



Efficient Composition of Scenario-based Hardware Specifications

Alessandro de Gennaro , Paulius Stankaitis, Andrey Mokhov

School of Engineering, Newcastle University, UK

Corresponding author: a.de-gennaro@ncl.ac.uk

Abstract—Complex hardware systems can be designed by breaking down their behaviour into high-level descriptions of constituent *scenarios* and then composing these scenarios into an efficient hardware implementation using a form of high-level synthesis. There are a few existing methodologies for such scenario-based specification and synthesis, and in this paper we focus on highly concurrent systems, whose scenarios are typically described using explicit concurrency models such as partial orders.

We propose a new algorithm for composition of partial order scenarios. Unlike previously published methods, the proposed algorithm supports *composition constraints*, which allow the designer to restrict certain aspects of the composition in order to reuse legacy IP. Furthermore, our implementation is more scalable and can cope with specifications comprising hundreds of scenarios at the cost of only $\simeq 5\%$ of area overhead compared to optimal solutions obtained by exhaustive search.

The proposed algorithm is implemented in an open-source EDA tool, validated on a set of benchmarks, and compared to the state-of-the-art behavioural composition approaches and to other existing methodologies that make use of behavioural synthesis.

I. INTRODUCTION

Hardware systems grow more complex every year: processors gain new features and application-specific instructions [1], and the number of processing cores and other IP components steadily increases following the need for IP reuse [2]. Conventional approaches to the development of hardware systems rely on HDL system descriptions, which require designers to deal with low-level implementation details. When the level of complexity increases, it is convenient to raise the level of abstraction for easier system representation, for using automatic hardware synthesis features, and, consequently, for an increased productivity [3]. See examples [4], [5].

In this work, we use the high-level methodology based on the Conditional Partial Order Graphs (CPOG) [6] formalism to design hardware architectures in the control domain. This methodology, originally conceived for the design of processor instruction set architectures (ISAs) [7], is supported by automated hardware synthesis flow, and by algorithms for the derivation of efficient hardware implementations [8]. However, previously published algorithms do not scale to large numbers of behavioural scenarios and have no support for composition constraints, which are important for real-life systems heavily relying on IP reuse. This motivates our research.

The paper comprises the following sections.

- **Background:** Section II reviews the CPOG formalism and the related methodology [7].

- **Related work:** Section III compares the methodology with other approaches in the field of behavioural synthesis, and reviews existing algorithms for efficient composition of scenarios.
- **Scenario composition algorithm:** Section IV presents our main contribution: a new algorithm for CPOG composition that scales to systems comprising hundreds of partial order scenarios and supports composition constraints.
- **Design automation:** Section V describes the developed open-source tool SCENCO [9], which is integrated in the WORKCRAFT framework [10] as an external plugin and implements the CPOG methodology.
- **Algorithm and tool validation:** Section VI validates the presented contributions on a set of benchmarks that includes ad-hoc controllers, processor instruction sets, and software output logs.

We discuss achieved results and future research in Section VII.

This paper is an extended version of [11] and includes the following changes. We review the CPOG-based design methodology in Section II-B and summarise differences with other existing behavioural synthesis approaches in Section III-A. The new algorithm for scenario composition is described in greater detail: in particular, we discuss how to reduce the space of possible solutions to improve the performance of the algorithm (Section IV-A), describe how the composition algorithm handles constraints (Sections IV-B and IV-D), and analyse the algorithm's correctness and complexity (Sections IV-E and IV-F). We describe how to synthesise the interface between the controller and the controlled datapath modules in Section V-A. Finally, the presented algorithm and tool are evaluated on an extended set of benchmarks in Section VI.

II. BACKGROUND

Complex systems are designed by breaking them down into their constituent behaviours, or scenarios. In this paper, a *scenario* is a list of operations that are executed in a specified order. Formally, a scenario $s = (\mathcal{O}, \prec)$ is a *partial order* (PO) [12], i.e. a binary precedence relation \prec describing dependencies between a set of operations \mathcal{O} that satisfies two properties:

- Irreflexivity: $\forall a \in \mathcal{O}, \neg(a \prec a)$
- Transitivity: $\forall a, b, c \in \mathcal{O}, (a \prec b) \wedge (b \prec c) \Rightarrow (a \prec c)$

A *scenario specification* formally captures the behaviour of a system by the set of its constituent scenarios $S = \{s_1, \dots, s_n\}$. As an example, the behaviour of a processor can be specified by the set of instructions it can execute, see Figure 1a.

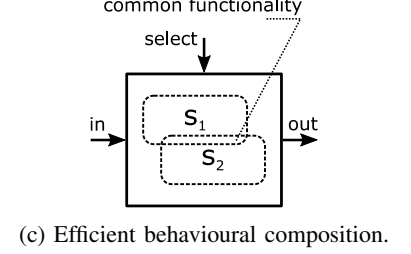
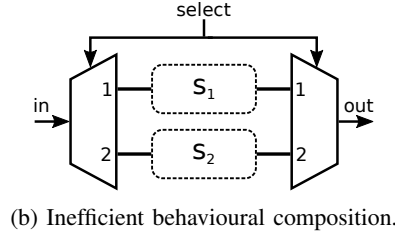
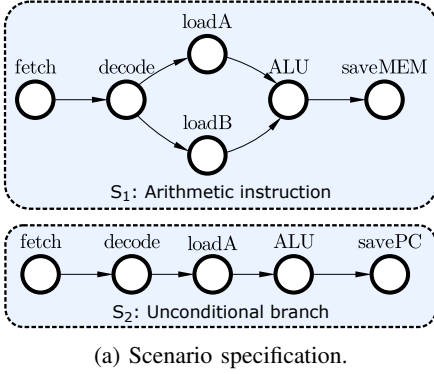


Fig. 1: Subfigure (a) shows a scenario specification comprising two processor instructions whose behaviour is expressed using partial orders. Scenario s_1 corresponds to the *arithmetic instruction* that fetches an instruction from the program memory, decodes it, loads the two operands concurrently ($\text{loadA} \parallel \text{loadB}$), uses them to perform an arithmetic operation (ALU), and subsequently saves the result into the memory via the saveMEM operation. Scenario s_2 is the *unconditional branch*, which takes one operand to compute the jump address (ALU) and saves the result into the program counter register (savePC). Subfigures (b-c) show two approaches to behavioural composition of scenarios.

We use *Conditional Partial Order Graphs* (CPOGs) (reviewed in Section II-A) for representation of scenario specifications. CPOGs are supported by efficient scenario composition methodology that allows to take advantage of the similarities between scenarios. The methodology will be described in detail in Section II-B, but here we provide an intuitive explanation of what we mean by ‘efficient composition’. Figure 1b shows an inefficient composition where each scenario is synthesised in isolation and the right scenario is selected by means of (de)multiplexers in runtime. A more efficient approach consists of deriving a hardware implementation where system resources and common parts of behaviour are shared, as shown in Figure 1c.

This paper presents a new approach to composition of scenarios for deriving efficient hardware implementations of control circuits, such as interface controllers and processor instruction decoders.

A. Conditional Partial Order Graphs

A CPOG is a collection of scenarios in the form of partial orders. Formally, a CPOG [6] is a tuple $H = (V, E, B, \phi)^1$:

- V is a set of vertices which correspond to operations (or events) in a modelled system.
- $E \subseteq V \times V$ is a set of arcs representing dependencies between the operations.
- B is a set of Boolean variables $\{b_1, b_2, \dots, b_{|B|}\}$. A *code* is an assignment $c : B \rightarrow \{0, 1\}$ of these variables, e.g. $B = \{b_1, b_2\}$, $c(b_1) = 0$ and $c(b_2) = 1$ that will be further denoted as $c = 01$ for brevity. A code selects a particular PO from those contained in the CPOG.
- Function $\phi : (V \cup E) \rightarrow F(B)$, with $F(B)$ being the set of all Boolean functions over variables in B , assigns a Boolean condition $\phi(z) \in F(B)$ to every vertex and arc $z \in V \cup E$.

¹A CPOG is $H = (V, E, B, \phi, \rho)$ in [6], we do not use ρ in this paper.

CPOGs can be represented graphically: vertices are depicted as circles \bigcirc , and arcs are depicted as arrows \rightarrow . Vertices and arcs are labelled by their conditions $\phi(z)$. For example, Figure 2 shows a CPOG with two possible *projections* on top (we define projections in Section II-B3). The purpose of conditions ϕ is to switch vertices and arcs on (off) when the conditions on them are (not) satisfied. We use dashed circles and arrows to represent vertices and arcs that have been switched off by their conditions.

The example in Figure 2 shows that a CPOG can be used to represent multiple behavioural scenarios compactly by overlaying their common parts. In practice CPOGs remain compact and easy to understand even when the number of scenarios increases, making the formalism suitable for representing a large class of hardware systems.

B. Design methodology

This section reviews the design methodology based on the CPOGs [7], see Figure 3. The scenarios of a system are formally specified as a *scenario specification* (in the form of a set of partial orders). Scenarios are composed into a *system specification* (in the form of a CPOG), which represents the complete system behaviour. The latter is used to synthesise a *hardware controller* (in the form of gate-level description in Verilog). The presented approach enables the specification of *composition constraints* (in the form of codes). The controller is then automatically interfaced to the specified *datapath modules* in the final *system implementation*.

As a running example, the methodology is applied to the system described by the scenarios in Figure 1a.

1) *Scenario specification*: A hardware system is described by a collection of scenarios, each in the form of a partial order. Vertices and arcs constitute the basic elements of these graphs, where vertices represent system operations (or events), and arcs represent dependencies between them.

