Newcastle University e-prints

Date deposited: 10th September 2010


Peer Review Status: Peer reviewed

Citation for published item:
Khomenko V. Logic Decomposition of Asynchronous Circuits Using STG Unfoldings. Newcastle upon Tyne: School of Computing Science, University of Newcastle upon Tyne, 2010

Further information on publisher website

Publishers copyright statement:

Use Policy:

The full-text may be used and/or reproduced and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not for profit purposes provided that:

- A full bibliographic reference is made to the original source
- A link is made to the metadata record in Newcastle E-prints
- The full text is not changed in any way.

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Robinson Library, University of Newcastle upon Tyne, Newcastle upon Tyne.
NE1 7RU.
Tel. 0191 222 6000
Logic Decomposition of Asynchronous Circuits Using STG Unfoldings

Victor Khomenko
Logic Decomposition of Asynchronous Circuits Using STG Unfoldings

V. Khomenko

Abstract

A technique for logic decomposition of asynchronous circuits which works on STG unfolding prefixes rather than state graphs is proposed. It retains all the advantages of the state space based approach, such as the possibility of multiway acknowledgement, latch utilisation and highly optimised circuits. Moreover, it significantly alleviates the state space explosion, and thus has superior memory consumption and runtime.
Abstract

A technique for logic decomposition of asynchronous circuits which works on STG unfolding prefixes rather than state graphs is proposed. It retains all the advantages of the state space based approach, such as the possibility of multiway acknowledgement, latch utilisation and highly optimised circuits. Moreover, it significantly alleviates the state space explosion, and thus has superior memory consumption and runtime.

About the author

Victor Khomenko obtained his MSc with distinction in Computer Science, Applied Mathematics and Teaching of Mathematics and Computer Science in 1998 from Kiev Taras Shevchenko University, and PhD in Computing Science in 2003 from University of Newcastle upon Tyne. He is a Program Committee Chair for the International Conference on Application of Concurrency to System Design (ACSD’10). He also organised the Workshop on UnFOLDing and partial order techniques (UFO’07) and Workshop on BALSA Re-Synthesis (RESYN'09). From September 2005 Victor is a Royal Academy of Engineering/EPSRC Post-doctoral Research Fellow, working on the Design and Verification of Asynchronous Circuits (DAVAC) project. Victor’s main interests include model checking of Petri nets, Petri net unfolding techniques, verification and synthesis of self-timed (asynchronous) circuits.

Suggested keywords

LOGIC DECOMPOSITION
UNFOLDING
ASYNCHRONOUS CIRCUITS
SAT
STG
Logic Decomposition of Asynchronous Circuits Using STG Unfoldings

Victor Khomenko

Abstract—A technique for logic decomposition of asynchronous circuits which works on STG unfolding prefixes rather than state graphs is proposed. It retains all the advantages of the state space based approach, such as the possibility of multiway acknowledgement, latch utilisation and highly optimised circuits. Moreover, it significantly alleviates the state space explosion, and thus has superior memory consumption and runtime.

I. INTRODUCTION

ASYNCHRONOUS circuits (ACs) are circuits without clocks. This is a promising type of digital circuits, as they often have lower power consumption and electro-magnetic emission, no problems with clock skew and related subtle issues, and are fundamentally more tolerant of voltage, temperature and manufacturing process variations. The International Technology Roadmap for Semiconductors report on Design [1, Table DESN4] predicts that 22% of the designs will be driven by handshake clocking (i.e. asynchronous) in 2013, and this percentage will raise up to 40% in 2020.

Though the listed advantages look rather attractive in the view of the current and anticipated microelectronics design challenges, correct and efficient ACs are notoriously difficult to synthesise. This paper tackles the problem of logic decomposition of ACs, i.e. the problem of decomposing large logic gates into smaller ones without introducing hazards. This is arguably one of the most complicated problems in the design flow.

We focus on an important subclass of ACs, called speed-independent (SI) circuits; this model follows the classical Muller’s approach [2] and regards each gate as an atomic evaluator of a Boolean function, with a delay element associated with its output. In the SI framework this delay is unbounded, i.e. the circuit must work correctly regardless of its gates’ delays, and the wires are assumed to have negligible delays (or, alternatively, wire forks are assumed to be isochronic — in such a case the circuit is often referred to as quasi-delay-insensitive (QDI); for the purposes of this paper, these two models are indistinguishable). Signal Transition Graphs (STGs) [3], [4] are a formalism for the specification of such circuits. They are Petri nets in which transitions are labelled with the rising and falling edges of circuit signals, see the example in Fig. 1(a–c).

It should be noted that logic decomposition of ACs is considerably more complicated than the corresponding problem in synchronous flows. In the traditional synchronous case the problem can be formulated on a multi-level combinational Boolean network, which should be mapped to a given gate library by applying the conventional Boolean methods (in particular, algebraic or Boolean division). During this process, the existing algorithms try to minimise some cost function that takes into account the estimated area and/or delay (sometimes other metrics such as power consumption are also used).

When moving to ACs, several levels of complexity are added to the described setup. First of all, the problem can no longer be formulated as a combinational optimisation, and one has to deal with a sequential circuit. Second, it is no longer possible to break up a complex-gate into several smaller ones, together computing the same function, as hazards can easily be introduced in this way. (Note that in synchronous circuits such hazards also occur, but are filtered out by the clock.) In fact, some STGs which are implementable by complex-gates can be not implementable in an SI way in some fixed gate library (e.g. one comprising only one- and two-input gates). Finally, gate libraries commonly contain latches, and the best ACs can often be obtained only by utilising them.

The described issues are illustrated by means of an often used example of a simplified VME bus controller shown in Fig. 1 (see also [5, Chapter 2]). Assuming that the encoding conflicts have already been resolved by insertion of an additional internal signal \( csc \) (this task should have been accomplished in the preceding stages of the design flow), one can start with the STG shown in Fig. 1(c). The complex-gate implementation shown in Fig. 1(d) can be synthesised from this STG. Assuming that the gate library contains only one- and two-input gates and latches, there is a problem of decomposing the complex-gate implementing \( csc \) into smaller gates.

Unfortunately, it turns out that the naïve logic decomposition shown in Fig. 1(e) is not SI and even violates the original specification (though it would be acceptable in the synchronous framework). Indeed, consider the following sequence of events:

\[
\begin{array}{cccccc}
\circ & dsr^+ & csc^+ & lds^+ & ltdack^+ & d^+ & dtack^+ & dsr^- \\
& csc^- & d^- & dtack^- & dsr^+ & (2) & csc^+ & d^+
\end{array}
\]

At the initial state marked \( \circ \), \( x = 1 \) and all the other signals are 0. At the state marked \( (1) \), \( x^- \) becomes enabled. Since an SI circuit should work regardless of the gate delays, one has to allow that the gates implementing \( x \) and \( lds \) may be relatively slow, and so the rest of the shown sequence is feasible. When the state marked \( (2) \) is reached, \( csc^+ \) becomes enabled —
something not expected in the STG in Fig. 1(c). Though $\text{csc}$ is an internal signal which is not observable by the environment, this malfunction can propagate to an observable output by producing an unexpected $d^+$. Note that the difference between the complex-gate implementation in Fig. 1(d) and this naive implementation is that in the latter the gate implementing $x$ is allowed to have an arbitrary delay, while in the former it is ‘inside’ a complex-gate which is assumed to be atomic (and thus has no internal delays).

