predecessors. So, we have to decide how to combine information from incoming edges.

Secondly, in STE-sequences, the state at each time-point depends only on the previous states, and, thus, never on itself. In GSTE sequence graphs, however, cycles may be present, therefore a state may, via a cycle, depend on itself.

In the following, we gradually construct a closure function for sequence graphs by considering closure functions for increasingly more complex sets of sequence graphs. First, we only consider *linear* sequence graphs, then we look at *acyclic* sequence graphs, and finally we consider general sequence graphs.

Linear sequence graphs. Let us take one step at the time. So, first, assume we have a sequence graph where each vertex has at most one successor and at most one predecessor, and no cycles are present. Such a sequence graph has the following form:

$$\text{init} \longrightarrow v_1 \longrightarrow v_2 \longrightarrow \dots \longrightarrow v_d$$

For edges that have exactly one incoming edge, we define the function $\text{pre} \{ \text{pre}(e) \} = \text{in}(e)$. (Recall that $\text{in}(e)$ is the set of all incoming edges of e .)

For the above sequence graph, a closure function F_{line}° can be defined in the same way as in STE. For the initial edge, no information is propagated from a previous time-point, so only closure of the initial state is needed. For each other time-point, information from the previous state should be propagated.

This yields the closure function F_{line}° :

$$F_{\text{line}}^\circ(\Sigma)(e) = \begin{cases} F(\Sigma(e)), & e \text{ is initial} \\ F(\Sigma(e) \sqcup \text{next}(F_{\text{line}}^\circ(\Sigma)(\text{pre}(e))), & \text{otherwise} \end{cases}$$

The function F_{line}° is well defined as the graphs we consider here are acyclic and edges have at most one predecessor. Note the similarity with the closure function for sequences F^\rightarrow on page 12.

Note that, just like in STE, the initial values of registers are ignored.

F_{line}° calculates precisely the same information as F^\rightarrow . That is, for each initial path ρ in a sequence graph Σ of the above form,

$$\mathbf{seq}(F_{\text{line}}^\circ(\Sigma), \rho) = F^\rightarrow(\mathbf{seq}(\Sigma, \rho)).$$

Acyclic sequence graphs. Now, let us consider a more general situation: an acyclic graph.

Example 3.5. Consider the following sequence graph:

$$\text{init} \begin{array}{c} \xrightarrow{11X} \\ \xleftarrow{000} \end{array} v \xrightarrow{XXX} w$$

The edge starting at v has two incoming edges. For this edge, the state at the previous time-point can be any of the states at the predecessor edges, that is 11X and 000. The first state gives node out' value 1, so if this state had been the only predecessor state, we would have concluded that node out should have value 1 at the edge starting at v . However, the state at the second incoming edge gives node out' value 0, so according to this state, node out should have value 1 at the edge starting at v . Therefore, as the two incoming edges do not agree on the value of node out' , nothing can be derived about the value of node out at the edge starting in v . So, no more information can be derived from this sequence graph. \square

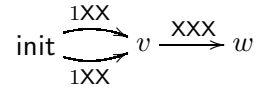
So, only if all the states of the incoming edges agree on a Boolean value of an input to a register, should this value be propagated to the output of the register. Thus, to combine the values on the inputs of the register, the *greatest lower bound* should be used. This yields the closure function F_{nocycle}° :

$$F_{\text{nocycle}}^\circ(\Sigma)(e) = \begin{cases} F(\Sigma(e)), & e \text{ is initial} \\ F(\Sigma(e) \sqcup \bigcap_{i \in \text{in}(e)} \text{next}(F_{\text{nocycle}}^\circ(\Sigma)(i))), & \text{otherwise} \end{cases}$$

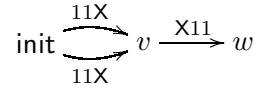
Example 3.6. Applied to the sequence graph in Example 3.5, the closure function yields the same sequence graph. In the graph, the top edge between *init* and *v* gives value 1 to *out'*, the bottom edge gives value 0 to this node, so $0 \sqcap 1 = \mathbf{X}$ is propagated for the value of node *out* for the edge starting in *v*.

In this example, the closure function for sequence graphs derives less information than the closure function for sequences for the paths in the assertions. The reason is that the greatest lower-bound operator is used to combine conflicting information from incoming edges.

Applied to



the closure function yields



As both incoming edges give value 1 to node *out'* this value is propagated to node *out*. \square

General sequence graphs. Now that we have dealt with sequence graphs where edges can have multiple predecessors, it is time to tackle the next challenge: cycles. The following example illustrates that when cycles are present, the equations for F_{nocycle}° no longer define a function, but, instead, may have more than one solution.

Example 3.7. Consider the sequence graph:

$$\text{init} \xrightarrow{1\mathbf{X}\mathbf{X}} v \circlearrowright \mathbf{X}\mathbf{X}\mathbf{X} \quad (3.2)$$

Here, the result for the initial edge still can be calculated (yielding 11X), but the result for the self-loop at vertex *v* is problematic. The equations state:

$$F_{\text{nocycle}}^\circ(\Sigma)((v, v)) = F(\mathbf{X}\mathbf{X}\mathbf{X} \sqcup (\text{next}(11\mathbf{X}) \sqcap \text{next}(F_{\text{nocycle}}^\circ(\Sigma)((v, v)))))$$

This can be simplified to:

$$F_{\text{nocycle}}^\circ(\Sigma)((v, v)) = F(\mathbf{X}11 \sqcap \text{next}(F_{\text{nocycle}}^\circ(\Sigma)((v, v)))),$$

further simplified to:

$$F_{\text{nocycle}}^\circ(\Sigma)((v, v)) = F(\mathbf{X}11 \sqcap \mathbf{X}\mathbf{X}(F_{\text{nocycle}}^\circ(\Sigma)((v, v))(\text{out}'))),$$

and finally simplified to:

$$F_{\text{nocycle}}^\circ(\Sigma)((v, v)) = F(\mathbf{X}\mathbf{X}(1 \sqcap F_{\text{nocycle}}^\circ(\Sigma)((v, v))(\text{out}'))).$$

This equation can be rewritten to:

$$F_{\text{nocycle}}^{\circ}(\Sigma)((v, v)) = (\lambda s.F(\text{XX}(1 \sqcap s((v, v))(\text{out}')))) F_{\text{nocycle}}^{\circ}(\Sigma)((v, v))$$

This equation has as solutions precisely the fixpoints of

$$(\lambda s.F(\text{XX}(1 \sqcap s(\text{out}'))))$$

The two fixed-points are XXX or X11.