System scenarios can be specified either graphically (see Section V) or textually in a file. Text files containing scenarios

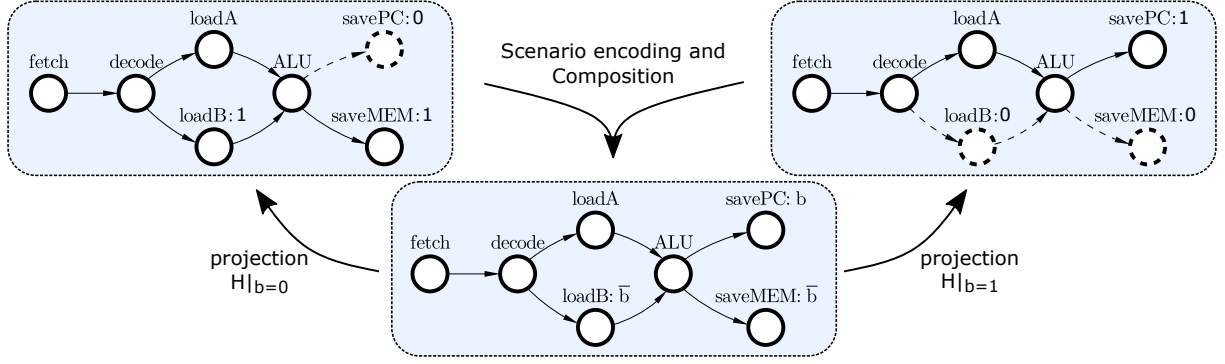


Fig. 2: Example of CPOG with 2 projections: $H|_{b=0}$ on the left side, $H|_{b=1}$ on the right.

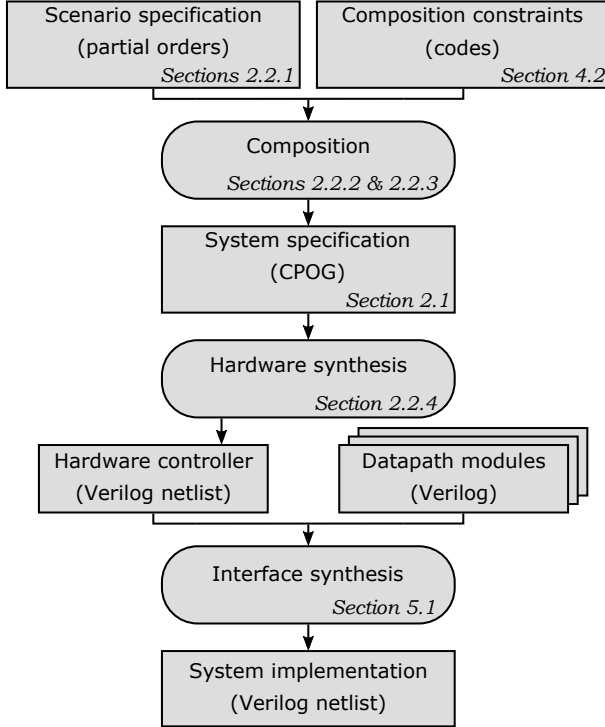


Fig. 3: Specification and hardware synthesis flow.

are parsed, and each scenario is converted into a graph. As an example, text-based descriptions of the scenarios in Fig. 1a are shown below.

$s_1 = \text{fetch} \rightarrow \text{decode} \rightarrow (\text{loadA} + \text{loadB}) \rightarrow \text{ALU} \rightarrow \text{saveMEM}$
 $s_2 = \text{fetch} \rightarrow \text{decode} \rightarrow \text{loadA} \rightarrow \text{ALU} \rightarrow \text{savePC}$

The effort required by engineers to produce such scenario specifications is high, and it is desirable to extract scenarios from higher-level descriptions. There are several examples of high-level specification languages targeting processor architectures, e.g. see Arm's Architecture Specification Language [13] and Sail [14]. This aspect of automation is outside the scope of this paper; we refer the reader to [15] for a relevant example.

2) *Scenario encoding*: Scenario encoding is the process of finding an injective function between a set of scenarios and a set of codes. Let n be the number of scenarios. The following definitions will be used to formally state the *CPOG encoding problem*.

- S is the set of scenarios $\{s_1, s_2, \dots, s_n\}$ described as POs.
- C is the universe of codes $\{c_1, c_2, \dots, c_{|C|}\}$ satisfying the following two properties:

- 1) $|C| = 2^{|B|}$;
- 2) $c_i \neq c_j$ for $1 \leq i < j \leq |C|$;

e.g. given a set of Boolean variables $B = \{b_1, b_2\}$, the corresponding code universe is $C(B) = \{00, 01, 10, 11\}$.

- *Encoding* is a set of n pairs $\{(s_1, c_1), \dots, (s_n, c_n)\}$, where each scenario s_i is encoded by the code c_i , such that:

$$- s_i \neq s_j \wedge c_i \neq c_j \text{ for all } 1 \leq i < j \leq n;$$

The arithmetic instruction scenario s_1 and the unconditional branch scenario s_2 in Fig. 1a can be encoded by one Boolean variable $B = \{b\}$, with the code universe $C(B) = \{0, 1\}$. The encoding illustrated in Fig. 2 is $e = \{(s_1, 0), (s_2, 1)\}$.

Different encodings lead to different CPOGs, and consequently to different hardware implementations, see next sections.

3) *Composition*: Let $e = \{(s_1, c_1), \dots, (s_n, c_n)\}$ be a scenario encoding for a CPOG $H = (V, E, B, \phi)$. The following definitions will be used to formally state the *CPOG synthesis problem*.

- A *projection* $H|_{c_i}$ applies the code c_i to all Boolean conditions of H . The result is a graph H_i , whose vertex/arc conditions are now fully evaluated to 1 or 0, see Figure 2.
- The operation $\text{scen}(H_i)$ removes vertices and arcs with 0 condition, and applies the transitive closure to the resulting graph, obtaining the scenario s_i .

The purpose of the above definitions is to let a code c_i select a scenario s_i from the CPOG:

$$\forall 1 \leq i \leq n, \text{scen}(H|_{c_i}) = s_i$$

The CPOG synthesis process uses the encoding e to synthesise the CPOG H . It produces the encoding functions $F(B) = \{f_1, f_2, \dots, f_n\}$, so that the code $c_i \in e$ selects the scenario $s_i \in e$. Following [6], we represent the CPOG H as the following linear combination of projections:

$$H = f_1 H|_{c_1} + \dots + f_n H|_{c_n} = \sum_{1 \leq i \leq n} f_i H_i = \sum_{1 \leq i \leq n} f_i \text{scen}^{-1}(s_i)$$

The CPOG synthesis requirement is satisfied if the encoding functions are orthogonal ($f_i f_j = 0, 1 \leq i < j \leq n$), and are not contradictions, i.e $f_i \neq 0$ for all $1 \leq i \leq n$.

As an example, consider the encoding $\{(s_1, 0), (s_2, 1)\}$ of the scenarios in Figure 1a. The resulting CPOG should be in the form of $H = f_1 H|_{c_1} + f_2 H|_{c_2}$ such that $\text{scen}(H|_{c_1}) = s_1$ and $\text{scen}(H|_{c_2}) = s_2$. The CPOG is represented by the linear combination $H = \bar{b}H|_0 + bH|_1$, and the encoding functions $f_1 = \bar{b}$ and $f_2 = b$ satisfy the synthesis requirement. Figure 2 shows the resulting CPOG H at the bottom, and the projections $H|_{b=0}$ and $H|_{b=1}$ on the top. The CPOG represents the system specification.

4) *Hardware synthesis*: The hardware synthesis step of the design flow extracts a set of Boolean equations from the derived CPOG, obtaining an implementation of the *controller*. Its area, latency and power strongly correlate with the CPOG complexity [16], defined as the number of Boolean literals of conditions ϕ . An operation $v \in V$ can be executed if:

- 1) it belongs to the current projection, i.e. $\phi(v) = 1$;
- 2) all preceding vertices have already been executed:
 $\forall u \in V, (u \prec v) \Rightarrow \text{ack}(u)$.

This is captured in terms of Boolean equations as follows:

$$\text{req}(v) = \phi(v) \wedge \prod_{u \in V} [\phi(u) \wedge \phi((u, v)) \Rightarrow \text{ack}(u)],$$

where (u, v) is the arc from u to v , $\text{req}(v)$ is the request signal which activates the v operation, while $\text{ack}(u)$ is the acknowledgement signal which comes from the u operation, and indicates its completion. As an example, the hardware implementation (in the form of Boolean equations) of the CPOG in Figure 2 is shown below:

$$\begin{cases} \text{req}(\text{fetch}) = \text{go} \\ \text{req}(\text{decode}) = \text{ack}(\text{fetch}) \\ \text{req}(\text{loadA}) = \text{ack}(\text{decode}) \\ \text{req}(\text{loadB}) = \bar{b} \wedge \text{ack}(\text{decode}) \\ \text{req}(\text{ALU}) = \text{ack}(\text{loadA}) \wedge (\bar{b} \Rightarrow \text{ack}(\text{loadB})) \\ \text{req}(\text{savePC}) = b \wedge \text{ack}(\text{ALU}) \\ \text{req}(\text{saveMEM}) = \bar{b} \wedge \text{ack}(\text{ALU}) \\ \text{done} = (b \Rightarrow \text{ack}(\text{savePC})) \wedge (\bar{b} \Rightarrow \text{ack}(\text{saveMEM})) \end{cases}$$

Signals *go* and *done* are automatically added into set of operations to delimit the start and the end of a scenario execution. The above Boolean equations are used for the synthesis of the gate-level description of the system hardware controller (in Verilog), which is in compliance with its scenario specification. Finally, the controller can be connected to the specified synchronous or asynchronous datapath modules automatically, see Section V-A, and the final system implementation can be further processed by conventional EDA tools.

