Logic Decomposition of Asynchronous Circuits Using STG Unfoldings

Victor Khomenko*
School of Computing Science, Newcastle University, UK
Victor.Khomenko@ncl.ac.uk

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.
Keywords: *Logic decomposition, unfolding, asynchronous circuits, SAT, STG.*

1. 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 [10, 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 synthesis flow for an important subclass of ACs, called *speed-independent (SI)* circuits. This model follows the classical Muller's approach [22] 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. *Signal Transition Graphs (STGs)* [5, 25] 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).

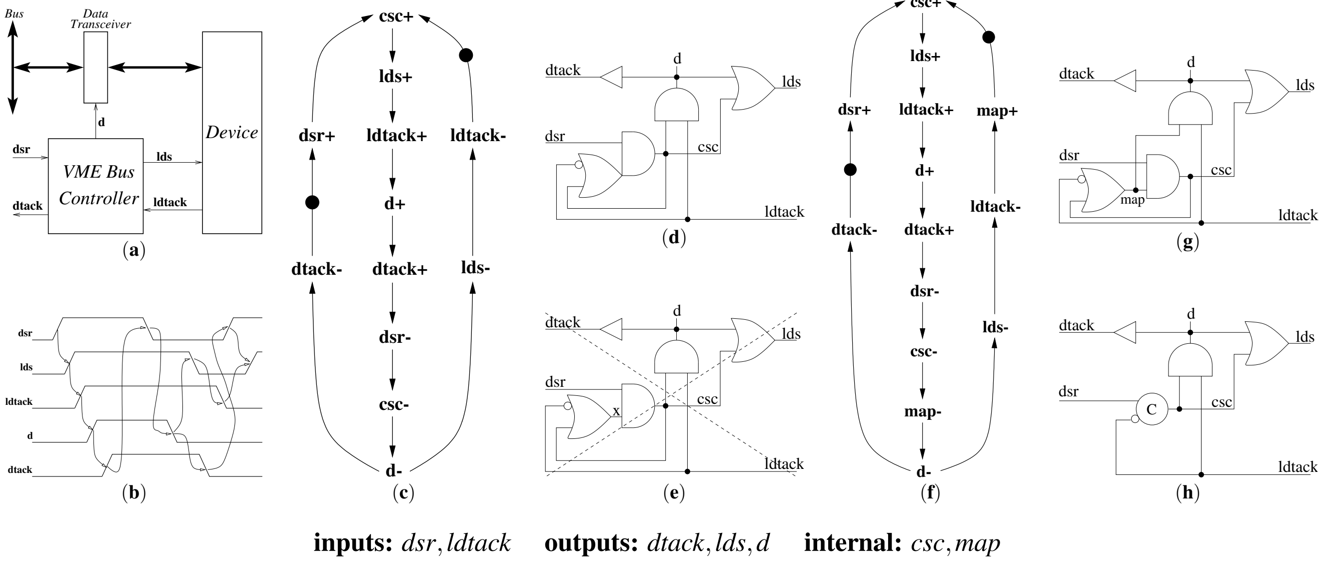
*V. Khomenko is a Royal Academy of Engineering/EPSRC Post-Doctoral Research Fellow. This research was supported by the RAEng/EPSRC research fellowship EP/C53400X/1 (DAVAC) and EPSRC grant EP/G037809/1 (VERDAD).

It should be noted that logic decomposition of SI circuits 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.) 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 [8, Chap. 2]). Assuming that the encoding conflicts have already been resolved by inserting an internal signal *csc* (this task should have been accomplished in the preceding stages of the synthesis 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



inputs: $dsr, ldtack$ **outputs:** $dtack, lds, d$ **internal:** csc, map

Figure 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 naïve logic decomposition exhibiting hazards (e), an STG with a new signal map and the corresponding logic decomposition with multiway acknowledgement (f, g), and an implementation utilising a C-element (h).

original specification (though it would be acceptable in the synchronous framework). Indeed, consider the following sequence of events:

$$⓪\ dsr^+\ csc^+\ lds^+\ ldtack^+\ d^+\ dtack^+\ dsr^-\ csc^-\ ①\ d^-\ dtack^-\ dsr^+\ ②\ csc^+\ d^+$$

At the initial state marked ⓪, $x = 1$ 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 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 naïve 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 *map* is *acknowledged* by two gates (see [20, 28] for the concept of acknowledgement). This illustrates the concept of *multiway acknowledgement*, when different transitions of a signal can be acknowledged by different gates; e.g. in this example map^+ is acknowledged by csc^+ and 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). The transformation of the circuit in Fig. 1(d) to that in Fig. 1(g) is rather not obvious at the circuit level; the corresponding STG transformation, see Fig. 1(c,f), is much easier to understand.