A correct decomposition into two-input gates is shown in Fig. 1(g). Note that in contrast to the described hazardous solution, the new internal signal $\text{map}$ is acknowledged by two gates (see [6], [7] for the concept of acknowledgement). This illustrates the concept of multway acknowledgement, when different transitions of a signal can be acknowledged by different gates; e.g. in this example $\text{map}^+$ is acknowledged by $\text{csc}^+$ and $\text{map}^-$ by $d^-$ (as opposed to the simple case of local acknowledgement, where the newly inserted signal is acknowledged only by the gate being decomposed; unlike the synchronous case, where the local acknowledgement is sufficient for decomposition, and the multiway one is used only for optimisation purposes, in the asynchronous case it is quite common that a complex-gate is not decomposable using local acknowledgements only, but is decomposable using multiway ones). This transformation is rather not obvious at the circuit level; the corresponding STG shown in Fig. 1(f) is much easier to understand.

An implementation utilising a latch is shown in Fig. 1(h), where Muller’s C-element [2] with the next-state function $[c] = \overline{ab} \lor \overline{c(a \lor b)}$ is used. If this latch is present in the library, this implementation is likely to be superior in terms of area and performance to the one in Fig. 1(g). However, this transformation is also non-trivial, and is only possible due to the fact that there is no globally reachable state at which $\text{d} = \text{ldtack} = 0$ and $\text{csc} = 1$ (this is the condition when the complex-gate in Fig. 1(d) and the C-element in Fig. 1(h) behave differently); i.e. a global analysis of the state space of the specification is required, which also takes into account the knowledge of the environment’s behaviour.

PETRIFY [5], [8] is one of the commonly used tools for synthesis of ACs from STGs. It addresses the issues mentioned above, in particular it allows for multiway acknowledgements and can utilise latches. For synthesis, PETRIFY employs the state space of the STG (in the form of BDDs [9]), and so it suffers from the combinatorial state space explosion problem [10] — even a relatively small system specification can (and often does) yield a very large state space. This puts practical bounds on the size of control circuits that can be synthesised using such techniques, which are often restrictive, especially if the specification is not constructed manually by a designer but rather generated automatically from high-level hardware descriptions. For example, designing circuits with more than 20–30 signals with PETRIFY is often impossible.

In this paper, a different data structure for representing the state space, viz. STG unfolding prefix, is employed. The experiments in [11]–[13] show that for the application domain of ACs, they are much more compact than the explicit representation or BDDs, and thus significantly alleviate the state space explosion. An unfolding-based technique for logic decomposition of ACs is presented, which has all the nice features of PETRIFY’s algorithm, but can handle much larger STGs than PETRIFY, while delivering high-quality circuits. Together with [11]–[13], it essentially completes the synthesis flow for asynchronous circuits from STGs that does not involve building reachability graphs at any stage and yet is a fully fledged logic synthesis.

II. UNFOLDING PREFIXES

A finite and complete prefix $\text{Pref}_\Gamma$ of the unfolding $\text{Unf}_\Gamma$ of an STG $\Gamma$ is a finite acyclic net which implicitly represents

inputs: $\text{d}, \text{ldtack}$

outputs: $\text{dtack}, \text{lds}, \text{d}$

internal: $\text{csc}, \text{map}$

Fig. 1. VME bus controller (read cycle): interface (a), the timing diagram (b), the STG with an additional internal signal $csc$ resolving the encoding conflicts (c), a complex-gate implementation (d), a naive logic decomposition exhibiting hazards (e), an STG with a new signal $\text{map}$ and the corresponding logic decomposition with multway acknowledgement (f, g), and an implementation with a C-element (h).
all the reachable states of this STG together with transitions enabled at those states. Intuitively, $\text{Unf}_\Gamma$ can be obtained by successive firing of transitions starting from the initial marking of $\Gamma$, under the following assumptions: (i) for each new firing a fresh transition (called an event) is generated; (ii) for each newly produced token a fresh place (called a condition) is generated.

Due to its structural properties (such as acyclicity), the reachable states of $\Gamma$ can be represented using configurations of $\text{Unf}_\Gamma$. A configuration $C$ is a downward-closed set of events (being downward-closed means that if $e \in C$ and $f$ is a causal predecessor of $e$ then $f \in C$) without choices (i.e. for all distinct events $e, f \in C$, there is no condition $c \in \text{Unf}_\Gamma$ such that the arcs $(c, e)$ and $(c, f)$ are in $\text{Unf}_\Gamma$). Intuitively, a configuration is a partially ordered execution, i.e. an execution where the order of firing of some of its events (viz. concurrent ones) is not important. Moreover, $[e]$ will denote the local configuration of an event $e$, i.e. the smallest (w.r.t. $\subseteq$) configuration containing $e$ (it is comprised of $e$ and its causal predecessors).

$\text{Unf}_\Gamma$ is infinite whenever $\Gamma$ has an infinite run; however, if $\Gamma$ has finitely many reachable states then $\text{Unf}_\Gamma$ eventually starts to repeat itself and can be truncated (by identifying a set of cut-off events beyond which no further events are generated) without loss of information, yielding a finite and complete prefix $\text{Pref}_\Gamma$. Intuitively, an event $e$ can be declared cut-off if the already built part of the prefix contains a configuration $C^e$ (called the corresponding configuration of $e$) such that its final marking and encoding coincide with those of $[e]$ [14] and $C^e$ is smaller than $[e]$ w.r.t. some well-founded partial order on the configurations of $\text{Unf}_\Gamma$, called an adequate order [15]. Fig. 2 shows a finite and complete unfolding prefix of the STG shown in Fig. 1(c); the only cut-off event is depicted as a double box, and its corresponding configuration is $\{e_1, e_2\}$.

![Fig. 2. A finite and complete prefix of the STG in Fig. 1(c).](image)

Efficient algorithms exist for constructing unfolding prefixes [15], [16], which ensure that the number of non-cut-off events in $\text{Pref}_\Gamma$ can never exceed the number of reachable states of $\Gamma$. Moreover, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent STGs, because they represent concurrency directly rather than by multidimensional ‘diamonds’ as it is done in state graphs. For example, if $\Gamma$ consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with $2^{100}$ vertices, whereas $\text{Pref}_\Gamma$ will coincide with the net itself. Since practical STGs usually exhibit a lot of concurrency, but have rather few choice points, they are an ideal case for applying unfolding-based techniques; in fact, in many of the experiments conducted in [11]–[13] unfolding prefixes are just slightly bigger than the original STGs themselves. Thus, unfolding prefixes are well-suited for alleviating the state space explosion in STG based design.

In [12] the unfolding technique was applied to detection of encoding conflicts between reachable states of an STG. In [17], [18] a method for checking validity of transition insertions on unfolding prefixes was developed, which was successfully applied to resolution of encoding conflicts in [11]. In [13] the problem of complex-gate logic synthesis from an STG free from encoding conflicts was solved. The experiments in [11]–[13] showed that unfolding-based approach can handle much bigger STGs then PETRIY, without reducing the quality of produced circuits. This paper proposes a method for logic decomposition of SI circuits eliminating the necessity of using atomic complex-gates, which are not very realistic in practice. This completes the unfolding-based logic synthesis flow [11]–[13] for SI circuits that does not build the state graph at any stage. Combined with the STG decomposition approach of [19], it can be applied, e.g. for control re-synthesis of BALSA or TANGRAM/HASTE specifications as described in [11].

Throughout the paper, the following functions will be used. The final encoding of a configuration $C$ will be denoted by $\text{Code}(C)$; this is a Boolean vector whose elements correspond to the signals of the STG. Moreover, $\text{Code}_{x}(C)$ will denote the element of $\text{Code}(C)$ corresponding to a signal $x$, and $\text{Code}_{X}(C)$ is the projection of $\text{Code}(C)$ onto a set of signals $X$. The Boolean function $\text{Out}_{x}(C)$ is true iff $C$ enables an $x^\perp$-labelled event, and the next-state function $\text{Nxt}_{x}(C) \equiv \text{Code}_{x}(C) \oplus \text{Out}_{x}(C)$. Intuitively, the result computed by the complex-gate implementing an output or internal signal $x$ at the final state of $C$ should be $\text{Nxt}_{x}(C)$.