III. RELATED WORK

A. Behavioural synthesis

Behavioural synthesis is not new and several other approaches exist that allow the designer to formally describe the behaviour of a controller and synthesise the corresponding hardware implementation. The most relevant approaches are: the work by Cortadella *et al.* [17] that is based on Signal Transition Graphs (STG) as the formal specification model and produces asynchronous controllers; and the work

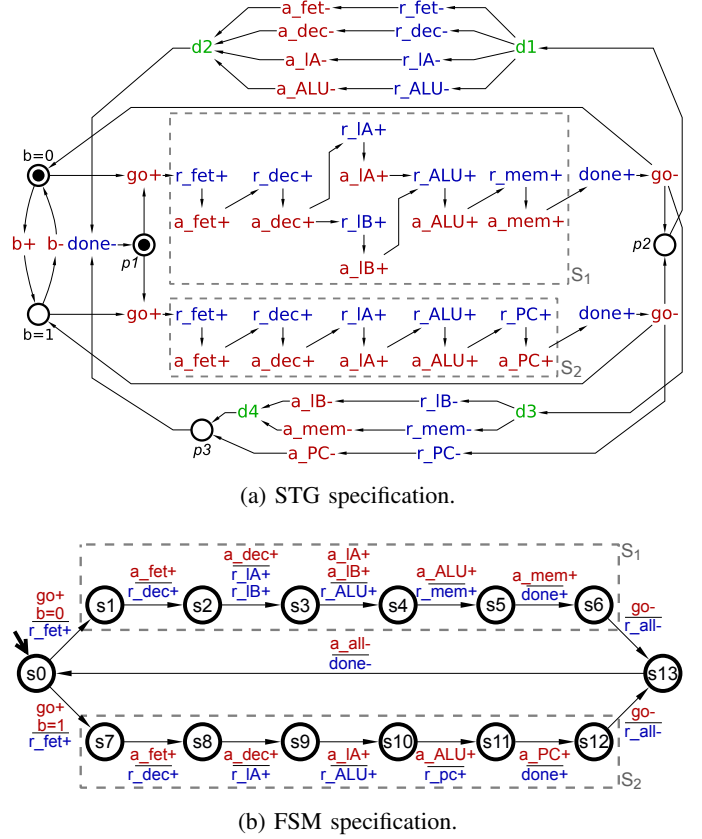


Fig. 4: The processor specification described as STG and FSM.

by De Micheli [18] that uses synchronous Finite State Machines (FSM) to derive controllers implemented as microcode memories or hard-wired control units.

In [6], CPOGs were compared to STGs and FSMs in terms of their compactness and ease of use when specifying asynchronous circuits. Below we highlight the main reasons for using CPOGs in the broader context of scenario-based synthesis.

- The separation of datapath (scenarios) and control (encoding) abstraction layers enables scenarios to remain unchanged when the encoding changes.
- Underlying partial orders can efficiently represent highly concurrent systems without incurring exponential state explosion.
- Scenario composition allows CPOGs to remain compact even when the size of the specification grows.
- Opportunity to minimise various design criteria (e.g. area, power, latency) by scenario encoding, which is our main goal.

In this paper, we compare CPOGs, STGs and FSMs practically by synthesising real scenario-based specifications. Our benchmarks, evaluated in Section VI, highlight that: (1) the STG methodology does not scale to specifications that include many scenarios, (2) the presented approach shows better results than the FSM methodology.

As an example of specifications, Fig. 4a and 4b show an STG and FSM model of the processor scenarios in Fig. 1a. In these figures: red transitions are the inputs of the designed

controller, blue ones are the outputs and green ones are dummy transitions (used to simplify the model). In the STG in Figure 4a, the two scenarios are mutually excluded via the choice place p_1 , and the causality dependencies of their operations are modelled via sequences of request/acknowledge transitions. The two scenarios are encoded by the same encoding used in Figure 2: $\{(s_1, 0), (s_2, 1)\}$ on one bit b . STG specifications are handled by the EDA tools Petrify [19] and MPSat [20], which synthesise asynchronous implementations using different algorithms. Petrify uses binary decision diagrams [17], while MPSat uses Petri net unfoldings [20].

In the FSM specification in Figure 4b, the two scenarios are selected via one bit b observed at the rising of the go signal, which starts the computation. Upon the completion of each scenario, all output requests r_all are reset, and the FSM returns to the initial state s_0 when all input acknowledgements a_all are also reset. Such specifications are described in VHDL as FSMs, and are handled by Design Compiler [21] to derive synchronous controllers. We applied concurrency reduction [6] to some of the considered FSM specifications not to incur state explosion, see *benchmarks* in [9].

B. CPOG scenario composition

The characteristics of the synthesised hardware controller correlate with the encoding selected [8]. In this work, we present a metric for extracting such a correlation, and an algorithm for approaching the efficient behavioural composition heuristically. In this section, we report other encoding techniques available for the efficient composition of scenarios into a CPOG.

The *Single-literal encoding* [16] is based on the graph colouring algorithm [22]. It finds an encoding under the constraint that each Boolean equation $\phi(z)$ of the synthesised CPOG can have at most 1 literal. The number of Boolean variables $|B|$ determines the colours available for solving the graph colouring problem, and can be increased above $\lceil \log_2 |S| \rceil$ automatically.

The *SAT-based encoding* [8] uses SAT solvers (CLASP [23] or MINISAT [24]) for minimising the synthesised CPOG Boolean equations. The number of Boolean variables $|B|$ for encoding is set by the user. In this paper, we set $|B| = \lceil \log_2 |S| \rceil$.

In Section VI, we show that the above approaches do not scale well to high number of scenarios ($|S| > 15$).

IV. SCENARIO COMPOSITION ALGORITHM

The optimal scenario encoding problem is NP-complete [16]. Finding the encoding that optimises a target hardware characteristic can be only achieved by synthesising and comparing all available encodings. In practice, this *exhaustive search* is infeasible due to the exponential growth of number of available encodings $|\mathcal{E}|$ when either the number of $|S|$ scenarios or $|C|$ codes increases, $|\mathcal{E}|$ defined in Section IV-A. This motivates the proposed *Heuristic encoding*, described in this section.

TABLE I: Symmetric encodings derivable from e_1 , e.g. e_2 is symmetric to e_1 , as it can be obtained by negating the Boolean variable b_1 in all the codes in e_1 .

| Scenarios | $e_1(b_1 b_2)$ | $e_2(\bar{b}_1 b_2)$ | $e_3(b_1 \bar{b}_2)$ | $e_4(\bar{b}_1 \bar{b}_2)$ |
|-----------|----------------|----------------------|----------------------|----------------------------|
| s_1 | 00 | 10 | 01 | 11 |
| s_2 | 01 | 11 | 00 | 10 |
| s_3 | 10 | 00 | 11 | 01 |
| s_4 | 11 | 01 | 10 | 00 |

A. Symmetric encodings

It is inefficient to inspect encodings that result in similar hardware implementations. This is the case for *symmetric encodings*, which are best explained by an example. The encoding $e_1 = \{(s_1, 00), (s_2, 01), (s_3, 10), (s_4, 11)\}$ has three symmetric encodings: e_2 , e_3 and e_4 , see examples in Table I.

A symmetric encoding can be obtained by negating one or more Boolean variables in all the codes of an encoding. We do not consider symmetric encodings, as the corresponding implementations differ only in terms of input inverters, which is insignificant. To rule out symmetric encodings, we always encode the first scenario by the first available code, e.g. the *zero code* $00..0$: $e = \{(s_1, 00..0), \dots, (s_{|S|}, c_{|S|})\}$ for all $e \in \mathcal{E}$.

The symmetry-breaking allows to restrict the universe of allowed encodings $\mathcal{E} = \{e_1, e_2, \dots, e_{|\mathcal{E}|}\}$ to the set that satisfies the two properties below:

- 1) All encodings are different: $e_i \neq e_j$ for $1 \leq i < j \leq |\mathcal{E}|$.
- 2) No two encodings e_i and e_j are symmetric.

Given $|S|$ scenarios and $|C|$ codes, the size of the universe of encodings is:

$$|\mathcal{E}| = \frac{(|C| - 1)!}{(|C| - |S|)!}$$

Note that at least $\lceil \log_2 n \rceil$ Boolean variables are needed to encode $|S|$ scenarios ($|B| \geq \lceil \log_2 |S| \rceil$). In this paper, we fix the number of such variables to the minimum, and restrict $|\mathcal{E}|$ using $|C| = 2^{\lceil \log_2 |S| \rceil}$ codes (see Section II-B2).

B. Composition constraints

In real-life systems, there are *composition constraints* that restrict the space of allowed encodings, for example due to backward compatibility requirements. Consider the two scenarios in Figure 1a, and assume that the following constraints must be met:

- The code of the arithmetic instruction (s_1) consists of an arbitrary 2-bit opcode, and two 3-bit operands A and B.
- The unconditional branch (s_2) consists of the fixed 00111 opcode, and a 3-bit branch offset.

The above requirements can be expressed with the *composition constraints* $G = \{(s_1, ??XXXXXX), (s_2, 00111XXXX)\}$, which uses 8 Boolean variables $B = \{b_1, b_2, \dots, b_8\}$.

- The arithmetic instruction 2-bit opcode ($b_1 b_2$) is denoted by $??$, where each $?$ is a *don't care* bit that becomes either 0 or 1 in the encoding. Each X is a *don't use* bit, which is not used for selecting a PO from those contained in the

CPOG. In fact, 6 bits are left unused for the two operands A ($b_3b_4b_5$) and B ($b_6b_7b_8$).