The first fixed-point yields the following sequence graph:

$$\text{init} \xrightarrow{11\text{X}} v \circlearrowright \text{XXX} \quad (3.3)$$

This contradicts our intuition: if at the first point in time node *in* has value 1, then we expect that, from the next time-point on, node *out* and *out'* have value 1 as well. So, only the second fixed-point gives the expected sequence graph:

$$\text{init} \xrightarrow{11\text{X}} v \circlearrowright \text{X11} \quad (3.4)$$

□

So, in general, when cycles are introduced, the equations for $F_{\text{nocycle}}^{\circ}$ no longer define a function: the equations may have more than one solution. Let us study this set of solutions more closely.

To do so, we define, for a given sequence graph Σ , the function $F_{\Sigma}^{\circ} : \text{SeqGraph} \rightarrow \text{SeqGraph}$ by:

$$F_{\Sigma}^{\circ}(\Delta)(e) = \begin{cases} F(\Sigma(e)), & e \text{ is initial} \\ F(\Sigma(e) \sqcup \bigsqcap_{i \in \text{in}(e)} \text{next}(\Delta(i))), & \text{otherwise} \end{cases}$$

Using this, the equations for $F_{\text{nocycle}}^{\circ}$ can be rewritten to:

$$F_{\text{nocycle}}^{\circ}(\Sigma) = F_{\Sigma}^{\circ}(F_{\text{nocycle}}^{\circ}(\Sigma))$$

The solutions of this equation are the set of fixpoints of F_{Σ}° . For example, for Σ equal to sequence graph (3.2) the fixpoints are sequence graphs (3.3) and (3.4).

The following lemma states that each fixpoint satisfies Property 3.4.

Lemma 3.8. *For each Δ that is a fixpoint of F_{Σ}° and for each initial path ρ in the sequence graph Σ , it holds that:*

$$\mathbf{seq}(\Delta, \rho) \leq F^{\rightarrow}(\mathbf{seq}(\Sigma, \rho))$$

Proof. The proof is by induction on the position in the sequence. The base-case ($t = 0$) follows directly from the definitions of \mathbf{seq} and F^{\rightarrow} . The induction hypothesis is:

$$\mathbf{seq}(\Delta, \rho)(t) \leq (F^{\rightarrow}(\mathbf{seq}(\Sigma, \rho)))(t)$$

If $\rho(t+1)$ is not initial, then

$$\mathbf{seq}(\Delta, \rho)(t+1) = F(\Sigma(\rho(t+1)) \sqcup \bigsqcap_{i \in \text{in}(\rho(t+1))} \text{next}(\Delta(i)))$$

Now:

$$\begin{aligned} & \bigsqcap_{i \in \text{in}(\rho(t+1))} \text{next}(\Delta(i)) \\ & \leq \text{next}(\Delta(\rho(t))) \quad (\rho(t) \in \text{in}(\rho(t+1))) \\ & = \text{next}(\mathbf{seq}(\Delta, \rho)(t)) \quad (\text{Definition } \mathbf{seq}) \\ & \leq \text{next}(F^{\rightarrow}(\mathbf{seq}(\Sigma, \rho))(t)) \quad ((\text{Induction hypothesis and monotonicity next})) \end{aligned}$$

Thus,

$$\mathbf{seq}(\Delta, \rho)(t+1) \leq F(\mathbf{seq}(\Sigma, \rho)(t+1) \sqcup \text{next}(F^\rightarrow(\mathbf{seq}(\Sigma, \rho))(t)))$$

So, by the definition of F^\rightarrow ,

$$\mathbf{seq}(\Delta, \rho)(t+1) \leq F^\rightarrow(\mathbf{seq}(\Sigma, \rho))(t+1)$$

The case for $\rho(t+1)$ is initial is similar but easier. \square

The question now is: which fixpoint should the closure function for sequence graphs choose? As each fixpoint satisfies Property 3.4, it is sound to choose any of them. The following property states there exists a unique greatest fixpoint.

Proposition 3.9. *For each Σ , the function F_Σ° has a unique greatest fixpoint.*

Proof. It is easy to see that F_Σ° is monotonic. The collection of sequence graphs with the same vertices and edges as Σ and giving values to the same circuit nodes as Σ is finite and forms, together with the order \leq on sequence graphs, a complete lattice. So, by Tarski's fixpoint theorem [13], F_Σ° has a greatest fixpoint. \square

As each fixpoint satisfies Property 3.4, we can safely choose the greatest one, giving the most information. Thus, we define the closure function for sequence graphs as follows:

Definition 3.10. Given a closure function F , the *closure function for sequence graphs*, written $F^\circ : \text{SeqGraph} \rightarrow \text{SeqGraph}$ is defined by:

$$F^\circ(\Sigma) = \text{gfp}\Delta.F_\Sigma^\circ(\Delta)$$

Proposition 3.11. *Given a closure function F , F° is a closure function as well.*

Proof. Suppose F is a closure function, we have to prove that F° is monotonic, extensive and idempotent. F° being extensive follows directly from the definition of F° .

We now prove that F° is monotonic. Suppose $\Sigma_1 \leq \Sigma_2$, $\Delta_1 = F^\circ(\Sigma_1)$ and $\Delta_2 = F^\circ(\Sigma_2)$, then

$$\Delta_1 = F_{\Sigma_1}^\circ(\Delta_1) \leq F_{\Sigma_2}^\circ(\Delta_1)$$

Tarski's fixpoint theorem [13] states that

$$\text{gfp}\Delta.F_{\Sigma_2}^\circ(\Delta) = \sqcup\{\Delta \mid \Delta \leq F_{\Sigma_2}^\circ(\Delta)\}$$

Thus $\Delta_1 \leq \text{gfp}\Delta.F_{\Sigma_2}^\circ(\Delta) = \Delta_2$.