An implementation utilising a latch is shown in Fig. 1(h), where Muller's C-element [22] with the next-state function $[c] = ab \vee c(a \vee b)$ 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 $dsr = ldtack = 0$ and $csc = 1$ (this is the condition when the complex-gate in Fig. 1(d) and the C-element in Fig. 1(h) would behave differently); i.e. global analysis of the state space is required, which also takes into account the environment's behaviour.

PETRIFY [7, 8] is one of the commonly used tools for synthesis of ACs from STGs (see also [1, 3] for related approaches). 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 [2]), and so it suffers from the combinatorial *state space explosion* problem [27]

— 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, synthesising 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 [13, 16, 17], as well as in this paper, 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. The reason is that practical STGs have a lot of concurrency but rather few choices, in which case unfolding prefixes perform particularly well and are likely to be exponentially smaller than the state graphs. 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 [13, 16, 17], 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.

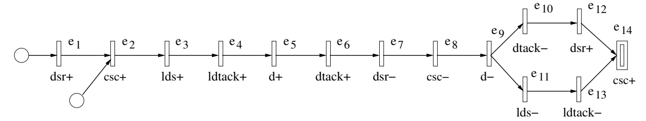
The full version of this paper can be found in the technical report [14] available on-line.

2. Unfolding prefixes

A finite and complete *prefix* $Pref_\Gamma$ of the *unfolding* Unf_Γ of an STG Γ is a finite acyclic net which implicitly represents the reachable states of Γ together with transitions enabled at those states. Intuitively, Unf_Γ can be obtained by successive firing of transitions starting from the initial marking of Γ, as follows: (i) for each new firing a fresh transition (an *event*) is generated; (ii) for each newly produced token a fresh place (a *condition*) is generated.

Due to its structural properties (such as acyclicity), the reachable states of Γ can be represented using *configurations* of Unf_Γ. 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 Unf_Γ such that the arcs (c,e) and (c,f) are in Unf_Γ). 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, $[e]$ denotes the *local* configuration of an event e, i.e. the smallest (w.r.t. $\subset$) configuration containing e (it is comprised of e and its causal predecessors).

Unf_Γ is infinite whenever Γ has an infinite run; however, if Γ has finitely many reachable states then Unf_Γ 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 $Pref_\Gamma$. 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 (i.e. signal values) coincide with those of $[e]$ [26] and C^e is smaller than $[e]$ w.r.t. some well-founded partial order on the configurations of Unf_Γ, called an *adequate order* [9]. The picture below 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 $\{e_1, e_2\}$.



Efficient algorithms exist for constructing unfolding prefixes [9, 11], which ensure that the number of non-cut-off events in $Pref_\Gamma$ can never exceed the number of reachable states of Γ. 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 Γ consists of 100 transitions which can fire once in parallel, the state graph is a 100-dimensional hypercube with 2^{100} vertices, whereas $Pref_\Gamma$ coincides 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 [13, 16, 17] 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 synthesis.

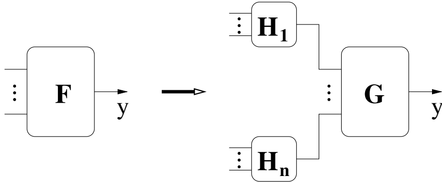
In [16] the unfolding technique was applied to detection of encoding conflicts between reachable states of an STG. In [12,15] a method for checking validity of transition insertions on unfolding prefixes was developed, which was successfully applied to resolution of encoding conflicts in [13]. In [17] the problem of complex-gate logic synthesis from an STG free from encoding conflicts was solved. The experiments in [13, 16, 17] showed that unfolding-based approach can handle much bigger STGs then PETRIFY, 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 [13, 16, 17] for SI circuits that does not build the state graph at any stage. Combined with the STG decomposition approach of [18], it can be applied, e.g. for control re-synthesis of BALSA or TANGRAM/HASTE specifications as described in [13].

3. The SI logic decomposition algorithm

In [6], a logic decomposition algorithm based on Boolean relations was proposed. This algorithm is outlined below (with minor changes).

forever do
  for all non-input signals y **do**
    $S[y] \leftarrow \emptyset$
    for all $G \in \{latches, gates\}$ **do**
      $S[y] \leftarrow S[y] \cup decompositions(y, G)$
    $best_H[y] \leftarrow$ best SI candidate in $S[y]$
  if for each y, $best_H[y]$ is implementable
    Library matching
    stop
  if for each y, $best_H[y]$ is empty
    fail
  $H \leftarrow$ the most complex $best_H$
  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_is, which are computed using a Boolean relation solver (see the picture below). For example, signal *csc* in Fig. 1(d) is implemented by the complex-gate $[csc] = dsr \wedge (csc \vee \overline{ldtack})$, and when decomposed with G being a two-input AND gate, $[H_1] = dsr$ and $[H_2] = csc \vee \overline{ldtack}$ 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_is. 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 $[H_2] = csc \vee \overline{ldtack}$ is inserted as shown in Fig. 1(f). In this STG *map* 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 *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 $[H_2] = csc \vee \overline{ldtack}$ and triggering only *csc*, the incorrect decomposition in Fig. 1(e) is not considered by the algorithm.)