- The unconditional branch opcode ($b_1b_2b_3b_4b_5$) is fixed to 00111, the remaining 3 bits are left unused for the branch offset operand ($b_6b_7b_8$).

As shown in the above example, a *constraint* g is an assignment $g : B \rightarrow \{0, 1, ?, X\}$ of the set of Boolean variables B . Sets of constraints are used to express composition constraints $G = \{(s_1, g_1), (s_2, g_2), \dots, (s_{|S|}, g_{|S|})\}$.

The presented scenario composition algorithm handles composition constraints. As an example, the constraints set above can be satisfied by $\{(s_1, 10XXXXXX), (s_2, 00111XXX)\}$. On the other hand, $\{(s_1, 00XXXXXX), (s_2, 00111XXX)\}$ is an incorrect encoding as the code 00111000 selects both the instructions. Codes such as 00XXXXXX and 00111XXX are said *conflicting*. The implementation details for satisfying the composition constraints and finding an initial encoding prior to the heuristic optimisation are described in Algorithm 1.

Algorithm 1: Algorithm for satisfying the given composition constraints and finding the initial encoding.

```

1 Function findInitialEncoding( $B, G$ );
   Input      : Boolean variables  $B$ , a set of  $|S|$  constraints  $G$ .
   Output     : Encoding  $enc$  or error.
   Parameter:  $MAX(= 10)$  number of possible iterations.
2  $C \leftarrow (0^{|B|}, \dots, 1^{|B|})$ ;      // universe of codes
3  $enc \leftarrow (-1, \dots, -_{|S|})$ ;      // empty encoding
4 foreach  $i$  such that  $? \notin G[i]$  do
5   if  $G[i] \notin C$  then return error;
6    $enc[i] \leftarrow G[i]$ ;      // fully constrained  $s_i$ 
7    $C \leftarrow C \setminus G[i]$ ;
8 foreach  $i$  such that  $(? \in G[i]) \wedge (G[i] \neq ?^{|B|})$  do
9    $iteration \leftarrow 0$ ;
10  do
11     $code \leftarrow \text{randomAssignment}(G[i])$ 
12    while  $(code \notin C \wedge iteration++ < MAX)$ ;
13    if  $code \notin C$  then return error;
14     $enc[i] \leftarrow code$ ;      // partially constr.  $s_i$ 
15     $C \leftarrow C \setminus code$ ;
16 if  $|C| < |- \in enc|$  then return error;
17 if  $(G[0] = ?^{|B|}) \wedge (0^{|B|} \in C)$  then
18    $enc[0] \leftarrow 0^{|B|}$  // avoid symmetric encs.
19    $C \leftarrow C \setminus enc[0]$ ;
20 foreach  $i$  such that  $G[i] = ?^{|B|}$  do
21    $enc[i] \leftarrow \text{pickRandom}(C)$ ;
22   // unconstrained  $s_i$ 
23    $C \leftarrow C \setminus enc[i]$ ;
23 return  $enc$ ;

```

The function findInitialEncoding takes as **input** the Boolean variables B for encoding and the set of composition constraints G . The latter is an array of size $|S|$ whose indexes represent the scenarios and whose elements represent the constraints. As running example, we consider the constraints on 5 scenarios $\{(s_0, ???), (s_1, ??X), (s_2, ???), (s_3, 110), (s_4, ???), (s_5, ???)\}$ on the 3 variables $B = \{b_1, b_2, b_3\}$, which is represented by the array $G = (???, ??X, ???, 110, ???, ???)$.

Initially, the universe of codes C is initialised with $2^{|B|}$ codes (**line 2**), and the encoding enc with $|G|$ no-code symbols ($-$) as the encoding is initially empty (**line 3**). The array enc represents the initial encoding, its indexes represent the scenarios and its elements represent the codes. In the example, C and enc are:

$C = (000, 001, 010, 011, 100, 101, 110, 111)$

$enc = (-, -, -, -, -, -)$

In **lines 4-7**, the codes paired with fully constrained scenarios (i.e. $? \notin G[i]$) are checked to be contained in the universe of code C (**line 5**). Subsequently, these scenarios are encoded by the provided codes (**line 6**). The latter are removed from C (**line 7**), as they cannot be used for encoding other scenarios. C and enc becomes:

$C = (000, 001, 010, 011, 100, 101, 111)$

$enc = (-, -, -, \underline{110}, -, -)$

In **lines 8-15**, partially constrained codes (i.e. $? \in G[i] \wedge G[i] \neq ?^{|B|}$) are turned into codes, i.e. the function randomAssignment($G[i]$) turns their don't care bits ($? \in G[i]$) into binary values $\{0, 1\}$ randomly (**line 11**). Such resulting codes (*code*) encode scenarios s_i (**line 14**), and are subsequently removed from C (**line 15**). In the example, the constraint $\{??X\}$ is turned to $\{01X\}$, and is used to remove $\{010, 011\}$ from C . C and enc become:

$C = (000, 001, 100, 101, 111)$

$enc = (-, \underline{01X}, -, 110, -, -)$

The function randomAssignment can introduce conflicting codes. In the example, the constraint $G[1] = ??X$ cannot be turned to $11X$, as the latter is already used for encoding s_3 ($11X \notin C$). **Lines 10-12** can be repeated up to a MAX of 10 times to increase the probability of satisfying all constraints. If the constraint is still not satisfied, an *error* is returned (**line 14**).

In **line 16**, an *error* is returned if the number of codes left for encoding ($|C|$) is less than the scenarios that need to be encoded ($|- \in enc|$). In this case, more codes and bits B are required for encoding the given S under the constraints G .

In **lines 17-19**, the first scenario s_0 is encoded by the zero code if it is unconstrained ($G[0] = ?^{|B|}$) and if the zero code has not been used ($0^{|B|} \in C$). This is necessary for avoiding symmetric encodings. In the example, C and enc become:

$C = (001, 100, 101, 111)$

$enc = (\underline{000}, 01X, -, 110, -, -)$

In **lines 20-22**, unconstrained scenarios ($G[i] = ?^{|B|}$) are encoded randomly. Codes left are extracted by C , used to encode scenarios s_i (**line 21**), and subsequently removed from C (**line 22**). In the example, C and enc become:

$$C = (101)$$

$$enc = (000, 01X, \underline{100}, 110, 00\underline{1}, \underline{111})$$

The **output** of Algorithm 1 is the encoding enc , which satisfies G and can be optimised via the heuristics that we will describe shortly.

C. Heuristic cost function

The main idea of the heuristics is to *encode similar scenarios by similar codes*. Similarities between codes are determined using the classic *Hamming distance* metric [25]. Similarities between scenarios, on the other hand, are determined by referring to their partial order representation. Consider two scenarios $s_1 = (\mathcal{O}_1, \prec_1)$ and $s_2 = (\mathcal{O}_2, \prec_2)$. The distance between s_1 and s_2 is computed following the two rules below:

- 1) An operation $o \in \mathcal{O}_1$ counts as a difference if $o \notin \mathcal{O}_2$.
- 2) A dependency $(o_s \prec o_t) \in \prec_1$ counts as a difference if $(o_s \prec o_t) \notin \prec_2$, and if it connects two operations which are both present in the operation sets of the two scenarios: $(o_s \in \mathcal{O}_1 \wedge o_s \in \mathcal{O}_2) \wedge (o_t \in \mathcal{O}_1 \wedge o_t \in \mathcal{O}_2)$.

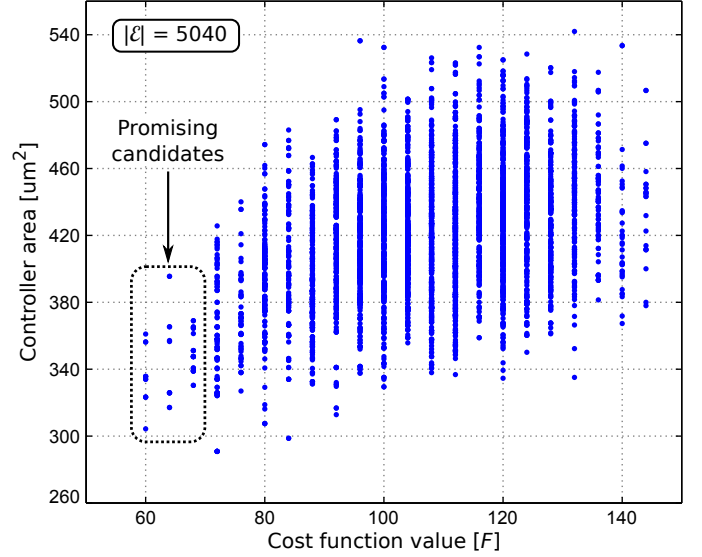
Distances between pairs of scenarios are elements of the *Scenario Distance Matrix* SD . Distances between pairs of codes are elements of the *Code Distance matrix* CD . Both SD and CD have size $|S| \times |S|$, where $|S|$ is the size of the scenario specification. Elements SD_{ij} and CD_{ij} represent the number of differences between the i_{th} and j_{th} scenarios and codes, respectively, in an encoding $e = \{(s_i, c_i), (s_j, c_j), \dots, (s_{|S|}, c_{|S|})\}$. These matrices are used to evaluate encodings heuristically via the below *cost function*:

$$\mathcal{F}(S, e) = \sum_{0 \leq i < j \leq |S|} (SD_{ij} - CD_{ij})^2 \quad (1)$$

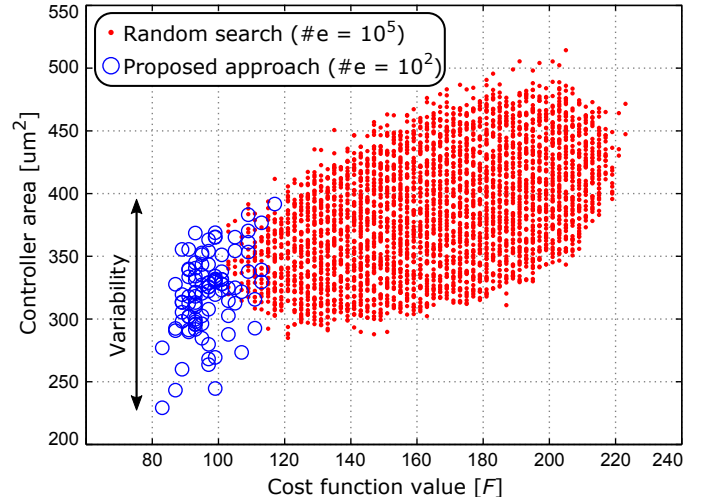
Intuitively, minimising \mathcal{F} means encoding similar scenarios with similar codes. We evaluated the cost function \mathcal{F} empirically, by analysing several scenario specifications.