### III. THE SI LOGIC DECOMPOSITION ALGORITHM

In [20], a logic decomposition algorithm based on Boolean relations has been proposed. This algorithm is outlined below (with minor changes).

```plaintext
forever do
  for all non-input signals $y$ do
    $S[y] \leftarrow \emptyset$
    for all $G \in \{\text{latches}, \text{gates}\}$ do
      $S[y] \leftarrow S[y] \cup \text{decompositions}(y, G)$
      $\text{best}_H[y] \leftarrow \text{best SI candidate in } S[y]$
  if for each $y$, $\text{best}_H[y]$ is implementable
    Library matching
  stop
  if for each $y$, $\text{best}_H[y]$ is empty
    fail
    $H \leftarrow$ the most complex $\text{best}_H$
    Insert a new signal $z$ implementing $H$ into the STG
First, the algorithm computes the complex-gate implementation for each non-input signal $y$. Then it decomposes this implementation top-down, using a latch or a gate $G$ from the library, so that the output of $G$ produces the desired signal, and its inputs are produced by some complex-gates $H_i$s, which are computed using a Boolean relation solver (see Fig. 3). For example, signal $\text{esc}$ in Fig. 1(d) is implemented by the
```
complex-gate \( [csc] = dsr \land (csc \lor ld tackled) \), and when decomposed with \( G \) being the two-input AND gate, \( [H_1] = dsr \) and \( [H_2] = csc \lor ld tackled \) form a possible decomposition.

Then the algorithm checks which of the computed decompositions are SI, by trying to insert in a SI way new signals implementing the non-trivial \( H_i \)'s. In the chosen example, as \( [H_1] = dsr \) is a trivial function, there is no point in implementing it as a new signal; hence a signal map implementing \( [H_2] = csc \lor ld tackled \) is inserted as shown in Fig. 1(f). In this STG, \( H_1 \) triggers not only the signal being decomposed \( (csc) \), but also \( d \); this is the reason why \( map \) appears in the fan-in of the gates implementing both \( csc \) and \( d \) in Fig. 1(g), resulting in a multiway acknowledgement for \( map \). (As it is impossible to insert in a SI way a new signal implementing \( [H_2] = csc \lor ld tackled \) and triggering only \( csc \), the incorrect decomposition in Fig. 1(c) is not considered by the algorithm.)

Once the decompositions are computed for all non-input signals, and their SI status is evaluated, a heuristically best SI decomposition is chosen, if there is one.

If all the non-input signals are directly implementable, the algorithm performs the library matching step to recover some area and delay before stopping. At this stage, small gates can be combined into a larger one, if the latter is in the library; this is guaranteed to preserve the SI property, provided that the matched gate is atomic. On the other hand, if no SI decompositions have been found, the algorithm stops and reports a failure. Otherwise, some decomposition is heuristically chosen, a new signal implementing one of its complex-gates \( H_i \) is inserted into the STG, and the loop is repeated. On the next iteration, the implementation of \( y \) will depend on the newly inserted signal, and hence will be simpler, and some heuristics are used to prevent a significant increase in the implementations of the other signals and to ensure progress.

The top-level structure of the algorithm proposed in this paper is essentially the same; the main difference is that the insertion of a signal implementing a given Boolean function is performed using the STG unfolding prefix rather than BDDs. (The algebraic division based decomposition algorithm described in [21] can also be handled using the techniques described in this paper.) Hence one can distill the task of inserting a new signal, whose implementation is the given Boolean function \( F(X) \), into the STG, and the rest of the paper focuses on how to solve it using unfolding prefixes.

IV. Transformations

This paper primarily focuses on SB-preserving transformations, i.e. ones preserving safeness and behaviour (in the sense that the original and the transformed STGs are weakly similar, provided that the newly inserted transitions are considered silent) of the STG. Below several kinds of transition insertions that will be used for SI logic decomposition are described, and the algorithms presented in [17], [18] allow one to check their validity.

Building an unfolding prefix of an STG can be a time-consuming operation. However, the approach described in [17], [18] allows one to avoid a potentially expensive re-unfolding after each transition insertion, by introducing local modifications to the existing prefix instead. Moreover, it yields a prefix similar to the original one, which is advantageous for visualisation and allows one to transfer some information from the original prefix to the modified one.

A. Sequential pre-insertion

A sequential pre-insertion is essentially a generalised transition splitting, and is defined as follows. Given a transition \( t \) and a set of places \( S \subseteq \cdot t \), the sequential pre-insertion \( S \upharpoonright t \) is the transformation inserting a new transition \( u \) (with an additional place ‘splitting off’ the places in \( S \) from \( t \). The picture below illustrates the sequential pre-insertion \( \{p_1, p_2\} \uparrow t \).

![Diagram of sequential pre-insertion](image)

We will write \( \upharpoonright t \) instead of \( S \upharpoonright t \) if \( S = \cdot t \).

One can easily show that sequential pre-insertions always preserve safeness and traces (i.e. firing sequences with the silent transitions removed). However, in general, the behaviour is not preserved, and so a sequential pre-insertion is not guaranteed to be SB-preserving (in fact, it can introduce deadlocks) [17]. Given an unfolding prefix, it is quite easy to check whether a pre-insertion is SB-preserving [17].

If a sequential pre-insertion \( S \upharpoonright t \) is applied to an STG, the inserted transition should not ‘delay’ an input (as this would impose a constraint on the environment which was not present in the original specification), and so \( t \) must be a non-input transition. Moreover, one should take care that the output-persistency is not violated. ([17] presents an algorithm allowing one to check that the newly inserted transition will not be in a dynamic choice relation with any other transition, which ensures output-persistency.)

B. Sequential post-insertion

Similarly to sequential pre-insertion, sequential post-insertion is also a generalisation of transition splitting, and is defined as follows. Given a transition \( t \) and a set of places \( S \subseteq \cdot t \), the sequential post-insertion \( t \upharpoonright S \) is the transformation inserting a new transition \( u \) (with an additional place ‘splitting off’ the places in \( S \) from \( t \). The picture below illustrates the sequential post-insertion \( t \upharpoonright \{q_1, q_2\} \).

![Diagram of sequential post-insertion](image)

We will write \( \upharpoonright S \) instead of \( t \upharpoonright S \) if \( S = \cdot t \).

One can easily show that sequential post-insertions always preserve safeness and traces (i.e. firing sequences with the silent transitions removed). However, in general, the behaviour is not preserved, and so a sequential post-insertion is not guaranteed to be SB-preserving (in fact, it can introduce deadlocks) [17]. Given an unfolding prefix, it is quite easy to check whether a post-insertion is SB-preserving [17].

If a sequential post-insertion \( t \upharpoonright S \) is applied to an STG, the inserted transition should not ‘delay’ an input (as this would impose a constraint on the environment which was not present in the original specification), and so \( t \) must be a non-input transition. Moreover, one should take care that the output-persistency is not violated. ([17] presents an algorithm allowing one to check that the newly inserted transition will not be in a dynamic choice relation with any other transition, which ensures output-persistency.)
We will write $t_1$ instead of $t \land S$ if $S = t^*$. It can happen that a sequential post-insertion $t \land S$ yields essentially the same net as a sequential pre-insertion $S' \lor t'$, where $t \in \{ t', t'' \}$; in particular, this happens if $S \subseteq S'$ and $D = D'$. Hence, in practice it makes sense to regard them equivalent if $S = \{ t' \} = \{ t'' \}$, as they are behaviourally equivalent in such a case.