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 SI decomposition is heuristically chosen, a new signal implementing one of its complex-gates H_i is inserted into the STG, and the loop continues. On the next iteration, the implementation of y will depend on the newly inserted signal, and hence will be simpler, and various heuristics are used to prevent a significant increase in the implementations of the other signals and to ensure progress.

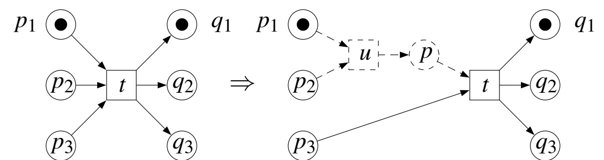
The top-level structure of the algorithm proposed in this paper is essentially the same; the main difference is that the insertion of a signal implementing a given Boolean function is performed using the STG unfolding prefix rather than BDDs. (The algebraic division based logic decomposition of [19] 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.

4. Transformations

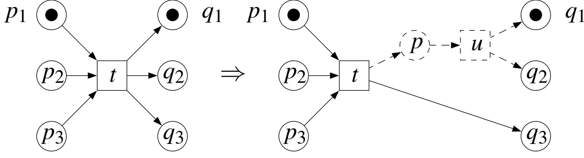
This paper primarily focuses on *SB-preserving* transformations, i.e. ones preserving safeness and behaviour (in the sense that the original and the transformed STGs are weakly bisimilar, provided that the newly inserted transitions are considered silent) of the STG. Below several kinds of transition insertions that will be used for SI logic decomposition are described, and the algorithms presented in [12,15] allow one to check their validity.

Building an unfolding prefix of an STG can be a time-consuming operation. However, the approach described in [12, 15] 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.

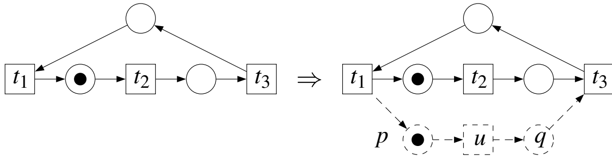
Sequential pre-insertion. A *sequential pre-insertion* is essentially a generalised transition splitting, and is defined as follows. Given a transition t and a set of places $S \subseteq {}^\bullet t$, the sequential pre-insertion $S \wr t$ is the transformation inserting a new transition u (with an additional place) 'splitting off' the places in S from t. The picture below illustrates the sequential pre-insertion $\{p_1, p_2\} \wr t$. We write $\wr t$ instead of $S \wr t$ if $S = {}^\bullet t$.



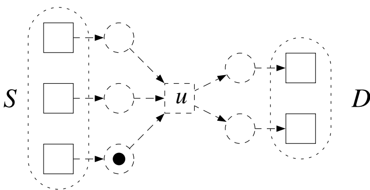
Sequential post-insertion. Similarly to sequential pre-insertion, *sequential post-insertion* is also a generalisation of transition splitting, and is defined as follows. Given a transition t and a set of places $S \subseteq t^\bullet$, the sequential post-insertion $t \wr S$ is the transformation inserting a new transition u (with an additional place) 'splitting off' the places in S from t. The picture below illustrates the sequential post-insertion $t \wr \{q_1, q_2\}$. We write $t\wr$ instead of $t \wr S$ if $S = t^\bullet$.



Concurrent insertion. *Concurrent transition insertion* can be advantageous for performance, since the inserted transition can fire in parallel with the existing ones. It is defined as follows. Given two distinct transitions, t' and t'', and an $n \in \{0, 1\}$, the concurrent insertion $t' \overset{n}{\rightarrow\!\!|\!\!\rightarrow} t''$ is the transformation inserting a new transition u (with a couple of additional places) between t' and t'', and putting n tokens in the place in its preset. We write $t' \rightarrow\!\!|\!\!\rightarrow t''$ instead of $t' \overset{0}{\rightarrow\!\!|\!\!\rightarrow} t''$ and $t' \overset{\bullet}{\rightarrow\!\!|\!\!\rightarrow} t''$ instead of $t' \overset{1}{\rightarrow\!\!|\!\!\rightarrow} t''$. The picture below illustrates the concurrent insertion $t_1 \overset{\bullet}{\rightarrow\!\!|\!\!\rightarrow} t_3$ (note that the token in p is needed to prevent a deadlock).