As an example, Figure 5a shows the analysis of a subset of 8 scenarios of the Intel 8051 [7] scenario specification, where the universe of encoding \mathcal{E} is fully inspected, and 5040 controllers are synthesised with a 90 nm technology library [9]. The size of the controllers is plotted against the heuristic value \mathcal{F} of the corresponding encodings. Figure 5b, in turn, shows the analysis of the scenario specification of the Arm Cortex M0+ [11], composed of 11 scenarios. In this figure, 10^2 controllers produced by the proposed algorithm, described in Section IV-D, are compared to 10^5 controllers produced by encoding scenarios randomly.

The two figures highlight the existence of a correlation between the controller area and \mathcal{F} , and suggest the following two claims:



(a) All controllers (\mathcal{E}) plotted with respect to the cost function \mathcal{F} (Intel 8051 [7], subset of 8 scenarios).



(b) 10^2 heuristic and 10^5 random controllers are compared, and plotted with respect to \mathcal{F} (Arm Cortex M0+ [11], 11 scenarios).

Fig. 5: The presented cost function is studied over two benchmarks.

- the likelihood of synthesising efficient implementations is higher where \mathcal{F} is lower, see *Promising candidates* in Fig. 5a;
- the likelihood of synthesising efficient implementations is proportional to the number of encodings inspected, due to the inaccuracy of the heuristics, see *Variability* span in Fig. 5b.

D. The heuristic encoding algorithm

The presented heuristic algorithm is based on the cost function \mathcal{F} , and on the below implementation of the simulated annealing (SA) [26]. The latter is a heuristic method for solving optimisation problems where a function must be minimised in a large search space. The algorithm pseudo-code is shown in Algorithm 2.

Algorithm 2: The presented Heuristic encoding algorithm.

```

1 Function heuristicEncoding ( $B, S, G$ )
   Input      : Boolean variables  $B$ , scenarios  $S$  and
                 their constraints  $G$ .
   Output     : Heuristic encoding  $enc$ .
   Parameters:  $t_0 = 10, a = 0.996, t_e = 0.1$ 

2  $enc \leftarrow \text{findInitialEncoding}(B, G);$ 
3  $enc_{best} \leftarrow enc;$ 
4  $C \leftarrow (0^{|B|}, \dots, 1^{|B|}) \setminus enc[0];$ 
5 while ( $t_0 > t_e$ ) do
6    $enc_{next} \leftarrow enc;$ 
7    $i \leftarrow \text{pickRandomScenario}(1 \leq i < |S|);$ 
8    $code \leftarrow \text{pickRandomCode}(C);$ 
9   if  $\exists j$  such that  $code \in enc_{next}[j]$  then
10     $enc_{next}[i] \leftrightarrow enc_{next}[j];$ 
11   else
12     $enc_{next}[i] \leftarrow code;$ 
13   if  $\forall i$  satisfy( $enc_{next}[i], G[i]$ ) then
14     if  $\mathcal{F}(S, enc_{next}) < \mathcal{F}(S, enc_{best})$  then
15        $enc_{best} \leftarrow enc_{next};$ 
16      $d \leftarrow \mathcal{F}(S, enc_{next}) - \mathcal{F}(S, enc);$ 
17      $v \leftarrow \text{pickRandomValue}(0 \leq v < 1);$ 
18     if  $v < e^{-\frac{d}{t_0}}$  then
19        $enc \leftarrow enc_{next};$ 
20    $t_0 \leftarrow t_0 \times a;$       // cool down  $t_0$  by  $a$ 
21 return  $enc_{best};$ 

22 Function satisfy ( $c, g$ )
   Input      : A code  $c$ , and a constraint  $g$ .
   Output     : Boolean values True or False.

23 if  $|c| \neq |g|$  then return False;
24 foreach  $1 \leq i \leq |g|$  do
25   if  $(c[i] = 1) \wedge (g[i] = 0)$  then return False;
26   if  $(c[i] = 0) \wedge (g[i] = 1)$  then return False;
27   if  $(c[i] \neq X) \wedge (g[i] = X)$  then return False;
28 return True;

```

The **inputs** of the function *heuristicEncoding* are the Boolean variables B , the scenarios S and constraints G . We continue the running example used for the Algorithm 1, where constraints $G = \{???, ??X, ???, 110, ???, ???\}$ were turned to $enc = \{000, 01X, 100, 110, 001, 111\}$ by the *findInitialEncoding* function (**line 2**). The encoding is also copied into enc_{best} (**line 3**), which represents the best encoding found during the SA search.

Simulated annealing parameters were calibrated experimentally. The initial temperature is $t_0 = 10$. The cooldown factor alpha is $a = 0.996$, and the ending temperature is $t_e = 0.1$. These parameters can be modified for increasing or decreasing the number of iterations for the SA optimisation.

Line 4 initialises the universe of codes C . The code of

the first scenario $enc[0]$ is removed for avoiding symmetric encodings.

Lines 5-21 minimise the initial encoding heuristic value $\mathcal{F}(S, enc)$ by repeatedly swapping pairs of codes in the encoding, until the initial temperature t_0 reaches the ending temperature t_e (**line 5**). **Line 6** stores the current encoding enc into the the next encoding enc_{next} . **Lines 7-8** select a random scenarios s_i in enc ($1 \leq i < |S|$), and a random code c_j in C , respectively. Such indexes are used for swapping codes in enc_{next} (see **lines 9-12**). In the example, if $i = 4$ and $j = 7$, the fourth scenario (encoded by 001) is swapped with the code 111 (which identifies s_5 in enc). enc and enc_{next} become:

$$enc = (000, 01X, 100, 110, \underline{001}, \underline{111})$$

$$enc_{next} = (000, 01X, 100, 110, \underline{111}, \underline{001})$$

The next encoding enc_{next} is considered if it satisfies the composition constraints G (**line 13**). The function *satisfy* (**lines 22-28**) checks that the bit size of the code matches the bit size of the constraint (**line 23**), and that the bits constrained by $\{0, 1, X\}$ hold these values in the final code (**lines 24-27**). Notice that bits constrained by $?$ do not need to be checked, as both logic values $\{0, 1\}$ satisfy such constraints.

In **lines 14-15**, enc_{next} replaces the best encoding enc_{best} found during the SA optimisation if the former has a lower heuristic value than the latter, i.e. $\mathcal{F}(S, enc_{next}) < \mathcal{F}(S, enc_{best})$. In **lines 16-19**, enc_{next} also replaces enc either if the former has a lower heuristic value than the latter (i.e. $v < e^{-\frac{d}{t_0}}$ for all $0 \leq v < 1 \wedge d \leq 0$), or if the extracted random value v is lower than $e^{-\frac{d}{t_0}}$, with $d > 0$. In the second case, a worse encoding (with a higher \mathcal{F}) replaces enc .

The randomness allows the *heuristicEncoding* to return a different enc_{best} (**output**) at every execution. The solution space is connected, as all $e \in \mathcal{E}$ are reachable by a set of swap moves.

The current implementation of the algorithm is run in a single thread of execution. However, multiple instances of the *heuristicEncoding* function can be run on multiple threads, resulting in several encodings to be produced concurrently. The parallelisation of the presented algorithm is left as future research.

E. Correctness

An encoding enc , constrained by composition constraints G , is said to be *correct* if:

- 1) enc satisfies G for all $1 \leq i \leq |S|$. I.e. Every code $enc[i]$ can be derived from $G[i]$ by substituting every $?$ by either 0 or 1.
- 2) the encoding enc does not contain conflicting codes, which do not identify scenarios univocally (see Section IV-B);

Whenever Algorithm 1 terminates, a *correct* encoding enc is returned by construction. I.e. the result enc is constructed by selecting the codes for encoding from the universe C , which only contains valid codes being derived by the number of bits $|B|$ selected for encoding (**line 2**). Fully and partially constrained scenarios are always encoded by codes derived

by their constraints, see **lines 6 and 9-14**, respectively. Thus the resulting *enc* always satisfies the constraints G . Also, overlapping codes cannot be introduced in the final encoding result *enc*: a code is always removed from C when it is used to encode a scenario and thus cannot be reused to encode a different scenario, see **lines 7, 15, 19 and 22**.

On the other hand, Algorithm 1 generates an *error* if the constraints G cannot be met for any of the following reasons:

- The user introduces overlapping constraints, see **line 5**.
- A partially constrained code is not turned into a code c left for encoding ($c \notin C$) in any of the MAX iterations, see **lines 9-13**. An optimal solution would be to run an exhaustive search, which we avoid to reduce the algorithmic complexity.
- The number of codes $|C|$ is not enough for encoding a set of scenarios with size $|G|$ with the given constraints G , see **line 16**.