Finally, we prove that F° is idempotent. Suppose $F^\circ(\Sigma) = \Delta$ and $F^\circ(\Delta) = \Delta'$. We need to prove that $\Delta = \Delta'$. By monotonicity of F° follows $\Delta \leq \Delta'$. We prove that $\Delta' \leq \Delta$ by proving that Δ' is a fixpoint of F_Σ° (then, because Δ is the greatest fix-point of F_Σ° , it follows that $\Delta' \leq \Delta$). The case for when e is initial is trivial. Suppose e is not initial.

$$\begin{aligned} & F_\Sigma^\circ(\Delta')(e) \\ &= F(\Sigma(e) \sqcup \bigsqcup_{i \in \text{in}(e)} \text{next}(\Delta'(i))) && \text{(Definition } F_\Sigma^\circ) \\ &= F(\Sigma(e) \sqcup \bigsqcup_{i \in \text{in}(e)} \text{next}(\Delta(i)) \sqcup \bigsqcup_{i \in \text{in}(e)} \text{next}(\Delta'(i))) && \text{(Prop } \sqcup, \Delta \leq \Delta') \\ &= F(F(\Sigma(e) \sqcup \bigsqcup_{i \in \text{in}(e)} \text{next}(\Delta(i))) \sqcup \bigsqcup_{i \in \text{in}(e)} \text{next}(\Delta'(i))) && (F \text{ is closure function}) \\ &= F(\Delta(e) \sqcup \bigsqcup_{i \in \text{in}(e)} \text{next}(\Delta'(i))) && (\Delta \text{ is fixpoint of } F_\Sigma^\circ) \\ &= \Delta'(e) && (\Delta' \text{ is fixpoint of } F_\Delta^\circ) \end{aligned}$$

\square

3.2.1. *Trajectory Graphs.* We define a *trajectory graph* of F as a sequence graph that is a fixpoint of F° .

Definition 3.12. A sequence graph Σ is a *trajectory graph* of a closure function F , if

$$F^\circ(\Sigma) = \Sigma$$

3.3. **Assertion Graphs.** In GSTE, circuit properties are given by *assertion graphs*. An example of an assertion graph is:

$$\text{init} \xrightarrow{\text{in is 1}/\cdot} v \circlearrowright \cdot / \text{out is 1} \quad (3.5)$$

In the assertion graph, each edge is labelled with a pair A/C , here A is called the antecedent and C is called the consequent. Just like in STE, the antecedent represents assumptions made, and the consequent represents requirements.

Both A and C are, like in STE, formulas in trajectory evaluation logic (TEL). However, as each edge represents the state of a single time-point, no occurrences of the next-time operator \mathbf{N} are allowed. We call the subset of TEL in which no next-time operators occur GTEL.

The assertion graph above states that if at some time point, node *in* has value 1, then at each later time-point node *out* has value 1 as well.

Definition 3.13. An assertion graph is a four-tuple $G = (V, E, \text{ant}, \text{cons})$. Here, V is a set of *vertices* containing a vertex *init* which is called the *initial vertex*, E is a set of *edges* between the vertices. Finally, $\text{ant}, \text{cons} : E \rightarrow \text{GTEL}$ are functions from edges to formulas in GTEL.

Recall that path is called initial iff it starts in the initial vertex *init*. A finite initial path ρ of depth d in an assertion graph G represents an STE assertion $\text{Ass}(G, \rho)$ defined by

$$\text{Ass}(G, \rho) = \left(\mathbf{and}_{0 \leq i \leq d} \mathbf{N}^i \text{ant}(\rho(t)) \right) \Rightarrow \left(\mathbf{and}_{0 \leq i \leq d} \mathbf{N}^i \text{cons}(\rho(t)) \right)$$

An assertion graph represents a (possibly infinite) collection of STE-assertions: for each finite initial path ρ in the graph, an STE-assertion $\text{Ass}(G, \rho)$. The set of STE-assertions in assertions graph G , written $\text{Ass}(G)$, is defined by:

$$\text{Ass}(G) = \{ \text{Ass}(G, \rho) \mid \rho \text{ is a finite initial path in } G \}$$

Example 3.14. Assertion graph (3.5) above represents the following infinite set of STE-assertions:

$$\begin{aligned} \text{in is 1} &\Rightarrow \mathbf{N}(\text{out is 1}) \\ \text{in is 1} &\Rightarrow \mathbf{N}(\text{out is 1}) \mathbf{and} \mathbf{NN}(\text{out is 1}) \\ \text{in is 1} &\Rightarrow \mathbf{N}(\text{out is 1}) \mathbf{and} \mathbf{NN}(\text{out is 1}) \mathbf{and} \mathbf{NNN}(\text{out is 1}) \\ &\dots \end{aligned}$$

□

The idea is that when a circuit satisfies a GSTE assertion graph, the circuit graph also satisfies all STE assertions in the assertion graph. The converse, however, does not hold, as we will see in the next section.

3.4. Satisfiability. Satisfaction of a GTEL-formula f , by a circuit state s : **State** and a valuation $\phi : W \rightarrow \{0, 1\}$ of the symbolic constants (written $\phi, s \models f$) is defined by

$$\begin{aligned} \phi, s \models n \text{ is } b &\equiv \sigma(0)(n) = b \text{ , } b \in \{0, 1\} \\ \phi, s \models f_1 \text{ and } f_2 &\equiv \phi, s \models f_1 \text{ and } \phi, s \models f_2 \\ \phi, s \models P \rightarrow f &\equiv \phi \models_{\text{PROP}} P \text{ implies } \phi, s \models f \end{aligned}$$

Example 3.15. If $s(\text{in}) = 1$ and $s(\text{out}) = 0$, and $\phi(a) = 1$ and $\phi(b) = 0$, then

$$\phi, s \models (\text{in is } a) \text{ and } (\text{out is } b) \text{ and } (\text{in is } \neg(a \wedge b)) \quad \square$$

We say that a sequence graph (V, E, Σ) satisfies a function $f : E \rightarrow \text{GTEL}$, $f \in \{\text{ant}, \text{cons}\}$ and a valuation $\phi : W \rightarrow \{0, 1\}$ of the symbolic constants, written $\phi, \Sigma \models f$, if for all edges e :

$$\phi, \Sigma(e) \models f(e)$$

Note that the definition of satisfaction above requires that the shape of the sequence graph be identical to the shape of the assertion graph from which the antecedent or consequent is taken.