Generalised insertion. *Generalised transition insertion* is a generalisation of concurrent insertion. It is defined as follows [15]. Given two non-empty disjoint sets of transitions S and D, called respectively *sources* and *destinations*, the generalised insertion $S \rightarrowtail\!\!|\!\!\twoheadrightarrow D$ 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.



As the number of all possible generalised insertions usually grows exponentially with the STG size, their straightforward enumeration would be impractical. Hence, [15] developed a method for computing only potentially useful (in the context of logic decomposition) generalised insertions.

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. $t \wr \{p, q\}$ becomes ill-defined after applying $t \wr \{p\}$.)

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 *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 map^+ and map^-) has been applied to insert a new signal *map* with the given implementation $[map] = \overline{ldtack} \vee csc$ while preserving the consistency of the STG.

5. Function-guided signal insertion

As described above, logic decomposition boils down to inserting into an STG Γ a new internal signal *map* with a given implementation $[map] = F(X)$. That is, one has to compute a consistency-preserving and SB-preserving composite insertion $\widehat{I}$ such that, once the corresponding transitions are added to Γ, it is possible to label each of them map^+ or map^- so that the modified STG can be synthesised as an SI circuit, and $F(X)$ is an implementation of the newly inserted signal *map*. In [13], a related problem of inserting a new signal to resolve encoding conflicts has been solved by reducing it to Boolean satisfiability (SAT). Below we outline the main idea of that approach.

Given $Pref_\Gamma$, 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. 4. 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 $|{}^\bullet t| \leq c$ and $|t^\bullet| \leq c$ 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 generalised insertions can be exponential in the worst case, only 'potentially useful' [15] for inserting a signal implementing F generalised insertions are computed 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 $I \in \widehat{I}$. The SAT formula below ensures that for any satisfying assignment of a SAT instance to

be built, the corresponding composite insertion $\widehat{I}$ (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):

$$\mathcal{MUTEX} \wedge \mathcal{SA} \wedge \mathcal{CUTOFF},$$

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

Some further constraints can be appended to this formula to ensure additional properties. For example, [13] added a constraint $\mathcal{CORE}$ ensuring that some of the encoding conflicts are resolved (i.e. some progress is made); in this paper we add a constraint $\mathcal{FUN}$ instead, ensuring that the newly inserted signal *map* is implemented by a given Boolean function $F(X)$:

$$\mathcal{MUTEX} \wedge \mathcal{SA} \wedge \mathcal{CUTOFF} \wedge \mathcal{FUN}. \qquad (1)$$

Generation of $\mathcal{MUTEX}$, $\mathcal{SA}$ and $\mathcal{CUTOFF}$ constraints is described in [13]; hence we concentrate on generating the $\mathcal{FUN}$ constraint, which is the main contribution of this paper; it should be noted that though the used techniques resemble those in [13], this contribution is non-trivial and technically difficult.

The $\mathcal{FUN}$ constraint is generated in two steps: (i) the subset $\mathfrak{I}_F \subseteq \mathfrak{I}$ of insertions *compatible* with F is selected; (ii) incremental SAT is used to compute a set of clauses expressing $\mathcal{FUN}$ and depending only on variables $I \in \mathfrak{I}_F$.

In the rest of this sections, the following notation is used. The final encoding of a configuration C is denoted by $Code(C)$; this is a Boolean vector whose elements correspond to the signals of the STG. Moreover, $Code_x(C)$ denotes the element of $Code(C)$ corresponding to a signal x, and $Code_X(C)$ is the projection of $Code(C)$ onto a set of signals X. The Boolean function $Out_x(C)$ is true iff C enables an $x^{\pm}$-labelled event, and the *next-state* function $Nxt_x(C) \stackrel{\text{df}}{=} Code_x(C) \oplus Out_x(C)$. Intuitively, the result computed by the complex-gate implementing an output or internal signal x at the final state of C should be $Nxt_x(C)$. $F'_x \stackrel{\text{df}}{=} F|_{x=0} \oplus F|_{x=1}$ denotes 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 also write $F(C)$ instead of $F(Code_X(C))$, and similarly for F'_x.

5.1. Selecting compatible insertions

We now introduce a notion of a compatible insertion, and then show how to check the compatibility on $Pref_\Gamma$.

The theory developed in [12, 15] states that if some SB-preserving insertion I is applied to Γ, yielding the STG Γ^I, then for each configuration C of Unf_Γ there is a unique minimal w.r.t. $\subset$ configuration $\underline{\varphi}^I(C)$ of Unf_{Γ^I} such that C can be obtained from $\underline{\varphi}^I(C)$ by removing the events corresponding to the newly inserted transition t^I, i.e. $C = \psi^I(\underline{\varphi}^I(C))$, where ψ^I is the function projecting configurations of Unf_{Γ^I} to ones of Unf_Γ.