With regards to Algorithm 2, the function `heuristicEncoding` handles the output of the previous function *enc*, and advances to *enc_{best}* through a sequence of swap moves that inspects many intermediate encodings *enc_{next}*. Given a *correct enc* (see **line 2**), each intermediate encoding *enc_{next}* derived by a swap (see **lines 7-12**) is always an encoding with no conflicting codes (i.e. code swap does not introduce encoding conflicts). Intermediate encodings can replace *enc_{best}* only if they satisfy the constraints G (**lines 13 and 22-27**). Consequently, *enc_{best}* is also *correct* by construction. The two algorithms always terminate, as there are not infinite loops.

F. Time complexity analysis

The function `findInitialEncoding` is constituted by a sequence of three loops. The first one (**lines 4-7**) encodes fully constrained scenarios by moving their codes into the encoding *enc*. Its complexity only depends on the number of fully constrained codes introduced: $\mathcal{O}(|S|)$. The second loop (**lines 8-15**) encodes partially constrained scenarios by looping over the bits $|B|$ of each constraint, in order to flip every ? to $\{0, 1\}$. Thus, its complexity is: $\mathcal{O}(|S| \cdot |B|)$. The third loop (**lines 20-22**) makes use of the function `pickRandom` ($\mathcal{O}(1)$) to extract codes left in the code universe C and encode the unconstrained scenarios. Its complexity depends on the constraints: $\mathcal{O}(|S|)$. Consequently, the complexity of the Algorithm 1 (\mathcal{A}_1) comes from the second loop. In this paper, we assume that $|B| = \lceil \log_2 |S| \rceil$, hence the below equation:

$$\mathcal{O}(\mathcal{A}_1) = \mathcal{O}(|S|) + \mathcal{O}(|S| \cdot |B|) + \mathcal{O}(|S|) = \mathcal{O}(|S| \cdot \log |S|)$$

On the other hand, the function `heuristicEncoding`, excluding the internal `findInitialEncoding` function in line 2, is constituted by a loop that implements an *exponential multiplicative cooling* strategy of the simulated annealing algorithm (SA) [26], i.e. an initial temperature t_0 is multiplied by a constant factor a at each iteration, until an ending temperature t_e is reached. This causes a fixed number of iterations n that can be tweaked by modifying these parameters. At each iteration of the SA, the most computationally expensive statements are in **lines 13 and 22-28**: where the *enc_{next}* is checked against the constraints G , and in **lines 14**

and **16**: where the function \mathcal{F} has to be computed. The former has a complexity of $\mathcal{O}(|S| \cdot \lceil \log_2 |S| \rceil)$, as the encoding has to be checked for every bit of each constraint. The latter has a complexity of $\mathcal{O}(|S|^2)$, see Formula 1. Consequently, Algorithm 2 (\mathcal{A}_2) has the following time complexity:

$$\mathcal{O}(\mathcal{A}_2) = n \cdot [\mathcal{O}(|S| \cdot \lceil \log_2 |S| \rceil) + \mathcal{O}(|S|^2)] = \mathcal{O}(n \cdot |S|^2)$$

This analysis disregards the implementation details of the further set of functions (e.g. `pickRandomScenario`) that the proposed algorithm rely on. However, these additional functions do not increase the above time complexity if implemented reasonably.

V. DESIGN AUTOMATION

The design methodology described in Section II-B is implemented in the EDA tool SCENCO [9], which stands for *SCENario ENCOder*. It features the following scenario encoding algorithms.

- 1) **Exhaustive search** fully explores the universe of encodings \mathcal{E} .
- 2) **SAT-based encoding** and **Single-literal encoding** are described in Section III.
- 3) **Heuristic encoding** is the proposed algorithm (Section IV).
- 4) **Random search** encodes scenarios randomly.
- 5) **Sequential encoding** assigns codes sequentially, i.e. $\{(s_1, 000), (s_2, 001), (s_3, 010), \dots\}$.

SCENCO relies on *Espresso* [27] for Boolean minimisation, and *Abc* [28] for technology mapping, the gate library is specified in the GenLib format [29]. *Abc* is also used for producing synthesised controllers in the Verilog file format. SCENCO also uses *Clasp* [23] and *MiniSAT* [24] SAT solvers for supporting the SAT encoding.

SCENCO graphical user interface is described in [30], Figure 6 shows and describes an example of the applied design methodology in WORKCRAFT.

A. Interface synthesis

The synthesis of the interface between the controller and the datapath has been automated in the EDA tool [9], relying on the ideas elaborated in [31] and summarised below.

The controller can be interfaced either with asynchronous datapath modules, relying on the request/acknowledge handshake, and to synchronous modules using *matched delays* [32], which produce acknowledgement signals after a chosen delay. In turn, since the controller resets request signals only at the end of each scenario execution, *decouple* and *merge* [31] are needed to release datapath modules immediately after they acknowledge their completion. Also, *merge* is used when a module is executed multiple times within a scenario, see a schematic of the interface in Figure 7.

The developed tool [9] takes as input the datapath modules in the form of Verilog, and interfaces them to the synthesised controller automatically. The produced Verilog file contains the final system implementation, see Figure 3.

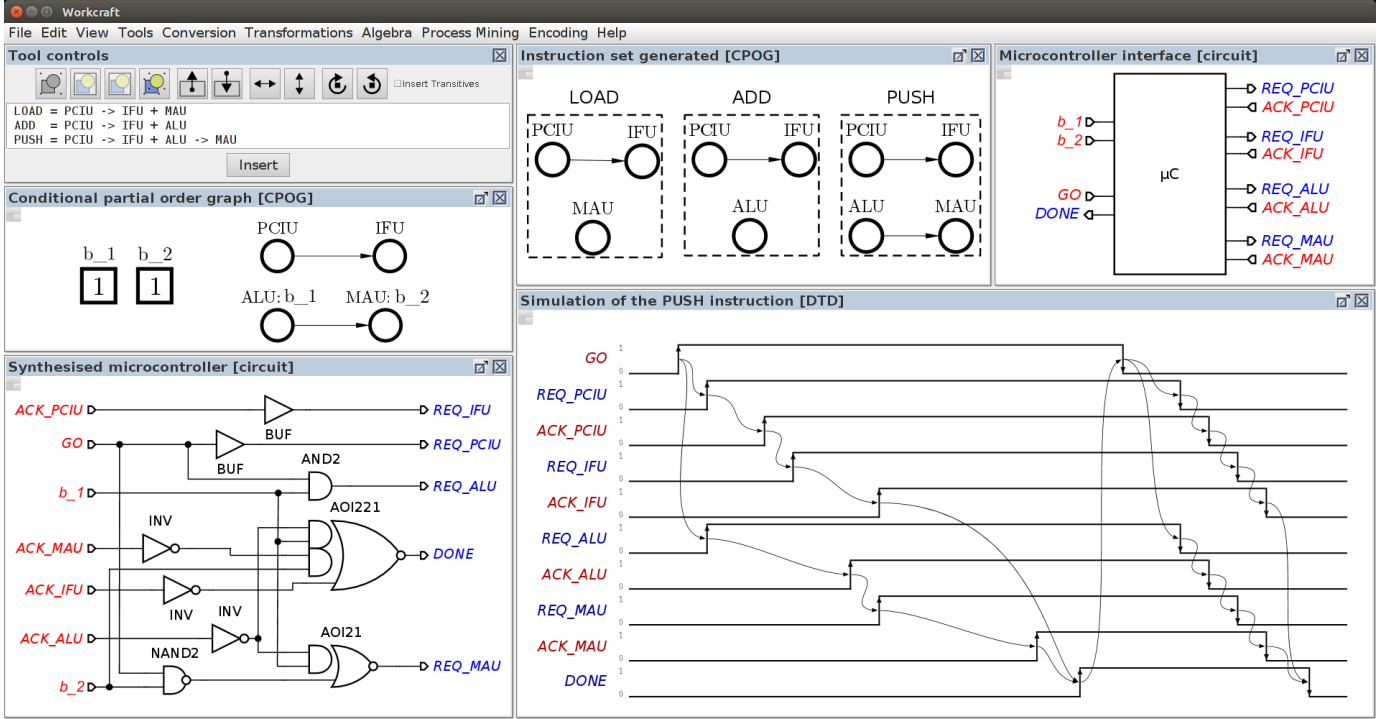


Fig. 6: The methodology described in Section II-B is implemented in WORKCRAFT. **Scenario specification:** three scenarios (LOAD, ADD and PUSH) are introduced in the form of text (*Tools controls* window), and are parsed and converted into partial orders (*Instruction set generated [CPOG]* window). Scenarios can be also entered or edited graphically. **Scenario encoding:** scenarios are encoded using the chosen algorithm (*Encoding* menu). The encoding $\{(\text{LOAD}, 01), (\text{ADD}, 10), (\text{PUSH}, 11)\}$ is found in the example. **Composition:** The CPOG is generated automatically (*Conditional partial order graph [CPOG]* window). **Hardware synthesis:** the hardware controller is synthesised from the CPOG (*Synthesised controller [circuit]* window). The controller interface is highlighted in the window *Controller interface [circuit]*. The hardware controller can be simulated and formally verified using other tools available in the WORKCRAFT framework. For example, the window *Simulation of the PUSH instruction [DTD]* shows a simulation of the PUSH scenario of the controller.

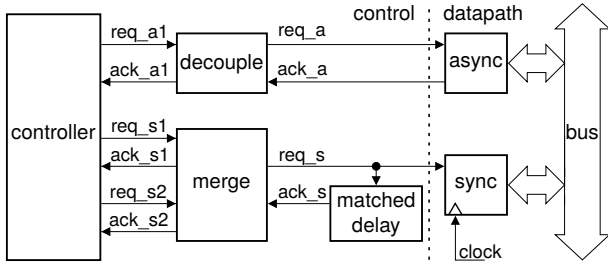


Fig. 7: Interface between controller and datapath.