Example 3.16. If $G = (V, E, \text{ant}, \text{cons})$ is assertion graph (3.5), Σ_1 is sequence graph (3.2), and Σ_2 is sequence graph (3.4), then for any ϕ : $\phi, \Sigma_1 \models \text{ant}$, $\phi, \Sigma_2 \models \text{ant}$, $\phi, \Sigma_1 \not\models \text{cons}$, and $\phi, \Sigma_2 \models \text{cons}$. \square

Just like in STE, in GSTE, there are several ways of dealing with the over-constrained value T. We can treat T just as any other value, leading to the *simple semantics* of GSTE. Or, we can treat an over-constrained value as an error, leading to the *cautious semantics* of GSTE.

In GSTE, however, we cannot treat T as a contradiction in the same way as we did in STE. The reason is the following. Consider a semantics in which a sequence graph that assigns a T to a circuit node at an edge satisfies any antecedent and consequent. In such a semantics, GSTE assertion graphs containing false STE-assertions may still be true. For example, given a GSTE assertion graph containing a false STE assertion, we can simply add a fresh initial edge with an inconsistent antecedent, making the GSTE assertion true.

If, instead, we require a T value at *each* path in the graph to deem a sequence graph contradictory, this problem does not occur. However, as the implementation of such a semantics in a GSTE model checker seems cumbersome, we will not elaborate on such a semantics further.

In the definition of *simple satisfaction* for GSTE, the value T is treated just like any other value, and models a local conflict of demands made by the assertion. In this paper, we consider this the ‘standard’ semantics for GSTE. As explained in Sect. 4, this turns out to correspond well with what most GSTE algorithms do in practice.

Definition 3.17. We say that a closure function F *simply satisfies* an assertion graph $G = (V, E, \text{ant}, \text{cons})$, written $F \models_{\text{Simple}} G$, if for all assignments of symbolic constants $\phi : W \rightarrow \{0, 1\}$, trajectory graphs Σ ,

$$\Sigma \models \text{ant} \Rightarrow \Sigma \models \text{cons}.$$

Example 3.18. If $G = (V, E, \text{ant}, \text{cons})$ is assertion graph (3.5), and F is the closure function for the circuit in Figure 4, then $F \models_{\text{Simple}} G$.

This can be explained as follows. It is easy to see that, for any ϕ , sequence graph (3.2) is the weakest sequence graph that makes the antecedent of G true. Let us call this

sequence graph Σ . Trajectory graph (3.4) is $F^\circ(\Sigma)$. We claim that $F^\circ(\Sigma)$ is the weakest trajectory graph satisfying *ant*. This can be proven easily. Suppose T is a trajectory graph satisfying *ant*, then $\Sigma \leq T$, so by monotonicity of F° and because T is a fix-point of F° , $F^\circ(\Sigma) \leq F^\circ(T) = T$. Thus, as the weakest trajectory graph satisfying *ant* also satisfies *cons*, all trajectory graphs that satisfy *ant* satisfy *cons* as well. So, $F \models_{\text{Simple}} G$.

In Section 5, we explain that, in the general case, to check whether a circuit simply satisfies an assertion graph, we only have to, for each ϕ , consider the weakest trajectory graph that satisfies the antecedent *ant*. \square

In the definition of cautious satisfaction for GSTE, the value \top is treated as an error.

Definition 3.19. We say that a circuit model F , *cautiously satisfies* an assertion graph $G = (V, E, \text{ant}, \text{cons})$, written $F \models_{\text{Cautious}} G$, if F simply satisfies G and for all assignments of symbolic constants $\phi : W \rightarrow \{0, 1\}$, there exists a trajectory graph Σ of F such that $\Sigma \models \text{ant}$.

The following example illustrates the difference between the two definitions.

Example 3.20. The circuit in Figure 4 simply satisfies the following assertion graph. It does, however, not cautiously satisfy it.

$$\text{init} \xrightarrow{\text{in is } 1/\cdot} v \text{ } \bigcirc \text{ } \text{out is } 0/\text{out is } 1 \quad \square$$

4. COMPARING WITH STE

In this section we compare STE with GSTE. The purpose is to make the relationship between STE and GSTE model checking clear.

The following proposition states that if a closure function satisfies an assertion graph, it simply satisfies all STE-assertions in the assertion graph as well.

Proposition 4.1. *Given an assertion graph $G = (V, E, \text{ant}, \text{cons})$ for a circuit with closure function F :*

$$F \models G \Rightarrow (\text{for all assertions } (A \Longrightarrow C) \in \text{Ass}(G) : F \models_{\text{Simple}} (A \Longrightarrow C))$$

Proof. Suppose $F \models G$, ρ is a finite path of depth d in G , $A \Longrightarrow C = \text{Ass}(G, \rho)$, ϕ a valuation of the symbolic constants, and τ a trajectory of F of depth d such that $\phi, \tau \models A$. We need to prove that $\phi, \tau \models C$.