In general, concurrent insertions preserve neither safeness nor behaviour. In fact, safeness is not preserved even if $n = 0$ (e.g. when in the original net $t'$ can fire twice without $t''$ firing), and deadlocks can be introduced even if $n = 1$ (e.g. when in the original net $t''$ should fire twice before $t'$ can become enabled). In [17], an efficient test whether a concurrent insertion is SB-preserving, working on an unfolding prefix, has been developed.

If a concurrent insertion $t'' \lor t'$ is applied to the STG, the output-persistency is guaranteed to be preserved, but the inserted transition should not ‘delay’ an input, and so $t''$ must be a non-input transition.

D. Generalised insertion

Generalised transition insertion (GTI) [18] is a generalisation of concurrent insertion. It is defined as follows. Given two non-empty disjoint sets of transitions $S$ and $D$, called respectively sources and destinations, the generalised insertion $S \longrightarrow D$ is the transformation inserting a new transition $u$ with $|S|$ new places in its preset and $|D|$ new places in its postset and connecting these places to the transitions in $S$ and $D$, respectively, as shown in the picture below. In addition, some of the new places in the preset of $u$ can be initially marked.

In [18] efficiently checkable on the unfolding prefix conditions guaranteeing that the transformation is SB-preserving have been developed. Moreover, since the number of all possible GTIs usually grows exponentially with the size of the STG, their straightforward enumeration would be impractical. Hence, [18] developed a method for computing only potentially useful (in the context of logic decomposition) GTIs.

If a generalised insertion $S \longrightarrow D$ is applied to the STG, the output-persistency is guaranteed to be preserved, but the inserted transition should not ‘delay’ an input, and so $D$ must not contain any input transitions.

E. Equivalent transformations

It can happen that a sequential post-insertion $t \land S$ yields essentially the same net as a sequential pre-insertion $S' \lor t'$, where $t \in \{ t', t'' \}$; in particular, this happens if $S \subseteq S'$ and $D = D'$ (note that the notion of equivalence is structural rather than behavioural). Furthermore, one can reduce the number of behaviourally equivalent GTIs by imposing a certain minimality condition on their sources and destinations, as described in [18].

Though a GTI $S \longrightarrow D$ cannot be structurally equivalent to a sequential pre-insertion $S' \lor t$, it still makes sense to regard them equivalent if $S = \{ t' \}$ and $D = \{ t \}$, as they are behaviourally equivalent in such a case.

Similarly, though a GTI $S \longrightarrow D$ cannot be structurally equivalent to a sequential post-insertion $tS'$, it still makes sense to regard them equivalent if $S = \{ t' \}$ and $D = \{ t \}$, as they are behaviourally equivalent in such a case.

A GTI $S \longrightarrow D$ is equivalent to a concurrent insertion $t'' \lor t'$ if $S = \{ t' \}$ and $D = \{ t \}$. Hence, in practice it makes sense to impose an additional constraint $|S \cup D| > 2$ to avoid incidentally generating a GTI that is equivalent to some concurrent insertion.

F. Commutative transformations

A pair of transformations commute if the result of their application does not depend on the order they are applied. (Note that a transformation can become ill-defined after applying another transformation, e.g. $t \land \{ p, q \}$ becomes ill-defined after applying $t \land \{ p \}$.) One can observe that:
• concurrent insertions and GTIs commute with any transition insertions;
• a sequential pre-insertion and a sequential post-insertion always commute;
• two sequential pre-insertions \( S \uparrow t \) and \( S' \uparrow t' \) commute iff \( t \neq t' \) or \( S \cap S' = \emptyset \);
• two sequential post-insertions \( t \downarrow S \) and \( t' \downarrow S' \) commute iff \( t \neq t' \) or \( S \cap S' = \emptyset \).

It is important to note that an SB-preserving transition insertion remains SB-preserving if another commuting SB-preserving transition insertion is applied first. Hence transformations whose validity has been checked can be cached, and after some transformation has been applied, the non-commuting transformations are removed from the cache and the new transformations that became possible in the modified STG are computed, checked for validity and added to the cache.

A composite transition insertion is a transformation defined as the composition of several pairwise commutative transition insertions. Clearly, if a composite transition insertion consists of SB-preserving transition insertions then it is SB-preserving, i.e. one can freely combine SB-preserving transition insertions, as long as they are pairwise commutative. This property is useful for logic decomposition: typically, several transitions of a new internal signal \( \text{map} \) have to be inserted in each iteration of the algorithm, in order to preserve the consistency of the STG. For example, in Fig. 1(f) a composite transformation comprising two commuting SB-preserving insertions (adding the new transitions \( \text{map}^+ \) and \( \text{map}^- \)) has been applied in order to insert a new signal \( \text{map} \) with the given implementation \( |\text{map}| = \text{data}\&csc \) while preserving the consistency of the STG.

V. Function-guided signal insertion

As described above, logic decomposition boils down to the inserting into an STG \( \Gamma \) a new internal signal \( \text{map} \) with a given implementation \( |\text{map}| = F(X) \). That is, one has to compute a consistency-preserving and SB-preserving composite insertion \( \hat{I} \) such that, once the corresponding transitions are added to \( \Gamma \), it is possible to label each of them \( \text{map}^+ \) or \( \text{map}^- \) so that the modified STG can be synthesised as an SI circuit, and \( F(X) \) is an implementation of the newly inserted signal \( \text{map} \). In [11], a similar problem of inserting a new signal to resolve encoding conflicts has been solved by reducing it to SAT. Below we outline the main idea of that approach.

Given \( \text{Pref}_\Gamma \), one can compute a set \( \mathfrak{I} \) of valid (i.e. SB-preserving, SI-preserving, not delaying an input, etc.) insertions as described in Sect. IV. Note that the number of transformations in \( \mathfrak{I} \) is relatively small:

• the number of valid sequential pre- and post-insertions is linear in the number of STG transitions, assuming that \(|\text{map}^+| \leq c \) and \(|\text{map}^-| \leq c \) for every transition \( t \) and some constant \( c \) that is independent of the STG;
• the number of valid concurrent insertions is at most quadratic in the number of STG transitions;
• though the number of valid GTIs can be exponential in the worst case, only ‘potentially useful’ [18] for inserting a signal implementing \( F \) GTIs are computed and subsequently used by the proposed approach, and their number is usually small.

Now one can formulate a SAT problem as follows. For each insertion \( I \in \mathfrak{I} \) we create a Boolean variable, also denoted by \( I \), indicating whether \( I \) is inserted. The SAT formula below ensures that for any satisfying assignment of a SAT instance to be built, the corresponding composite insertion \( \hat{I} \) (obtained by taking the insertions whose corresponding variables are assigned 1) is valid (i.e. it preserves the consistency of the STG, the chosen individual insertions commute and introduce no auto-conflicts or self-triggering):

\[
\text{MUTEX} \land \text{SA} \land \text{CUTOFF},
\]

where the \( \text{MUTEX} \) constraint ensures that no two signal insertions \( I, I' \in \mathfrak{I} \) are non-commuting, concurrent, in auto-conflict or one of them can trigger the other; the \( \text{sign alternation constraint} \) \( \text{SA} \) ensures that a consistent assignment of signs to the newly inserted transitions is possible, and the \( \text{CUTOFF} \) constraint is needed to ensure that the properties achieved by \( \text{MUTEX} \) and \( \text{SA} \) will hold not only for the configurations of the complete prefix, but also beyond its cutoff events, i.e. for the full unfolding.