VI. ALGORITHM AND TOOL VALIDATION

We validate the presented algorithm and tool over a set of benchmarks coming from three domains: ad-hoc controllers, processor instruction sets and process mining in Sections VI-B, VI-C and VI-D, respectively. Experimental results are compared with existing scenario composition approaches on all benchmarks, and also with the behavioural synthesis methodologies based on STGs and FSMs in the processor benchmarks. All used benchmarks can be found online (see *benchmarks* folder in [9]), and can be displayed and run in WORKCRAFT [10].

A. Configuration and notation

We run our experiments on an Intel-i7-3610QM 2.30 GHz CPU, with 8 GB DDR 1600 MHz RAM Memory. Benchmarks are:

- 1) Specified in the form of partial orders in WORKCRAFT [10], and synthesised by the presented tool SCENCO [9].
- 2) Specified as STGs in WORKCRAFT, and synthesised by Petriify [19] and MPSat [20].
- 3) Specified as synchronous FSMs in VHDL, and synthesised by Synopsys Design Compiler [21].

The same 90nm gate library is used for technology mapping.

For presenting benchmark results, we use the following notation. $\#e$ denotes the number of encodings generated and synthesised by the proposed algorithm. The smallest controller out of these is shown as result. **Area** ($|B|$) denotes the area [μm^2] of the resulting controller, with the number of bits used for encoding in brackets. **RT** denotes the tool runtime [s], which is the time that goes from parsing the specification to obtaining the final implementation. We only consider results produced within a runtime of 1 hour, denoted in turn as timeout **TO**. Finally, we use the **dash** character ‘—’ when a behavioural synthesis approach cannot be applied to

TABLE II: Comparison of CPOG scenario encoding algorithms over the ad-hoc controller benchmarks. *Units of measure:* Area ($|B|$) = [μm^2] (number of bits).

| Model | $ S $ | Exhaust. | Single-lit. | SAT-based | Proposed #e = 10 |
|-------------|-------|----------|-----------------|-----------|---------------------|
| Buck contr. | 4 | 266 (2) | 261 (3) | 266 (2) | 266 (2) |
| Rec. pipe. | 13 | TO | 357 (12) | TO | 481 (4) |

a benchmark due to a technical issue, see textual description for an explanation.

Controllers derived by FSMs and STGs include sequential components (registers and C-elements, respectively) for holding system states. In the results, we only consider the combinational part of the controllers for not penalising them in the comparison.

B. Ad-hoc controllers

The first set of benchmarks includes an on-chip power management controller of a buck converter [33], and an asynchronous controller for the reconfigurable pipeline of a dataflow processor [34].

The power management controller is required to regulate the activation of the PMOS (gp) and NMOS (gn) transistors in response to three signals coming from sensors within the power regulator: over-current (oc), under-voltage (uv) and zero-crossing (zc). The two transistors must never be on at the same time to avoid a short circuit. Two of the four scenarios that compose the power management controller are shown in Figure 8a and described below.

Over-current scenario: when the oc condition is detected (event oc+), the PMOS transistor must be switched off (event gp-). Afterwards, the NMOS transistor must be switched on (event gn+).

Zero-crossing followed by under-voltage scenario: If zc is detected before uv, the NMOS transistor must be switched off (event gn-). The two transistors must stay off until the arrival of the uv condition. Afterwards, the PMOS transistor must be switched on (event gp+).

The asynchronous dataflow processor contains a 16-stage reconfigurable pipeline for statistical analysis of data streams. Its controller manages the energy-quality of the result by controlling the number of active pipeline stages. It is an important case study, as it was fabricated in an ASIC and tested [34]. Three of its 13 scenarios that compose the reconfigurable pipeline are specified below in the text-form, i.e. the scenario s_1 activates 4 pipeline stages, the s_2 activates 5 stages, up to the scenario s_{13} that activates all 16 stages of the pipeline:

$$s_1 = \text{stage1} \rightarrow \text{stage2} \rightarrow \text{stage3} \rightarrow \text{stage4}$$

$$s_2 = \text{stage1} \rightarrow \text{stage2} \rightarrow \text{stage3} \rightarrow \text{stage4} \rightarrow \text{stage5}$$

\vdots

$$s_{13} = \text{stage1} \rightarrow \text{stage2} \rightarrow \dots \rightarrow \text{stage16}$$

Evaluation: Table II shows the results upon application of the state-of-the-art encoding algorithms. The Single-literal encod-

ing produces a 1.9% smaller *buck controller* in comparison to other approaches, and uses one more variable than needed ($|B| = 3$). On 2 variables, the optimal controller is generated by the Exhaustive search by definition. Such a controller is also achieved by the SAT-based and by the proposed Heuristic algorithm.

The *reconfigurable pipeline controller* is not produced within the considered timeout by the Exhaustive and the SAT-based algorithms, due to the complexity of the corresponding scenario specification. The Single-literal controller is $\simeq 25.8\%$ smaller than the controller produced by the proposed encoding technique, and uses $3\times$ more variables. In [34], we implemented the controller produced by the proposed encoding technique, as the final design was constrained by the pins of the external package.

The runtime of the tool for processing the above benchmarks is always less than 1 s.

C. Processor instruction sets

The second set of benchmarks includes different subsets of instructions of the ARM Cortex M0+ [11], Texas Instruments MSP430 [8] and Intel 8051 [7], [35]. These processor specifications were derived by analysing their corresponding ISA reference manuals, and identifying classes of instructions (scenarios) that share similar functionalities and addressing modes.

In regards to the design of real processors, the above manual scenario extraction approach is not ideal to obtain accurate specifications. However, recent research on specification languages for processor architectures (see Section II-B1) enables to fully specify the behaviour of modern systems comprising hundreds of instructions, and to derive accurate specifications for synthesising real processors. In this context, the presented algorithm is important as it scales well to hundreds of scenarios (as we show in Section VI-D) making the CPOG-methodology suitable to the design of such modern systems.

The ARM Cortex M0+ scenario specification is fully described in [11]. This processor has an ISA constituted of 68 instructions. The specification composed of 11 scenarios and 6 datapath modules (scenario operations) models 61 of these instructions. As an example, two scenarios of the specification are shown in Figure 8b and described below.

Load (reg.) covers the *LDR (reg.)* instruction. The ALU operation computes the memory address, the MAU loads a value from the memory and stores it into a specified register. The IFU fetches a new processor instruction.

Arit/Log (Imm.) covers arithmetical, logical and data transfer instructions with immediate addressing, e.g. *ADD (imm.)*, *LSR (imm.)*. An immediate value is fetched from the instruction register (PCIU \rightarrow IFU), and used as operand for the selected operation (ALU). The result is stored into a specified register. The ALU operation is executed concurrently with the program counter incrementation (PCIU/2). The resulting *PC* is used for fetching a new instruction (IFU/2).