Let Σ be the sequence graph that has the same shape as assertion graph G and is further defined by:

$$\Sigma(e) = \bigcap_{0 \leq t \leq d, \rho(t)=e} \tau(t)$$

Note that $\Sigma(e)(n) = \top$ for edges not in the path ρ . We now prove that $\phi, \Sigma \models \text{ant}$. As $\tau \models A$, and $A = \mathbf{and}_{0 \leq t \leq d} \mathbf{N}^t \text{ant}(\rho(t))$, for each t holds:

$$\phi, \tau(t) \models \text{ant}(\rho(t))$$

Thus for all $e \in E$:

$$\phi, \bigcap_{t \in \mathbf{N}, \rho(t)=e} \tau(t) \models \text{ant}(e)$$

Thus, $\phi, \Sigma \models \text{ant}$. As F° is extensive, $\phi, F^\circ(\Sigma) \models \text{ant}$ as well. As $F^\circ(\Sigma)$ is a trajectory graph, and $F \models G$, it holds that $\phi, F^\circ(\Sigma) \models \text{cons}$. By Property 3.4:

$$\mathbf{seq}(F^\circ(\Sigma), \rho) \leq F^\rightarrow(\mathbf{seq}(\Sigma, \rho))$$

Thus:

$$\mathbf{seq}(F^\circ(\Sigma), \rho) \leq F^\rightarrow(\tau) = \tau$$

Now, as $F^\circ(\Sigma) \models \text{cons}$, for all $e \in E$:

$$\phi, \Sigma(e) \models \text{cons}(e)$$

Thus:

$$\phi, \bigsqcap_{0 \leq t \leq d, \rho(t)=e} \tau(t) \models \text{cons}(\rho(t))$$

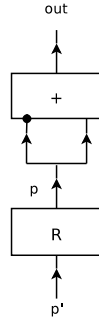
As $C = \mathbf{and}_{0 \leq i \leq d} \mathbf{N}^i \text{cons}(\rho(t))$, it follows that $\phi, \tau \models C$. □

The converse

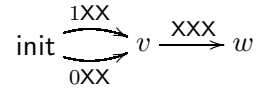
$$(\text{for all assertions } (A \implies C) \in \text{Ass}(G) : F \models_{\text{Simple}} (A \implies C)) \implies F \models G$$

however, is not true. The reason is that GSTE combines conflicting information between incoming edges by using the greatest lower bound operator. This is illustrated by the following example.

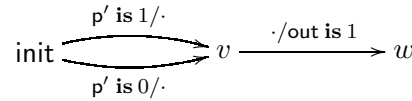
Example 4.2. Consider the following circuit:



The induced closure function of this circuit satisfies the STE-assertions $\mathbf{p}' \text{ is } 1 \implies \mathbf{N}(\text{out is } 1)$ and $\mathbf{p}' \text{ is } 0 \implies \mathbf{N}(\text{out is } 1)$. Consider the following sequence graph. In the picture, states are represented by vectors of truth-values, in the order $\mathbf{p}', \mathbf{p}, \text{out}$.

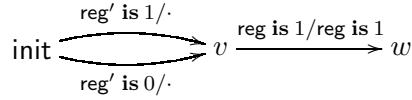


The sequence graph is a trajectory graph of the closure function. Thus, the closure function does not satisfy the below GSTE assertion graph.



The following example shows that if a GSTE assertion graph is cautiously satisfied (that is, no node has to assume value \top to satisfy the antecedent), there may still be an STE-assertion represented by the assertion graph that is not cautiously satisfied.

Example 4.3. Consider a circuit consisting of a single register with input reg' and output reg , and the following assertion graph.



The induced closure function of the circuit satisfies the assertion graph. The reason is that the two incoming edges at vertex v disagree on the value of node reg' , therefore the value X is propagated to the outgoing edge of vertex v . The outgoing edge of vertex v requires reg to have value 1, therefore the consequent at that edge is satisfied. The antecedent does not force any node take on value T , so the assertion graph is cautiously satisfied.

But, the STE-assertion corresponding to the bottom initial path

$$(\text{reg}' \text{ is } 0) \text{ and } \mathbf{N}(\text{reg is } 1) \implies \mathbf{N}(\text{reg is } 1)$$

is not cautiously satisfied as every trajectory that satisfies the antecedent gives node reg value T at time 1. \square

5. GSTE MODEL CHECKING

In [16, 15, 18, 17] model checking algorithms for GSTE are described. In this section, we show the correspondence between the GSTE semantics presented in this paper and a standard model checking algorithm. We do this by first relating our semantics to a GSTE algorithm designed by ourselves, which uses a non-standard fixpoint computation. We proceed by showing that our algorithm computes the same result as the algorithm presented in [17].

Furthermore, as we are concerned with precisely describing abstraction only, we ignore extensions of GSTE algorithms such as backwards information flow and fairness constraints.

5.1. Fundamental theorem of GSTE. Comparable to STE, GSTE model checking is based on the following: Instead of checking that for every trajectory graph, the antecedent implies the consequent, a unique weakest trajectory graph satisfying the antecedent is calculated. We call this graph the *defining trajectory graph*.

To check whether a circuit simply satisfies an assertion graph, it suffices to check whether the defining trajectory graph satisfies the consequent part of the assertion graph.

Before giving the definition of the defining trajectory graph, we first introduce the concept of the *defining sequence graph*. The defining sequence graph of an antecedent is the unique weakest sequence satisfying the antecedent and is defined as follows.

Given an antecedent function $\text{ant} : E \rightarrow \text{GTEL}$, and an assignment of symbolic constants ϕ , we define the *defining sequence graph* of ant and ϕ , written $\phi[\text{ant}]$ by:

$$\phi[\text{ant}](e) = \phi[\text{ant}(e)]_{\text{state}}$$

where

$$\begin{aligned} \phi[m \text{ is } b]_{\text{state}}(n) &= \begin{cases} b, & \text{if } m=n \\ \mathbf{X}, & \text{otherwise} \end{cases} \\ \phi[f_1 \text{ and } f_2]_{\text{state}} &= \phi[f_1]_{\text{state}} \sqcup \phi[f_2]_{\text{state}} \\ \phi[P \rightarrow f]_{\text{state}} &= \begin{cases} \phi[f]_{\text{state}}, & \text{if } \phi \models P \\ \mathbf{X}, & \text{otherwise} \end{cases} \end{aligned}$$

Proposition 5.1. $\phi[ant]$ is the weakest sequence graph satisfying ant and ϕ .

Proof. Trivial, by considering one edge at the time and induction on the structure of the antecedent at that edge. \square

Given an antecedent function $ant : E \rightarrow \text{GTEL}$, a closure function F , and an assignment of symbolic constants ϕ , we define the *defining trajectory graph* of ant , F and ϕ , written $\phi_F[[ant]]$ by:

$$\phi_F[[ant]] = F^\circ(\phi[ant])$$

Proposition 5.2. $\phi_F[[ant]]$ is the weakest trajectory graph satisfying ant .