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

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

One can observe that only compatible insertions can be used to implement F, as incompatible ones change the value of *map* when $F'_x(\underline{\varphi}^I(C)) = 0$, i.e. when F must be stable.

Using the correspondence between the configurations of Unf_Γ and Unf_{Γ^I}, one can re-formulate the compatibility of an $I \in \mathfrak{I}$ as a simple reachability-like property of Γ, which can efficiently be checked on $Pref_\Gamma$ ([11] outlines an approach for checking reachability-like properties on unfolding prefixes). Hence, one can compute $\mathfrak{I}_F$ by simply checking this property for each insertion in $\mathfrak{I}$. For example, the valid insertions compatible with the function $\overline{ldtack} \vee csc$ in the VME bus controller example are listed below:

$$\begin{array}{rlrl}
 & & I_7: & csc^{-}\wr \\
 & & I_8: & csc^{-} \rightarrow\!\!\!\!+\, dtack^{-} \\
I_1: & ldtack^{-}\wr & I_9: & csc^{-} \overset{\bullet}{\rightarrow\!\!\!\!+}\, d^{+} \\
I_2: & ldtack^{-} \overset{\bullet}{\rightarrow\!\!\!\!+}\, d^{+} & I_{10}: & csc^{-} \overset{\bullet}{\rightarrow\!\!\!\!+}\, csc^{+} \\
I_3: & ldtack^{-} \overset{\bullet}{\rightarrow\!\!\!\!+}\, d^{-} & I_{11}: & csc^{-} \overset{\bullet}{\rightarrow\!\!\!\!+}\, lds^{+} \\
I_4: & ldtack^{-} \overset{\bullet}{\rightarrow\!\!\!\!+}\, csc^{-} & I_{12}: & csc^{-} \rightarrow\!\!\!\!+\, lds^{-} \\
I_5: & ldtack^{-} \overset{\bullet}{\rightarrow\!\!\!\!+}\, lds^{+} & I_{13}: & csc^{-} \overset{\bullet}{\rightarrow\!\!\!\!+}\, dtack^{+} \\
I_6: & ldtack^{-} \overset{\bullet}{\rightarrow\!\!\!\!+}\, dtack^{+} & I_{14}: & \{csc^{-}\} \rightarrowtail\!\!\!|\!\!\twoheadrightarrow \{dtack^{-}, lds^{-}\}
\end{array}$$

5.2. Generating the $\mathcal{FUN}$ clauses

The set of clauses comprising $\mathcal{FUN}$ can now be computed as follows. For each configurations C of Unf_Γ enab-

ling an instance e_x of any $x \in X$ such that $F'_x(C) = 1$, let

$$\mathfrak{I}_C \stackrel{\text{df}}{=} \{I \in \mathfrak{I}_F \mid Trig(C,x,I)\}.$$

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

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

$$\mathcal{CONF}_C \wedge \mathcal{CODE}_{C,X} \wedge \mathcal{DER}^x_X \wedge \mathcal{OUT}^x_C \wedge \bigwedge_{I \in \mathfrak{I}_F} \left(I \iff \mathcal{TRIG}^{x,I}_C \right). \quad (2)$$

Here, $\mathcal{CONF}_C$, which depends only on variables conf_e corresponding to non-cut-off events of $Pref_\Gamma$, ensures that $C \stackrel{\text{df}}{=} \{e \mid \mathsf{conf}_e = 1\}$ is a configuration (and not just an arbitrary set of events); $\mathcal{CODE}_{C,X}$ computes $Code_X(C)$ by relating the variables code_x, $x \in X$, to the variables conf_e in such a way that if the values of all conf_e are fixed and satisfy $\mathcal{CONF}_C$ then $Code_x(C) = \mathsf{code}_x$ for all $x \in X$; $\mathcal{DER}^x_X$ depends only on the variables code_x, $x \in X$, and ensures that $F'_x(C) = 1$; $\mathcal{OUT}^x_C$ ensures that $Out_x(C) = 1$; and $\mathcal{TRIG}^{x,I}_C$ depends on the variables conf_e and computes the value of $Trig(C,x,I)$ for the given x and I. Note that the last part of the formula relates the variables I corresponding to the insertions in $\mathfrak{I}_F$ to the variables conf_e in such a way that if the values of all conf_e are fixed, $I = 1$ iff $Trig(C,x,I)$ holds; that is, I occurs in the clause being generated only if the computed satisfying assignment assigns it the value of 1.