Some further constraints can be appended to this formula to ensure additional properties. For example, [11] added a constraint \( \text{CORE} \) ensuring that some of the encoding conflicts are resolved (i.e. some progress is made); in this paper we will add a constraint \( \text{FUN} \) instead, ensuring that the newly inserted signal \( \text{map} \) is implemented by a given Boolean function \( F(X) \):

\[
\text{MUTEX} \land \text{SA} \land \text{CUTOFF} \land \text{FUN}.
\]

Generation of \( \text{MUTEX}, \text{SA} \) and \( \text{CUTOFF} \) constraints is described in [11]; hence we concentrate on generating the \( \text{FUN} \) constraint, which is the main contribution of this paper; it should be noted that though the used techniques resemble those in [11], this contribution is non-trivial and technically difficult.

The \( \text{FUN} \) constraint is generated in two steps. First, we select the subset \( \mathfrak{I}_F \subseteq \mathfrak{I} \) of insertions which are compatible with \( F \). Then incremental SAT is used to compute a set of clauses expressing \( \text{FUN} \), which depend only on variables \( I \in \mathfrak{I}_F \).

We denote by \( F'_x \! \equiv \! F|_{x=0} \oplus F|_{x=1} \) the partial derivative of \( F \) w.r.t. \( x \). Intuitively, \( F'_x(X) = 1 \iff \) \( F \) essentially depends on \( x \) when its inputs are a vector \( X \), i.e. its value changes if the component corresponding to \( x \) in \( X \) is flipped. (Note that \( F'_x \) itself does not essentially depend on \( x \), and the notation \( F'_x(X) \) is used only for convenience.) We will also write \( F(C) \) instead of \( F(\text{Code}(C)) \), and similarly for \( F'_x \).

A. Selecting compatible insertions

We now introduce a notion of a compatible insertion, and then show how to check the compatibility property on \( \text{Pref}_\Gamma \).

The theory developed in [17], [18] states that if some SB-preserving insertion \( I \) is applied to \( \Gamma \), yielding the STG \( \Gamma^I \),
then for each configuration $C$ of $\text{Unf}_F$ there is a unique minimal w.r.t. configuration $\psi^f(C)$ of $\text{Unf}_F$, such that $C$ can be obtained from $\psi^f(C)$ by removing the events corresponding to the newly inserted transition $t^f$, i.e. $C = \psi^f(\psi^f(C))$, where $\psi^f$ is the function projecting configurations of $\text{Unf}_F$ to ones of $\text{Unf}_F$.

Let $x$ be a signal of $\Gamma$ and $C$ be a configuration of $\text{Unf}_F$. The predicate $\text{Trig}$ is defined as follows: $\text{Trig}(C,x, I)$ holds if there is an $x^\pm$-labelled event $e_x$ in $\text{Unf}_F$, such that $\psi^f(\psi^f(C))$ does not enable an instance of the newly inserted transition $t_1$ and $\psi^f(C) \cup \{e_x\}$ is a configuration of $\text{Unf}_F$, enabling $t_1$. Intuitively, $\text{Trig}(C,x, I)$ holds if at the state given by $\psi^f(C)$, $x^\pm$ can fire and trigger $t_1$.

An insertion $I \in \mathcal{I}$ is called compatible with a Boolean function $F(X)$ defined over the set $X$ of signals of $\Gamma$ if for each configuration $C$ of $\text{Unf}_F$ and each $x \in X$ such that $\text{Trig}(C,x, I)$ holds, $F_x^f(\psi^f(C)) = 1$. The intuition behind this definition is as follows. Suppose $t_1$ is a transition of the newly inserted signal map implementing $F$. At the final state of $\psi^f(C)$, $t_1$ can be triggered by $e_x$, i.e. firing of $x$ changes the value of $F$. The final encodings of the configurations $\psi^f(C)$ and $\psi^f(C) \cup \{e_x\}$ differ only for $x$, i.e. $F$ must essentially depend on $x$ at these states, i.e. $F_x^f(\psi^f(C)) = 1$.

One can now observe that only compatible insertions can be used to implement $F$. Indeed, incompatible ones change the value of map when $F_x^f(\psi^f(C)) = 0$, i.e. when $F$ must be stable.

Using the correspondence between the configurations of $\text{Unf}_F$ and $\text{Unf}_F$, one can re-formulate the compatibility of an $I \in \mathcal{I}$ as a simple reachability-like property of $\Gamma$, which can efficiently be checked on $\text{PREF}_F$ ([16] outlines an approach for checking reachability-like properties on unfolding prefixes). Hence, one can compute $\mathcal{I}_C$ by simply checking this property for each insertion in $\mathcal{I}$. For example, the valid insertions compatible with the function $\text{ldtack} \lor \text{csc}$ in the VME bus controller example are listed below:

$I_1: \text{ldtack} \land \neg l$
$I_2: \text{ldtack} \land \neg l \lor d^+$
$I_3: \text{ldtack} \land \neg l \lor d^-$
$I_4: \text{ldtack} \land \neg l \lor c^-$
$I_5: \text{ldtack} \land \neg l \lor d^+$
$I_6: \text{ldtack} \land \neg l \lor d^-$

B. Generating the $\text{FUN}\mathcal{I}$ clauses

The set of clauses comprising $\text{FUN}\mathcal{I}$ can now be computed as follows. For each configurations $C$ of $\text{Unf}_F$, enabling an instance $e_x$ of at least one signal $x \in X$ such that $F_x^f(C)=1$, let

$\mathcal{I}_C \equiv \{ I \in \mathcal{I} \ | \ \text{Trig}(C,x, I) \}.$

If $\text{map}$ is the newly inserted signal implementing $F$ then its transition must be enabled at the state corresponding to $\psi^f(C \cup \{e_x\})$ in $I^f$, i.e. some insertion in $\mathcal{I}_C$ must be used to implement $\text{map}$. This can be expressed by ensuring that the clause $\bigvee_{I \in \mathcal{I}_C} I$ is in $\text{FUN}\mathcal{I}$ for each such a $C$. (Note that if $\mathcal{I}_C = \emptyset$ then an empty clause is in $\text{FUN}\mathcal{I}$, which means that (1) is unsatisfiable and so one cannot insert a signal).

An inefficient way of building $\text{FUN}\mathcal{I}$ would be to enumerate for each $x \in X$ the satisfying assignments of the following Boolean formula:

$$
\bigwedge_{I \in \mathcal{I}_C} (I \iff \text{TRIG}^f(I)).
$$

Here, $\text{CONF}_C \land \text{CODE}_C \land \text{DER}_C \land \text{EN}_C \land \bigwedge_{I \in \mathcal{I}_C} (I \iff \text{TRIG}^f(I)).$

For example, the valid insertions $\text{CONF}_C \land \text{CODE}_C \land \text{DER}_C \land \text{EN}_C \land \bigwedge_{I \in \mathcal{I}_C} (I \iff \text{TRIG}^f(I))$. (Note that a clause always subsumes the latter one. (Note that a clause always subsumes itself).

Technically, this can be implemented using incremental SAT, as follows. Whenever some satisfying assignment of (2) is computed, the corresponding clause $(I_1 \lor \ldots \lor \check{I}_k)$ is obtained from it and added to $\text{FUN}\mathcal{I}$. Then, before continuing the search, the clause $(\neg I_1 \lor \ldots \lor \neg I_k)$, which excludes all the solutions resulting in the clause subsumed by the current one, is added to (2), and the process is iterated until the formula becomes unsatisfiable. In effect, the minimal elements of the projection of the set of satisfying assignments of (2) onto the set of variables $\mathcal{I}_C$ are computed.