Proof. From F° being extensive, it follows directly that $\phi[ant] \leq F^\circ(\phi[ant])$, so $\phi_F[[ant]] \models ant$.

Suppose T is a trajectory graph satisfying ant , then $\phi[ant] \leq T$. From monotonicity of F° , it follows that $F^\circ(\phi[ant]) \leq F^\circ(T)$. As T is a fixpoint of F° it follows that $F^\circ(\phi[ant]) \leq T$. \square

Theorem 5.3 (Fundamental Theorem of GSTE). *For each closure function F , assignment of symbolic constants ϕ , and assertion graph $G = (V, E, ant, cons)$,*

$$(\phi[cons] \leq \phi_F[[ant]]) \Leftrightarrow F \models_{\text{Simple}} G$$

Proof. Directly from Proposition 5.2. \square

The fundamental theorem of GSTE states that to check whether a circuit with closure function F satisfies an assertion graph, we only have to check that, for each ϕ , the defining trajectory graph of F satisfies the consequent.

5.2. GSTE Algorithm. The GSTE algorithm calculates a symbolic representation of the defining trajectory graph of an antecedent. Then, it checks whether this symbolic defining trajectory graph meets the consequent.

We first present a scalar version of the algorithm.

5.2.1. A scalar GSTE-algorithm. In (our version of the) scalar GSTE-algorithm, the defining trajectory graph of the antecedent is calculated using the constructive version of Tarski's fixpoint theorem [13].

Proposition 5.4. *For each Σ , the greatest fixpoint of the function F_Σ° is equal to limit Δ_*^Σ of the sequence $\Delta_k^\Sigma = (F_\Sigma^\circ)^k(\mathbb{T})$. Here, \mathbb{T} represents the sequence graph with the same edges and vertices as Σ that gives value \mathbb{T} to each circuit node at each edge.*

Proof. A function is continuous if for all sequence d_0, d_1, d_2, \dots such that $d_{i+1} \leq d_i$ holds:

$$f(\bigsqcup_{k \in \mathbb{N}} d_k) = \bigsqcup_{k \in \mathbb{N}} f(d_k)$$

The constructive version of Tarski's fixpoint theorem [13] states that the greatest fixpoint of a monotone and continuous function f on a complete lattice is given by: $\prod_{k \in \mathbb{N}} f^k(\top)$.

We will use this version of Tarski's fixpoint theorem to prove the proposition. First, we prove that each monotonic function on a finite domain is also continuous. Suppose f is continuous on a finite domain, and d_0, d_1, d_2, \dots is a sequence such that $d_{i+1} \leq d_i$, then the sequence has a fixpoint d_* , thus:

$$f(\bigsqcup_{k \in \mathbb{N}} d_k) = f(d_*)$$

By monotonicity of f , also the sequence $f(d_0), f(d_1), f(d_2), \dots$ is increasing, so the sequence has the fix-point $f(d_*)$ as well. So:

$$\bigsqcup_{k \in \mathbb{N}} f(d_k) = f(d_*)$$

Therefore, F_Σ° is both monotone and continuous. Thus:

$$\text{gfp} \Delta.F_\Sigma^\circ(\Delta) = \prod_{k \in \mathbb{N}} (F_\Sigma^\circ)^k(\top) = \prod_{k \in \mathbb{N}} \Delta_k^\Sigma$$

We prove by induction on k that $\Delta_{k+1}^\Sigma \leq \Delta_k^\Sigma$ for each k .

The case for $k = 0$ is trivial. The induction hypothesis is $\Delta_{k+1}^\Sigma \leq \Delta_k^\Sigma$. We prove that $\Delta_{k+2}^\Sigma \leq \Delta_{k+1}^\Sigma$. The case for e is initial is trivial. Suppose e is not initial. Then,

$$\begin{aligned} & \Delta_{k+2}^\Sigma(e) \\ &= F_\Sigma^\circ(\Delta_{k+1}^\Sigma)(e) && \text{(Definition } \Delta_{k+2}^\Sigma) \\ &= F(\Sigma(e) \sqcup \prod_{i \in \text{in}(e)} \text{next}(\Delta_{k+1}^\Sigma(i))) && \text{(Definition } F_\Sigma^\circ) \\ &\leq F(\Sigma(e) \sqcup \prod_{i \in \text{in}(e)} \text{next}(\Delta_k^\Sigma(i))) && \text{(Induction Hypothesis)} \\ &= F_\Sigma^\circ(\Delta_{k+1}^\Sigma)(e) && \text{(Definition } F_\Sigma^\circ) \\ &= \Delta_{k+1}^\Sigma(e) && \text{(Definition } \Delta_{k+1}^\Sigma) \end{aligned}$$

So, the sequence $\Delta_0^\Sigma, \Delta_1^\Sigma, \Delta_2^\Sigma$ will eventually reach a fixpoint Δ_*^Σ .

Thus:

$$\text{gfp} \Delta.F_\Sigma^\circ(\Delta) = \Delta_*^\Sigma \quad \square$$

Definition 5.5 (Scalar GSTE-algorithm). Given an assertion graph G , and a closure function F , the scalar GSTE-algorithm calculates for every ϕ the defining trajectory graph $\phi_F[[\text{ant}]]$ by calculating $\Delta_*^{\phi[[\text{ant}]]}$ and checks whether

$$\phi[[\text{cons}]] \leq \phi_F[[\text{ant}]]$$

If this check fails for any ϕ the algorithm returns False, otherwise it returns True.