However, the number of configurations is usually very large, and it is infeasible to enumerate them using a naïve brute-force search. A more efficient approach outlined below exploits the following two observations:

- The same clause can be generated by many different configurations, and hence once one such configuration is found, the others can be excluded from the search.
- If the set of literals of one clause is a subset of the set of literals of another clause, then the latter clause is redundant and can be dropped; we say that the former clause *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 $(I_1 \vee \ldots \vee I_k)$ is obtained from it and added to $\mathcal{FUN}$. Then, before continuing the search, the clause $(\neg I_1 \vee \ldots \vee \neg I_k)$, 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 $\mathfrak{I}_F$ are computed.

Preliminary experiments show that this technique is quite efficient; in fact, the number of iterations is usually quite small in practice — $\mathcal{FUN}$ often contains less than five clauses. Moreover, technical report [14] describes some further optimisations.

For example, the $\mathcal{FUN}$ constraint for $\overline{ldtack} \vee csc$ in the VME bus controller example is

$$(I_1 \vee I_2 \vee I_3 \vee I_4 \vee I_5 \vee I_6)\ (I_7 \vee I_8 \vee I_9 \vee I_{10} \vee I_{11} \vee I_{12} \vee I_{13} \vee I_{14})\ .$$

Feeding (1) to a SAT solver now yields three possible composite transition insertions for the new signal implementing $\overline{ldtack} \vee csc$, viz. $\{I_1, I_7\}$, $\{I_1, I_{12}\}$ and $\{I_1, I_{14}\}$. However, $\{I_1, I_{12}\}$ yields

$$[csc] = dsr \wedge (map \wedge \overline{ldtack} \vee csc) \text{ or}$$
$$[csc] = dsr \wedge (map \wedge \overline{lds} \vee csc)$$

as the possible implementations of *csc* (i.e. the signal being decomposed), instead of the expected $[csc] = dsr \wedge map$, and so is heuristically rejected. (One can re-formulate the problem of checking whether a given composite insertion yields the expected implementation for the signal being decomposed as a separate reachability-like property of Γ, which can be efficiently checked on $Pref_\Gamma$ [11].) The insertion $\{I_1, I_{14}\}$, though it yields the expected implementation for *csc*, is heuristically rejected because it triggers more signals (and thus upsets their implementations) than $\{I_1, I_7\}$ (in fact, $\{I_1, I_{14}\}$ yields a very bad circuit with four-input complex-gates for *lds* and *dtack*), and so $\{I_1, I_7\}$ is chosen by the decomposition algorithm, cf. Fig. 1(f).

5.3. Cost function

Once the $\mathcal{FUN}$ 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.) See technical report [14] for more detail.

5.4. Correctness and encoding conflicts

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). The proof can be found in technical report [14].

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

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 [8]. 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 states that the USC property is preserved by a function-guided signal insertion. The proof can be found in [14].

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

In [19] it was claimed that a function-guided signal insertion always preserves the CSC property. This is actually not the case, as the counterexample in [14] shows: in fact, a USC conflict can be 'promoted' to a CSC one by such an insertion. Hence, in certain rare cases, $\Gamma^{\widehat{I}}$ 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 *map* will always be synthesisable due to Prop. 1). A possible strategy to cope with this problem is outlined in [14].

6. Experimental results

The unfolding-based logic decomposition algorithm described in this paper has been implemented in the MPSAT tool. In this section we present the results of running it on a number of benchmarks. To solve the arising SAT instances, the ZCHAFF solver [21] was used. The results are compared with those produced by PETRIFY v4.2, which uses BDDs to represent and manipulate the state graph of the STG. The gate library `petrify.lib` that comes with PETRIFY 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*^{TM} IV/3.2GHz processor and 1Gb RAM.

6.1. Assorted small benchmarks

Table 1 presents the experimental results for a number of assorted small benchmarks from [4], with CSC conflicts resolved using the method described in [13]. 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).[1] Furthermore, it turned out that occasionally PETRIFY produced incorrect circuits[2] — 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.