Preliminary experiments show that this technique is quite efficient; in fact, the number of iterations is usually quite small in practice — $\text{FUN}\mathcal{I}$ often contains less than five clauses. For example, the $\text{FUN}\mathcal{I}$ constraint for $\text{ldtack} \lor \text{csc}$ in the VME bus controller example is

$$(I_1 \lor I_2 \lor I_3 \lor I_4 \lor I_5 \lor I_6)(I_7 \lor I_8 \lor I_9 \lor I_{10} \lor I_{11} \lor I_{12} \lor I_{13}).$$

Feeding (1) to a SAT solver now yields two possible composite transition insertions for the new signal implementing $\text{ldtack} \lor \text{csc}$.
**C. Correctness proof**

Below we state that the proposed method is sound (note that the method is incomplete due to the greedy nature of the search performed by the decomposition algorithm, and because only ‘structural’ insertions are used).

**Proposition 1** (Soundness). Let $\Gamma^\tilde{I}$ be the result of applying to an STG $\Gamma$ a composite insertion $\tilde{I}$ obtained from some satisfying assignment of (I). Then $[\text{map}] = F(X)$ is a possible implementation for the newly inserted signal map in $\Gamma^\tilde{I}$.

Proof: The composite insertion $\tilde{I}$ is SB-preserving and preserves the consistency and output-persistency of the original STG due to the theory developed in [11], [17], [18]. Hence we only need to prove that $[\text{map}] = F(X)$ is a possible implementation for the newly inserted signal map in $\Gamma^\tilde{I}$. It is enough to show that for any configuration $C$ of $\text{Unf}_\Gamma$, $\text{Nxt}_{\text{map}}(C) = F(C)$.

For the sake of contradiction, suppose there is a ‘bad’ configuration of $\text{Unf}_\Gamma$, for which this property does not hold. Let $C$ be a minimal w.r.t. $\subset$ bad configuration. Since the cut-off condition for STG unfoldings takes into account the encodings [14], and due to the CUTOFF constraint, if a bad configuration exists in $\text{Unf}_\Gamma$ then a configuration with the same final marking and encoding (including map) already exists in $\text{Pref}_\Gamma$, i.e. w.l.o.g., $C$ is a configuration of $\text{Pref}_\Gamma$ containing no cut-offs.

Assuming that the phase assignment (map$^+$ or map$^-$) to the newly inserted transitions is such that $\text{Nxt}_{\text{map}}(\emptyset) = F(\emptyset)$, $C \neq \emptyset$. Moreover, since $C$ is minimal w.r.t. $\subset$, each causally maximal event of $C$ is labelled by a signal in $X \cup \{\text{map}\}$, where $X$ is the support of $F$, since the events with different labels cannot trigger or disable map (as map is output-persistent due to the theory developed in [11], [17], [18]) or change the value of $F$, and so do not affect the validity of $\text{Nxt}_{\text{map}}(C) = F(C)$.

Let $e$ be a causally maximal event of $C$. We consider the possible cases, and show that each of them leads to a contradiction.

**Case 1:** Suppose $e$ is labelled by map$^\pm$. Then, due to the minimality of $C$, the configuration $C \setminus \{e\}$ is not bad, i.e. $\text{Nxt}_{\text{map}}(C \setminus \{e\}) = F(C \setminus \{e\})$. Since $C$ is bad, $\text{Nxt}_{\text{map}}(C) \neq F(C)$. Since Code$_X(C) = \text{Code}_X(C \setminus \{e\})$, $\text{Nxt}_{\text{map}}(C \setminus \{e\}) \neq \text{Nxt}_{\text{map}}(C)$, i.e. Code$_{\text{map}}(C \setminus \{e\}) \oplus \text{Out}_{\text{map}}(C \setminus \{e\}) \neq \text{Code}_{\text{map}}(C) \oplus \text{Out}_{\text{map}}(C)$. Since

$$\text{Code}_{\text{map}}(C) = \neg\text{Code}_{\text{map}}(C \setminus \{e\}) \text{ and } \text{Out}_{\text{map}}(C \setminus \{e\}) = 1, \text{Out}_{\text{map}}(C) = 1,$$ i.e. map is either auto-conditional or self-triggering, contradicting the MUTEX constraint.

**Case 2:** Suppose $e$ is labelled by some $x \in X$.

**Case 2.1:** If $F(X \setminus \{e\}) = 0$ then $F(C \setminus \{e\}) = F(C)$. Since $C \setminus \{e\}$ is not bad and $C$ is bad, $\text{Nxt}_{\text{map}}(C \setminus \{e\}) \neq \text{Nxt}_{\text{map}}(C)$. Since Code$_{\text{map}}(C \setminus \{e\}) \neq \text{Code}_{\text{map}}(C)$; $\text{Out}_{\text{map}}(C \setminus \{e\}) \neq \text{Out}_{\text{map}}(C)$. Since firing $e$ cannot disable an instance of map (as non-output-persistent transformations are rejected), $\text{Out}_{\text{map}}(C \setminus \{e\}) = 0$ and $\text{Out}_{\text{map}}(C) = 1$, i.e. $e$ triggers some instance of map, i.e. the property $\text{Trig}(\psi^f(C \setminus \{e\}, x, I))$ holds, where $I$ is the insertion corresponding to this instance of map. Hence, $I$ is not a compatible insertion, a contradiction.

**Case 2.2:** If $F(X \setminus \{e\}) = 1$ then $F(C \setminus \{e\}) \neq F(C)$. Since $C \setminus \{e\}$ is not bad and $C$ is bad, $\text{Nxt}_{\text{map}}(C \setminus \{e\}) = \text{Nxt}_{\text{map}}(C)$. Since Code$_{\text{map}}(C \setminus \{e\}) \neq \text{Code}_{\text{map}}(C)$, $\text{Out}_{\text{map}}(C \setminus \{e\}) \neq \text{Out}_{\text{map}}(C)$. If $\text{Out}_{\text{map}}(C \setminus \{e\}) = 0$ then an empty clause would be present in $\text{FUN}$, which would make the whole instance unsatisfiable, leading to a contradiction. Hence, $\text{Out}_{\text{map}}(C \setminus \{e\}) = \text{Out}_{\text{map}}(C) = 1$. Since non-output-persistent transformations are rejected, both $C \setminus \{e\}$ and $C$ enable the same transformation map, i.e. the property $\text{Trig}(\psi^f(C \setminus \{e\}, x, I))$ did not hold for the corresponding insertion $I$, and so $I \notin \mathcal{S}^f(C \setminus \{e\})$. Due to the MUTEX constraint, the instances of map cannot be concurrent or in conflict, and since $C$ enables an instance of $I$, it cannot enable instances of any other insertions in $\tilde{I}$. Hence $\mathcal{S}^f(C \setminus \{e\}) = \emptyset$ and an empty clause is present in $\text{FUN}$, making the whole instance unsatisfiable, which leads to a contradiction.

**D. Encoding conflicts**

Two distinct reachable states of an STG are in the Universal State Coding (USC) conflict if they have the same encoding, and are in the Complete State Coding (CSC) conflict if they have the same encoding and enable different sets of non-input signals. Obviously, a CSC conflict is always a USC one, but, in general, not vice versa. An STG satisfies the USC/CSC property if it is free from USC/CSC conflicts.

It is well-known that the CSC property is one of the necessary conditions required for implementability of an STG as an SI circuit [5]. Note that an STG with USC conflicts still can be synthesised, as long as it does not contain CSC conflicts. However, USC conflicts indicate redundancy in the specification (at the state graph level, the states in USC conflict can be fused without affecting the correctness), and so STGs with USC conflicts but without CSC ones are rare in practice. The result below shows that the USC property is preserved by a function-guided signal insertion.

**Proposition 2** (USC). Let $\Gamma^\tilde{I}$ be the STG obtained from an STG $\Gamma$ by applying a composite insertion $\tilde{I}$ implementing the function-guided signal insertion $[\text{map}] = F(X)$ as described in Prop. 1, and $\Gamma$ had the USC property. Then $\Gamma^\tilde{I}$ also has the USC property.