Proposition 5.6. *The scalar algorithm is sound and complete with respect to the presented semantics for GSTE.*

Proof. Directly from the fundamental theorem of GSTE and Proposition 5.4. □

Comparing with earlier presentation In [17] the fixpoint is calculated in a slightly different way. If we adjust the presentation to use the closure function F instead of a transition relation, the following sequence is defined for a given sequence graph Σ :

$$\begin{aligned}\Gamma_0^\Sigma(e) &= \begin{cases} F(\Sigma(e)), & e \text{ is an initial edge} \\ \top, & \text{otherwise} \end{cases} \\ \Gamma_{k+1}^\Sigma(e) &= \Gamma_k^\Sigma(e) \sqcap F(\Sigma(e) \sqcup \bigsqcap_{i \in \text{in}(e)} \text{next}(\Delta_k(i)))\end{aligned}$$

Proposition 5.7. *For each Σ , $\Gamma_*^\Sigma = \Delta_*^\Sigma$.*

Proof. By definition,

$$\begin{aligned}\Delta_0^\Sigma(e)(n) &= \top \\ \Delta_{k+1}^\Sigma(e) &= \begin{cases} F(\Sigma(e)), & e \text{ is initial} \\ F(\Sigma(e) \sqcup \bigsqcap_{i \in \text{in}(e)} \text{next}(\Delta_k(i))), & \text{otherwise} \end{cases}\end{aligned}$$

We prove by induction on k that for each initial edge e , for each k , $\Gamma_k^\Sigma = F(\Sigma(e))$. The base case is trivial. Now suppose for each initial edge e , $\Gamma_k^\Sigma(e) = F(\Sigma(e))$, then for an arbitrary initial edge e :

$$\begin{aligned}\Gamma_{k+1}^\Sigma(e) &= \Gamma_k^\Sigma(e) \sqcap F(\Sigma(e) \sqcup \dots) \quad (\text{Definition } \Gamma_{k+1}^\Sigma) \\ &= F(\Sigma(e)) \sqcap F(\Sigma(e) \sqcup \dots) \quad (\text{Induction Hypothesis}) \\ &= F(\Sigma(e)) \quad (\text{Property } \sqcup, \sqcap)\end{aligned}$$

So, for each initial edge and $k > 0$, $\Gamma_k^\Sigma(e) = \Delta_k^\Sigma(e)$.

We prove by induction on k that for each non-initial edge e , for each $k > 0$,

$$\Gamma_k^\Sigma(e) = \Delta_k^\Sigma(e)$$

In the base-case, k is equal to 1,

$$\begin{aligned}\Gamma_1^\Sigma(e) &= \Gamma_0^\Sigma(e) \sqcap F(\Sigma(e) \sqcup \bigsqcap_{i \in \text{in}(e)} \text{next}(\Gamma_0(i))) \quad (\text{Definition } \Gamma_{k+1}^\Sigma) \\ &= \top \sqcap F(\Sigma(e) \sqcup \bigsqcap_{i \in \text{in}(e)} \text{next}(\top(i))) \quad (\text{Definition } \Gamma_0^\Sigma) \\ &= F(\Sigma(e) \sqcup \bigsqcap_{i \in \text{in}(e)} \text{next}(\top(i))) \quad (\text{Property } \sqcap) \\ &= F(\Sigma(e) \sqcup \bigsqcap_{i \in \text{in}(e)} \text{next}(\Delta_0(i))) \quad (\text{Definition } \Delta_0^\Sigma) \\ &= \Delta_1^\Sigma \quad (\text{Definition } \Delta_1^\Sigma)\end{aligned}$$

Now suppose $\Gamma_k^\Sigma(e) = \Delta_k^\Sigma(e)$, then:

$$\begin{aligned}\Gamma_{k+1}^\Sigma(e) &= \Gamma_k^\Sigma(e) \sqcap F(\Sigma(e) \sqcup \bigsqcap_{i \in \text{in}(e)} \text{next}(\Gamma_k(i))) \quad (\text{Definition } \Gamma_{k+1}^\Sigma) \\ &= \Delta_k^\Sigma(e) \sqcap F(\Sigma(e) \sqcup \bigsqcap_{i \in \text{in}(e)} \text{next}(\Delta_k(i))) \quad (\text{Induction Hypothesis}) \\ &= \Delta_k^\Sigma(e) \sqcap \Delta_{k+1}^\Sigma(e) \quad (\text{Definition } \Delta_{k+1}^\Sigma(e)) \\ &= \Delta_{k+1}^\Sigma(e) \quad (\text{Property } \sqcap, \Delta_{k+1}^\Sigma(e) \leq \Delta_k^\Sigma(e))\end{aligned}$$

So, for each non-initial edge and $k > 0$, $\Gamma_k^\Sigma(e) = \Delta_k^\Sigma(e)$. So, for $k > 0$, $\Gamma_k^\Sigma = \Delta_k^\Sigma$. Thus, $\Gamma_*^\Sigma = \Delta_*^\Sigma$. \square

5.2.2. *A symbolic GSTE-algorithm.* In actual implementations of GSTE, the above algorithm is implemented symbolically. That is, instead of calculating the defining trajectory graph for a specific valuation ϕ , it calculates, using BDDs, a symbolic defining trajectory graph in terms of the symbolic constants in ϕ .

Then, a BDD is constructed that specifies under which conditions on the symbolic constants the symbolic defining trajectory graph satisfies the consequent. If this BDD is equal to the logical constant True, the property is proven. Otherwise, the BDD indicates for which valuations of the symbolic constants the antecedent does not imply the consequent.

6. FUTURE WORK

Extension to the semantics for GSTE. There exist several extensions of the GSTE algorithm that considerably improve the algorithm's proving power. Examples of such extensions are *precise nodes* [18, 15] and *knots* [7]. We would like to give semantic characterisations of these extensions.

In [17, 16, 18], a backwards algorithm for GSTE is described. Using this algorithm properties can be proven that depend on a backwards (that is, from outputs to inputs, and from time $t + 1$ to t) information flow. In [17, 18] a semantics for this form of GSTE is given. The semantics is however not faithful as the algorithm is incomplete w.r.t. the semantics [17]. A faithful semantics for this form of GSTE could be a topic of future work.

SAT-based GSTE model checking. The model checking algorithms for GSTE described in the current literature are based on BDDs. In previous work we described how a faithful semantics for STE [9] enabled us to construct a new SAT-based model checking algorithm for STE [8]. In the same way, our faithful semantics for GSTE could be used to construct a SAT-based model checking algorithm for GSTE. The aim would be to create a tool very much like *satGSTE* [14] that actually respects the GSTE semantics, so that it can possibly find *all* counter examples. In this way, the tool could be used seamlessly in conjunction with a GSTE model checker.