Benchmark	`petrify2.lib` PFY	`petrify2.lib` MPSAT	`petrify3.lib` PFY	`petrify3.lib` MPSAT	`petrify.lib` PFY	`petrify.lib` MPSAT
ADFAST	F	F	F	656	416	F
DUPLICATOR	184	184	184	184	184	184
ALLOC-OUTBOUND	296	288	256	248	256	248
NAK-PA	304	384	328	328	328	344
NOWICK	240	256	280	264	280	264
RAM-READ-SBUF	320	368	312	360	304	312
SBUF-RAM-WRITE	B	760	496	536	496	608
SBUF-READ-CTL	256	264	248	248	264	264
IRCV-BM	T	T	B	T	B	632
MMU	712	F	712	712	712	696
MMU0	600	F	616	672	624	664
MMU1	496	552	B	544	B	576
MOD4_COUNTER	T	536	528	520	528	472
MR0	504	552	480	520	488	504
MR1	B	504	464	560	440	528
PAR(4)	568	584	504	520	504	560
SEQ_MIX	344	320	328	304	328	304
SEQ(8)	744	632	688	632	688	632
MASTER_1882	552	672	528	640	528	640
SPEC_SEQ(4)	328	280	304	280	304	280
TRCV-BM	T	T	B	F	B	F
TSEND-BM	F	T	B	F	B	720
Total	5136	5336 +3.89%	7256	7528 +3.75%	7256	7504 +3.42%

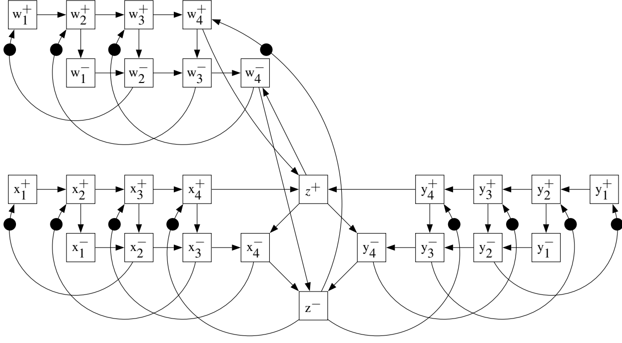
Table 1. Assorted small STGs.

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 'Library matching'[3] step of the logic decomposition algorithm in Sect. 3 has not been implemented in MPSAT yet, so non-optimised numbers are given in the MPSAT columns; it seems reasonable to expect that this optimisa-

[1] Note that these two kinds of failures are pertinent to the decomposition algorithm in Sect. 3 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.

[2] The VERSIFY tool [24] was used to check correctness.

[3] 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 ('bubbles') 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 such re-shuffling preserves the SI property both PETRIFY and MPSAT use the pragmatic assumption that inverters at gates' inputs have negligible delays.



outputs: $w_1, w_2, w_3, w_4; x_1, x_2, x_3, x_4; y_1, y_2, y_3, y_4; z$

Figure 2. The PpWkCsc$(3,4)$ STG modelling three weakly synchronised pipelines.

Benchmark	STG $\|P\|/\|T\|$	STG *Sig*	Reachable states	Prefix $\|B\|/\|E\|$	Time, [s] Pfy	Time, [s] MPSat
PpWkCsc(3,3)	34 / 20	10	1024	63 / 36	2	1
PpWkCsc(3,6)	70 / 38	19	524288	183 / 96	52	2
PpWkCsc(3,9)	106 / 56	28	268435456	357 / 183	8475	5
PpWkCsc(3,12)	142 / 74	37	137438953472	585 / 297	>15hrs	11

Table 2. Scalable pipelines.

tion could recover 10–15% of the area. It is also worth noting that MPSat's failures for Trcv-bm benchmark are due to the effect described in Sect. 5.4, namely due to promoting a USC conflict to a CSC one (the current implementation does not try to recover in such a case and simply fails).

One should treat the provided results with caution, as the parameters like the success rate and the quality of the resulting circuits reflect the quality of the heuristics used for selecting a decomposition on each step of the logic decomposition algorithm, rather than the quality of the function-guided signal insertion sub-routine, which is the main point of this paper. However, the following important conclusion seems justified. When performing a signal insertion at the level of the state graph, Petrify can completely restructure the STG, whereas unfolding-based insertion performed by MPSat is currently limited to structural insertions considered in Sect. 4. Nevertheless, these experiments seem to indicate that for logic decomposition such structural insertions are not too restrictive in practice compared to the signal insertion at the state graph level, i.e. STG restructuring is useful only in rare cases. On the other hand, unfolding-based insertions scale better (see below), which is in practice a much more desirable quality than the ability to do restructuring.

6.2. Scalable benchmarks

We also compared the described method with Petrify on the PpWkCsc$(3,n)$ series of scalable benchmarks modelling three weakly synchronised pipelines. They are the benchmarks from the corresponding series used in [16]. Fig. 2 shows the PpWkCsc(3,4) STG.

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 and success rate of the decomposition algorithm and the quality of the resulting circuit are 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 Petrify and MPSat. The advantage of the PpWkCsc$(3,n)$ series is that the impact of the decomposition selection heuristics is very restricted: each signal except z in these benchmarks can be implemented either by an inverter or by a two-input C-element (with one input inverted), and z 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 `petrify2.lib` 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 Petrify and MPSat compute very similar solutions by inserting a single signal.