Proof: For the sake of contradiction, suppose there are two configurations $C$ and $C'$ of $\text{Unf}_\Gamma$, whose final states are
in USC conflict. If one of the configuration can be obtained from the other by adding a \( \text{map}^- \)-labelled event then their final encodings differ at the position corresponding to \( \text{map} \), a contradiction. Otherwise, the final states of \( \psi^l(C) \) and \( \psi^l(C') \) are distinct and \( \text{Code}(\psi^l(C)) = \text{Code}(\psi^l(C')) \), i.e. \( \Gamma \) had a USC conflict, a contradiction.

In [21] it was claimed that a function-guided signal insertion always preserves the CSC property. Fig. 4 demonstrates that this is not the case: in fact, a USC conflict can be ‘promoted’ to a CSC one by such an insertion. Indeed, the original STG has USC conflicts which are not CSC ones, and is implementable by the circuit \( [x] = a, [y] = b, [u] = c, [v] = c; \) however, after the function-guided insertion \( [\text{map}] = c \) shown by dashed boxes, signals \( u \) and \( v \) are no longer implementable due to the CSC conflict between the states following \( c^+ \) in the two branches.

\[
\begin{align*}
  a^+ & \rightarrow x^+ \rightarrow a^- \rightarrow x^- \\
  b^+ & \rightarrow y^+ \rightarrow b^- \rightarrow y^- \\
  \text{map}^+ & \rightarrow u^+ \\
  \text{map}^- & \rightarrow v^+ \\
  \text{inputs:} & \ a, b, c \quad \text{outputs:} \ x, y, u, v \quad \text{internal:} \ \text{map}
\end{align*}
\]

Fig. 4. An STG illustrating that a USC conflict can be promoted to a CSC one by a signal insertion.

Hence, in certain rare cases, \( \Gamma^l \) will not satisfy the CSC property and thus one will not be able to synthesise some of its output or internal signals (although the newly inserted signal \( \text{map} \) will always be synthesisable due to Prop. 1).

To cope with this problem, the following optimistic strategy can be used. The algorithm can try to perform an insertion in the hope that in most cases the CSC property will be preserved. If the resulting STG does contain a CSC conflict (this will be detected during the derivation of a complex-gate implementation in the SI logic decomposition algorithm), then the corresponding CSC core (see [11]) can be mapped (using \( \psi^l \)) to a USC core in the original STG. Then the algorithm backtracks, and solves the problem again, this time with additional constraints prohibiting composite insertions which would result in this USC core becoming a CSC core in the modified STG. The process is iterated until the modified STG has the CSC property or the SAT instance becomes unsatisfiable due to the additional constraints.

\section{Optimisations}

Below we propose a number of optimisations which can significantly reduce the computation cost of the proposed method.

First of all, one can observe that the decomposition algorithm can attempt to insert the same function \( F \) several times. Since this is the most expensive part of the algorithm, it makes sense to cache all the insertion results. For this, it is convenient to represent \( F \) as a BDD, since this ensures the canonicity and allows for easy comparisons of functions and for simply using pointers as keys for a hash table. Moreover, since the composite insertions for \( F \) and \( \mathcal{T} \) essentially coincide (only the phases of the inserted transitions are flipped), there is no need to compute the insertion for \( F \) if one for \( \mathcal{T} \) is already in the cache. (Note that complementing a BDD is a constant-time operation.)

When selecting insertions compatible with \( F \), one can quickly rule out many non-compatible ones by applying the following cheap test. \( I \) can be compatible only if each its instance \( J \) is triggered only by events labelled by signals in \( X \), and for each such \( x \)-labelled trigger \( c \) of \( J, F^{+}(C \setminus \{x\}) = 1 \), where \( C \) is the smallest (w.r.t. \( C \)) configuration enabling \( J \). Intuitively, \( I \) should be enabled only if firing an \( x \in X \) simultaneously triggers an instance of \( I \) and changes the value of \( F \) at the current state.

Since only the insertions compatible with \( F \) are considered when the clauses comprising \( \mathcal{F}UN \) are derived, one can restrict the search space only to configurations whose maximal events are labelled with signals in \( X \). This can be easily achieved by augmenting the \( CONF \) with additional clauses.

\section{Cost Function}

Once the \( \mathcal{F}UN \) constraint has been generated, the SAT problem (1) has to be solved. Typically this problem has several solutions, and a heuristic cost function is used to guide the search towards ‘good’ ones, resulting in small area and/or performance overhead. The constructed SAT instance is solved several times, with constraints on the value of the cost function appended to the formula, so that a solution minimising the value of the cost function is eventually computed. (The process resembles a binary search on the value of the cost function.) The cost function we used is similar to the one used for resolution of encoding conflicts used in [11], with the components calculating the number of remaining encoding conflicts dropped. It takes into account:

- the estimated delay introduced by the insertion;
- the total number of syntactic triggers of all output and internal signals;
- the number of inserted transitions of a signal;
- the number of signals which are not ‘locked’\footnote{Two signals are in the ‘lock’ relation [22] if their instances alternate in every execution sequence, always starting from the same signal. ‘Locking’ the newly inserted signal with as many other signals as possible is a good heuristic for logic simplification [23].} with the newly inserted signal.

The user can choose the relative weights of the components of the cost function to guide the resolution process towards solutions with the desired area/latency trade-off.

\section{Experimental Results}

The unfolding-based logic decomposition algorithm described in this paper has been implemented in the MPS tool. In this section we present the results of running it on a number of benchmarks. To solve the arising SAT instances, the \( z\text{CHAFF} \) solver [24] was used. The results are compared with those produced by \textsc{peterfy} v4.2, which uses BDDs to represent and manipulate the state graph of the STG. The gate library \textsc{peterfy}.lib that comes with \textsc{peterfy} was used as
the target library; it has combinational gates and latches with up to four inputs, and the derived libraries petrify2.lib and petrify3.lib were produced by selecting from it only the gates and latches with up to two and three inputs, respectively. All the experiments were conducted on a PC with a Pentium™ IV/3.2GHz processor and 1Gb RAM.

### A. Assorted small benchmarks

Table I presents the experimental results for a number of assorted small benchmarks from [25], with CSC conflicts resolved using the method described in [11]. The logic decomposition has been performed using the three gate libraries mentioned above, and the areas of the resulting circuits are reported. We use ‘F’ to indicate that the tool has terminated failing to decompose a circuit and ‘T’ to indicate that the tool has not terminated within 10min (this happens when the tool keeps inserting new signals without making progress). Furthermore, it turned out that occasionally PETRIFY produced incorrect circuits — apparently, there is a bug in its implementation of the logic decomposition algorithm; these cases are indicated in the table by ‘B’. The totals in the table are taken over the benchmarks for which both PETRIFY and MPSAT succeeded and produced correct solutions.