Monitor circuits for GSTE assertion graphs. In [4, 7] methods for automatic construction of monitor circuits for GSTE assertion graphs are described.

The papers explain how monitor circuits can be used to quickly debug and refine GSTE specifications before trying to use, more labour intensive, GSTE model checking.

The monitor circuits implement the \forall -semantics for GSTE. However, as explained in this paper, the GSTE model checking algorithms are not faithful to this semantics. Therefore, monitor circuits cannot be used to debug and refine assertion graphs that are true in the \forall -semantics, but yield a spurious counter-example when trying to prove them with a GSTE model checker. Future work could consist of constructing monitor circuits that can be used to debug and refine assertion graphs in this class. Here, the faithful semantics for GSTE can be used as a starting point.

Reasoning about GSTE assertion graphs. Using the construction of monitor circuits for GSTE assertion graphs, [5] describes two algorithms that can be used in compositional verification using GSTE. The first algorithm decides whether one assertion graph implies another. The second algorithm can be used to model check an assertion graph under the assumption that another assertion graph is true.

The algorithms, and the corresponding soundness and completeness proofs, are based on the \forall -semantics. Therefore, as the algorithms are based on GSTE model checking, the methods are incomplete when abstraction is used. A possible direction for future work is explaining how the GSTE abstraction affects the completeness of the algorithms.

7. CONCLUSION

The semantics for GSTE given in [17, 18] are not faithful to the proving power of GSTE model checking algorithms, that is, the algorithms are incomplete with respect to the semantics. The reason is that the semantics do not capture the abstraction used in GSTE precisely.

The abstraction used in GSTE makes it hard to understand why a specific property can, or cannot, be proven by GSTE. The semantics mentioned above cannot help the user in doing so. So, in the current situation, users of GSTE often have to revert to the GSTE algorithm to understand why a property can or cannot be proven by GSTE.

In this paper, we have presented a semantics for GSTE that is faithful to the proving power of the main GSTE model checking algorithm. We believe that this semantics is an important contribution to the research on GSTE for at least two reasons.

First of all, a faithful semantics makes GSTE more accessible to novice users: a faithful semantics enables users to understand the abstraction used in GSTE, without having to understand the details of the model checking algorithm.

Furthermore, a faithful semantics for GSTE can be used as basis for research on new GSTE model checking algorithms and other GSTE tools. To illustrate this, in previous work [8], we described a new SAT-based model checking algorithm for STE and proven that it is sound and complete w.r.t. to our faithful semantics for STE presented in [9]. Without a faithful semantics for STE, we would have been forced to prove the correctness of our algorithm by relating it to other model checking algorithms for STE. This is clearly a more involved and less elegant approach. In fact, we believe that without constructing a faithful semantics for STE first, we would not have obtained the level of understanding of STE needed to develop the new SAT-based model checking algorithm.

In the same way, we expect that the faithful semantics for GSTE presented in this paper will open the door for new research on GSTE model checking algorithms and other GSTE tools.

Acknowledgements. Thanks to Tom Melham, Mary Sheeran, Rachel Tzoref, and the anonymous referees for commenting on earlier drafts of this paper.

REFERENCES

- [1] Mark Aagaard, Robert B. Jones, Thomas F. Melham, John W. O’Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2000.

- [2] Mark Aagaard and John W. O’Leary, editors. *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*, volume 2517 of *Lecture Notes in Computer Science*. Springer, 2002.
- [3] FORTE.
<http://www.intel.com/software/products/opensource/tools1/verification>.
- [4] Alan J. Hu, Jeremy Casas, and Jin Yang. Efficient generation of monitor circuits for GSTE assertion graphs. In *International Conference on Computer-Aided Design (ICCAD)*, pages 154–160. IEEE Computer Society / ACM, 2003.
- [5] Alan J. Hu, Jeremy Casas, and Jin Yang. Reasoning about GSTE assertion graphs. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2003.
- [6] Thomas F. Melham and Robert B. Jones. Abstraction by symbolic indexing transformations. In Aagaard and O’Leary [2], pages 1–18.
- [7] Kelvin Ng, Alan J. Hu, and Jin Yang. Generating monitor circuits for simulation-friendly GSTE assertion graphs. In *International Conference on Computer Design (ICCD)*, pages 409–416. IEEE Computer Society, 2004.
- [8] Jan-Willem Roorda and Koen Claessen. A new SAT-based algorithm for symbolic trajectory evaluation. In Dominique Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*, pages 238–253. Springer, 2005.
- [9] Jan-Willem Roorda and Koen Claessen. Explaining symbolic trajectory evaluation by giving it a faithful semantics. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *International Computer Science Symposium in Russia (CSR)*, volume 3967 of *Lecture Notes in Computer Science*, pages 555–566. Springer, 2006.
- [10] Thomas Schubert. High level formal verification of next-generation microprocessors. In *Design Automation Conference (DAC)*, pages 1–6. ACM, 2003.
- [11] Roberto Sebastiani, Eli Singerman, Stefano Tonetta, and Moshe Y. Vardi. GSTE is partitioned model checking. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 229–241. Springer, 2004.
- [12] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2), 1995.
- [13] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific J. of Mathematics*, 5:285–309, 1955.
- [14] Jin Yang, Rami Gil, and Eli Singerman. satGSTE: Combining the abstraction of GSTE with the capacity of a SAT solver. In *Designing Correct Circuits (DCC), A satellite event of the ETAPS 2004 group of conferences*, 2004.
- [15] Jin Yang and Amit Goel. GSTE through a case study. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *International Conference on Computer-Aided Design (ICCAD)*, pages 534–541. ACM, 2002.
- [16] Jin Yang and C.-J. H. Seger. Introduction to generalized symbolic trajectory evaluation. In *International Conference on Computer Design (ICCD)*, pages 360–367, Washington - Brussels - Tokyo, 2001. IEEE.
- [17] Jin Yang and Carl Seger. Generalized symbolic trajectory evaluation. Unpublished draft, 2001.
- [18] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In Aagaard and O’Leary [2], pages 70–87.