The experimental results for these benchmarks are summarised in Table 2. For each benchmark, this table gives the STG size (numbers of places/transitions 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 Petrify and MPSat. The unfolding prefixes were built using the Punf tool [23]; the corresponding runtime is not reported because it was negligible in all cases ($\ll$1sec).

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 Petrify 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. This confirms the observation [13, 16, 17] 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.

7. 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 [6] (highly optimised circuits, the possibility of multiway acknowledgement, latch utilisation), and significantly alleviates the state space explosion. Together with [13, 16, 17], 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 syn-

thesis. Combined with the STG decomposition approach of [18], this synthesis flow can be applied for control resynthesis of BALSA or TANGRAM/HASTE specifications.

This technique has been implemented in the MPSAT tool. The experimental results show that PETRIFY and MPSAT have similar success rates and similar quality of the produced circuits, which suggests that structural insertions considered in Sect. 4 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 in 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

[1] P. Beerel and T.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. ICCAD'92*, pages 581–586, 1992.

[2] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, C-35-8:677–691, 1986.

[3] S. Burns. General conditions for the decomposition of state holding elements. In *Proc. ASYNC'96*, pages 48–57, 1996.

[4] J. Carmona and J. Cortadella. Encoding large asynchronous controllers with ILP techniques. *IEEE Trans. on CAD*, 27(1):20–33, 2008.

[5] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, Lab. for Comp. Sci., MIT, 1987.

[6] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev. Decomposition and technology mapping of speed-independent circuits using Boolean relations. *IEEE Trans. on CAD*, 18(9):1221–1236, 1999.

[7] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. PETRIFY: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Trans. on Inf. and Syst.*, E80-D(3):315–325, 1997.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer, 2002.

[9] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *FMSD*, 20(3):285–310, 2002.

[10] International technology roadmap for semiconductors: Design, 2009. URL: `http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Design.pdf`.

[11] V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, School of Comp. Sci., Newcastle Univ., 2003.

[12] V. Khomenko. Behaviour-preserving transition insertions in unfolding prefixes. In *Proc. ATPN'07*, volume 4546 of *LNCS*, pages 204–222. Springer, 2007.

[13] V. Khomenko. Efficient automatic resolution of encoding conflicts using STG unfoldings. *IEEE Trans. on VLSI Syst.*, 17(7):855–868, 2009. Special Section on Asynchronous Circuits and Systems.

[14] V. Khomenko. Logic decomposition of asynchronous circuits using STG unfoldings. Technical Report CS-TR-1215, School of Comp. Sci., Newcastle Univ., 2010. URL: `http://www.cs.ncl.ac.uk/publications/trs/abstract/1215`.

[15] V. Khomenko. A new type of behaviour-preserving transition insertions in unfolding prefixes. In *Proc. ICGT'10*, pages 75–90. Springer, 2010.

[16] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state coding conflicts in STG unfoldings using SAT. *Fund. Inf.*, 62(2):1–21, 2004.

[17] V. Khomenko, M. Koutny, and A. Yakovlev. Logic synthesis for asynchronous circuits based on Petri net unfoldings and incremental SAT. *Fund. Inf.*, 70(1–2):49–73, 2006.

[18] V. Khomenko and M. Schaefer. Combining decomposition and unfolding for STG synthesis. In *Proc. ATPN'07*, volume 4546 of *LNCS*, pages 223–243. Springer, 2007.

[19] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Logic decomposition of speed-independent circuits. *Proceedings of the IEEE*, 87(2):347–362, 1999.

[20] A. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proc. 6th MIT Conf. on Adv. Research in VLSI*, pages 263–278. MIT Press, 1990.

[21] S. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. CHAFF: Engineering an efficient SAT solver. In *Proc. DAC'01*, pages 530–535. ASME Tech. Publ., 2001.

[22] D. Muller and W. Bartky. A theory of asynchronous circuits. In *Proc. Int. Symp. of the Theory of Switching*, pages 204–243, 1959.

[23] PUNF home page. URL: `http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf`.

[24] O. Roig. *Formal Verification and Testing of Asynchronous Circuits*. PhD thesis, Univ. Politecn. de Catalunya, 1997.

[25] L. Rosenblum and A. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proc. Int. Workshop on Timed Petri Nets*, pages 199–206. IEEE Comp. Soc. Press, 1985.

[26] A. Semenov. *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding*. PhD thesis, School of Comp. Sci., Newcastle Univ., 1997.

[27] A. Valmari. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, chapter The State Explosion Problem, pages 429–528. Springer, 1998.

[28] V. Varshavsky, editor. *Self-Timed Control of Concurrent Processes*. Kluwer Acad. Publ., 1990. Transl. from Russian, publ. by Nauka, Moscow, 1986.