From this table one can observe that PETRIFY and MPSAT are quite comparable: they have succeeded more or less on the same benchmarks, and the areas of the resulting circuits are quite similar (the overall difference is less than 4% for each of the three gate libraries). It should be noted that the

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>petrify2.lib</th>
<th>petrify3.lib</th>
<th>petrify5.lib</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>F</td>
<td>MPSAT</td>
</tr>
<tr>
<td>ADJUST</td>
<td>184</td>
<td>184</td>
<td>184</td>
</tr>
<tr>
<td>Duplicator</td>
<td>184</td>
<td>184</td>
<td>184</td>
</tr>
<tr>
<td>Alloc-outbound</td>
<td>296</td>
<td>288</td>
<td>256</td>
</tr>
<tr>
<td>NAK-PA</td>
<td>304</td>
<td>384</td>
<td>328</td>
</tr>
<tr>
<td>NOWICK</td>
<td>240</td>
<td>256</td>
<td>280</td>
</tr>
<tr>
<td>RAM-READ-SRUF</td>
<td>320</td>
<td>368</td>
<td>312</td>
</tr>
<tr>
<td>SRUF-RAM-WRITE</td>
<td>B</td>
<td>760</td>
<td>496</td>
</tr>
<tr>
<td>SRUF-READ-CTL</td>
<td>256</td>
<td>264</td>
<td>248</td>
</tr>
<tr>
<td>TFCV-BM</td>
<td>T</td>
<td>T</td>
<td>B</td>
</tr>
<tr>
<td>MMU</td>
<td>712</td>
<td>F</td>
<td>712</td>
</tr>
<tr>
<td>MmuU</td>
<td>600</td>
<td>F</td>
<td>616</td>
</tr>
<tr>
<td>MmuU</td>
<td>496</td>
<td>552</td>
<td>B</td>
</tr>
<tr>
<td>Mod4_counter</td>
<td>T</td>
<td>536</td>
<td>528</td>
</tr>
<tr>
<td>M0R</td>
<td>504</td>
<td>552</td>
<td>480</td>
</tr>
<tr>
<td>M1R</td>
<td>B</td>
<td>504</td>
<td>464</td>
</tr>
<tr>
<td>PAR(4)</td>
<td>568</td>
<td>584</td>
<td>504</td>
</tr>
<tr>
<td>SEQ_MIX</td>
<td>344</td>
<td>320</td>
<td>328</td>
</tr>
<tr>
<td>Seq(8)</td>
<td>744</td>
<td>632</td>
<td>688</td>
</tr>
<tr>
<td>Master1882</td>
<td>552</td>
<td>672</td>
<td>528</td>
</tr>
<tr>
<td>Spec_seq(4)</td>
<td>328</td>
<td>280</td>
<td>304</td>
</tr>
<tr>
<td>TFCV-BM</td>
<td>T</td>
<td>T</td>
<td>B</td>
</tr>
<tr>
<td>TSEND-BM</td>
<td>F</td>
<td>T</td>
<td>B</td>
</tr>
<tr>
<td>Total</td>
<td>5336</td>
<td>5336</td>
<td>7256</td>
</tr>
</tbody>
</table>

Note that these two kinds of failures are pertinent to the decomposition algorithm in Sect. III due to its ‘greedy’ selection of the decomposition on each step (without the possibility of backtracking) and the fact that there is no theoretical guarantee that every circuit can be decomposed using a given finite gate library.

The VERSIFY tool [26] was used to check correctness.

### B. Scalable benchmarks

We also compared the described method with PETRIFY on the PpWkCsc(3, 4) series of scalable benchmarks modelling three weakly synchronised pipelines. They are the benchmarks from the corresponding series used in [12]. In these benchmarks all the signals are considered outputs, i.e. the control

Recall that library matching tries to combine small gates into larger ones (this does not violate the SI property) and to re-shuffle inverters at gates’ inputs and outputs so that the total number of inverters is minimised and as many gates as possible have an inverted output (as the ‘negative’ logic gates are usually smaller and faster). To ensure that re-shuffling of the inverters preserves the SI property both PETRIFY and MPSAT use the pragmatic assumption that inverters at gates’ inputs (‘bubbles’) have negligible delays.
logic is designed as a closed circuit. The inputs are inserted after the synthesis is completed, by breaking up some outputs and inserting the environment into the breaks, thus forming handshakes (sometimes with an inverter attached to the output and inserting the environment into the breaks, thus forming &function-guided signal insertion, which is the focus of this paper.

The purpose of this series is to distill as much as possible the complexity of the core sub-routine in SI logic decomposition, viz. function-guided signal insertion, which is the focus of this paper. As mentioned earlier, the performance of the decomposition algorithm and the quality of the resulting circuit is so much affected by the multitude of heuristics for choosing the decomposition on each step that it is difficult to compare the two implementations of this core sub-routine in P\textsc{etrfiy} and MPSAT. The advantage of the P\textsc{etrfiy2Lib} (with one input inverted), and itself can be implemented by a three-input C-element, which can be decomposed in two two-input C-elements. Hence, when decomposing using the p\textsc{etrfiy2Lib} gate library (which contains, among other gates and latches, an inverter and two-input C-elements with all possible input inversions), the impact of the heuristics is minimised, and P\textsc{etrfiy} and MPSAT compute very similar solutions by inserting a single signal.

The experimental results for these benchmarks are summarised in Table II. For each benchmark, this table gives the STG size (numbers of places/transition and signals), the number of reachable states, the size of the unfolding prefix (numbers of conditions/events), and the runtime (in seconds) to perform logic decomposition by P\textsc{etrfiy} and MPSAT. The unfolding prefixes were built using the p\textsc{unf} tool [27]; the time is not reported because it was negligible in all cases ($\ll$1 sec).

From this table one can see that the number of reachable states grows exponentially with the size of the STG, whereas the size of the prefix grows only quadratically. Though using BDDs helps P\textsc{etrfiy} to alleviate the state space explosion, its performance stills suffers considerably, as it struggled to decompose a circuit with 37 signals. Overall, MPSAT was clearly superior in terms of runtime and memory consumption: the cases which were intractable for P\textsc{etrfiy} were solved by MPSAT relatively easily. This confirms the observation [11]–[13] that unfolding prefixes provide an excellent data structure for representing STG state spaces, as the practical STGs usually have high concurrency but rather few choices — the ideal case for unfolding-based techniques.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P\textsc{etrfiy2Lib}(3,3)</td>
<td>34 / 20 10</td>
<td>1024</td>
<td>63 / 36</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>P\textsc{etrfiy2Lib}(3,6)</td>
<td>30 / 38 19</td>
<td>524288</td>
<td>183 / 96</td>
<td>52</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P\textsc{etrfiy2Lib}(3,9)</td>
<td>106 / 56 28</td>
<td>268435456</td>
<td>357 / 183</td>
<td>8475</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>P\textsc{etrfiy2Lib}(3,12)</td>
<td>142 / 74 37</td>
<td>137438953472</td>
<td>585 / 297</td>
<td>&gt;15hrs</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

TABLE II EXPERIMENTAL RESULTS: SCALABLE BENCHMARKS P\textsc{etrfiy2Lib}(3, \( n \)) MODELLING THREE WEAKLY SYNCHRONISED PIPELINES.

VIII. CONCLUSIONS

In this paper, we proposed an unfolding-based technique for solving the logic decomposition problem for SI circuits. It has all the attractive features of the state space based technique of [20] (highly optimised circuits, the possibility of multi-way acknowledgement, latch utilisation), and significantly alleviates the state space explosion. Together with [11]–[13], this essentially completes the unfolding-based synthesis flow for SI circuits which does not generate state graphs at any stage and yet is a fully fledged logic synthesis. Combined with the STG decomposition approach of [19], this design cycle can be applied for control re-synthesis of BALS or TANGRAM/HASTE specifications.

This technique has been implemented in the MPSAT tool. The experimental results show that P\textsc{etrfiy} and MPSAT have similar success rates and similar quality of the produced circuits, which suggests that structural insertions considered in Sect. IV are usually sufficient for logic decomposition, and complex transformations like STG restructuring are rarely useful in practice. Furthermore, unfolding-based logic decomposition scales much better.

As future work, it is planned to implement the library matching algorithm in MPSAT to recover some area and performance for the produced circuits. Furthermore, improving the heuristics for selecting the best decomposition on each step of the logic decomposition algorithm is an important direction of research, as these heuristics significantly affect the success rate of the algorithm as well as the quality of the produced circuits.

REFERENCES