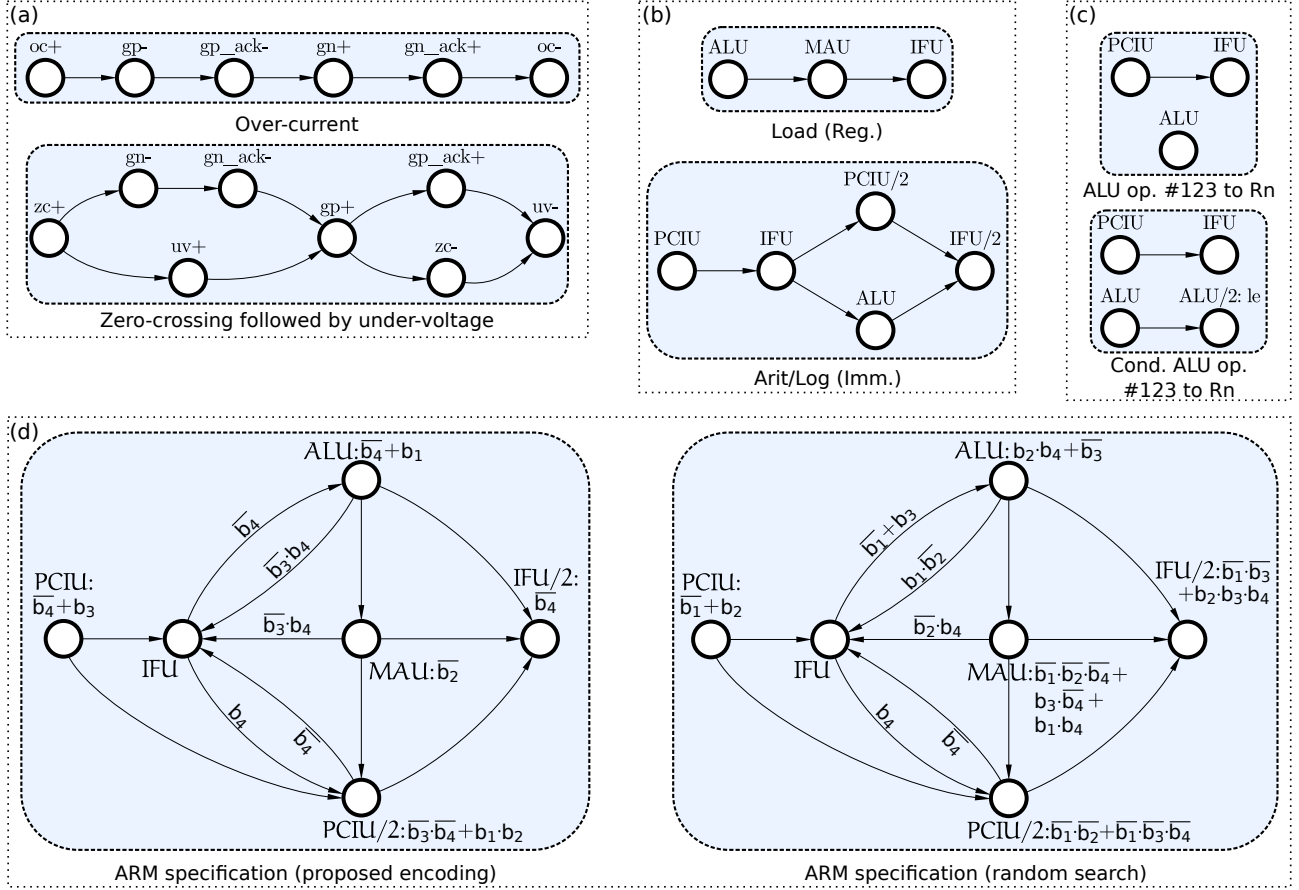


Fig. 8: **(a)** shows 2 scenarios of the Buck controller specification [33]. **(b)** shows 2 scenarios of the ARM CORTEX M0+ [11]. **(c)** shows 2 scenarios of the TI MSP430 specification [8], the scenario *Cond. ALU op. #123 to Rn* is a conditional scenario in the form of CPOG, due to the conditional operation $ALU/2$. **(d)** shows the CPOGs synthesised from full scenario specification in [11], i.e. the CPOG derived by the proposed encoding is on the left-hand side, and the one derived by the Random search is on the right-hand side.

The Texas Instruments (TI) MSP430 scenario specification has been introduced in [8]. The specification composed of 8 scenarios and 7 datapath modules models the full instruction set composed of 51 instructions. This benchmark is important as some of its scenarios have conditional elements. As an example, two of its scenarios are shown in Figure 8c and described below:

ALU op. #123 to Rn executes an arithmetic operation between two general purpose registers $\{A, B\}$, and writes the result back into one of them (ALU). Operations $PCIU \rightarrow IFU$ fetch a new instruction concurrently.

Cond. ALU op. #123 to Rn executes the above arithmetic operation between two general purpose registers $\{A, B\}$ conditionally ($ALU/2$), on the condition $le = A < B$ set by the ALU . The flag le is input to the synthesised controller, and is used for managing the activation of $ALU/2$.

The second scenario is said to be *conditional*, and can be described in the form of a CPOG. Conditional scenarios can be composed regularly with other scenarios, see [16] for further details.

The Intel 8051 specification supported the design of an

asynchronous version of this processor [35]. It comprises 37 scenarios and 17 datapath modules that model 255 processor instructions. It is important to the CPOG validation, as it contains $3 \times$ more scenarios and $2 \times$ more operations than the other processor benchmarks.

Evaluation: Table III shows the results of the applied state-of-the-art CPOG encoding algorithms to the described set of processor benchmarks. This set is also used to compare the proposed algorithm based on CPOGs to the methodologies based on FSMs and STGs, as it is the most diverse set being characterised by (1) specifications of different sizes (from 4 to 37 scenarios), (2) specifications with conditional scenarios (see TI MSP430), (3) specifications comprising a different number of datapath modules (from the ARM processor with 6, to the Intel with 17). For these reasons, it is able to highlight the characteristics of all used approaches to behavioural synthesis.

The *Exhaustive search* produces the smallest instruction decoders using $\lceil \log_2 |S| \rceil$ variables. In practice, it is applicable to specifications that contains up to 8 scenarios, as its runtime increases exponentially with the specification size.

The *Single-literal* encoding produces the smallest instruction decoders in most of the cases when it does not exceed the

TABLE III: The proposed algorithm is compared with existing CPOG composition techniques, and with the FSM and STG synthesis approaches, over 26 processor instruction set benchmarks. Bold results are the smallest controllers for each model. *Units of measure:* Area ($|B|$) = $[\mu m^2]$ (number of bits), Runtime (RT) = $[s]$.

| Model | $ S $ | Exhaustive | | Single-literal | | SAT-based | | Proposed #e = 10 | | Proposed constr., #e = 10 | | FSM (seq. encod.) dc_shell | | STG (one-hot encod.) Petrify / MPSat | |
|--------------------------|-------|----------------|------|----------------|----|----------------|----|---------------------|----|------------------------------|----|-------------------------------|----|---|----------|
| | | Area ($ B $) | RT | Area ($ B $) | RT | Area ($ B $) | RT | Area ($ B $) | RT | Area ($ B $) | RT | Area ($ B $) | RT | Area ($ B $) | RT |
| ARM Cortex M0+ | 4 | 162 (2) | 1 | 177 (4) | 1 | 162 (2) | 1 | 162 (2) | 2 | 167 (2) | 1 | 193 (2) | 5 | 265 / 200 (4) | 51 / 93 |
| | 5 | 179 (3) | 162 | 202 (5) | 1 | 201 (3) | 1 | 182 (3) | 2 | 192 (3) | 2 | 227 (3) | 7 | 243 / 242 (5) | 47 / 295 |
| | 6 | 201 (3) | 524 | 198 (5) | 1 | 225 (3) | 3 | 201 (3) | 2 | 241 (3) | 2 | 303 (3) | 7 | — / 263 (6) | — / 270 |
| | 7 | 209 (3) | 1051 | 198 (5) | 1 | 225 (3) | 3 | 226 (3) | 2 | 249 (3) | 2 | 316 (3) | 7 | — / 319 (7) | — / 409 |
| | 8 | 180 (3) | 1005 | 165 (5) | 2 | 224 (3) | 2 | 224 (3) | 2 | 230 (3) | 2 | 345 (3) | 7 | — / 343 (8) | — / 981 |
| | 9 | TO | TO | 220 (5) | 1 | 269 (4) | 1 | 224 (4) | 2 | 235 (4) | 2 | 457 (4) | 6 | — / 456 (9) | — / 2232 |
| | 10 | TO | TO | 218 (5) | 1 | 232 (4) | 1 | 241 (4) | 2 | 252 (4) | 2 | 467 (4) | 10 | — / TO | — / TO |
| | 11 | TO | TO | 212 (5) | 1 | 246 (4) | 2 | 249 (4) | 2 | 279 (4) | 2 | 498 (4) | 7 | — / TO | — / TO |
| Texas Instruments MSP430 | 4 | 154 (2) | 1 | 177 (4) | 1 | — | — | 154 (2) | 2 | 171 (2) | 1 | 288 (2) | 5 | 292 / TO (4) | 119 / TO |
| | 5 | 174 (3) | 162 | 181 (6) | 1 | — | — | 180 (3) | 2 | 193 (3) | 1 | 294 (3) | 7 | — / TO | — / TO |
| | 6 | 189 (3) | 489 | 191 (7) | 2 | — | — | 205 (3) | 2 | 235 (3) | 1 | 384 (3) | 7 | — / TO | — / TO |
| | 7 | 252 (4) | 1059 | 223 (8) | 1 | — | — | 276 (3) | 2 | 293 (3) | 2 | 376 (3) | 7 | — / TO | — / TO |
| | 8 | 299 (4) | 1145 | 304 (8) | 1 | — | — | 321 (3) | 2 | 345 (3) | 2 | 390 (3) | 6 | — / TO | — / TO |
| Intel 8051 | 4 | 175 (2) | 1 | 166 (3) | 1 | 175 (2) | 1 | 175 (2) | 2 | 175 (2) | 1 | 240 (2) | 7 | — / TO | — / TO |
| | 5 | 175 (3) | 165 | 170 (4) | 1 | 189 (3) | 1 | 178 (3) | 2 | 193 (3) | 2 | 335 (3) | 7 | — / TO | — / TO |
| | 6 | 214 (3) | 521 | 196 (5) | 1 | 235 (3) | 1 | 226 (3) | 2 | 224 (3) | 1 | 377 (3) | 7 | — / TO | — / TO |
| | 7 | 234 (3) | 1111 | 242 (6) | 1 | 239 (3) | 1 | 240 (3) | 2 | 260 (3) | 1 | 422 (3) | 7 | — / TO | — / TO |
| | 8 | 295 (3) | 1145 | 267 (7) | 1 | TO | TO | 302 (3) | 2 | 312 (3) | 2 | 498 (3) | 7 | — / TO | — / TO |
| | 9 | TO | TO | 286 (8) | 1 | 303 (4) | 13 | 322 (4) | 2 | 347 (4) | 2 | 519 (4) | 10 | — / TO | — / TO |
| | 10 | TO | TO | 464 (9) | 2 | TO | TO | 335 (4) | 2 | 368 (4) | 2 | 565 (4) | 10 | — / TO | — / TO |
| | 15 | TO | TO | TO | TO | TO | TO | 679 (4) | 2 | 736 (4) | 2 | 943 (4) | 12 | — / TO | — / TO |
| | 20 | TO | TO | TO | TO | TO | TO | 822 (5) | 2 | 842 (5) | 2 | 1187 (5) | 15 | — / TO | — / TO |
| | 25 | TO | TO | TO | TO | TO | TO | 1147 (5) | 3 | 1202 (5) | 3 | 1519 (5) | 17 | — / TO | — / TO |
| | 30 | TO | TO | TO | TO | TO | TO | 1376 (5) | 4 | 1450 (5) | 4 | 1879 (5) | 20 | — / TO | — / TO |
| | 35 | TO | TO | TO | TO | TO | TO | 1690 (6) | 4 | 1741 (6) | 4 | 2159 (6) | 21 | — / TO | — / TO |
| | 37 | TO | TO | TO | TO | TO | TO | 1879 (6) | 4 | 2037 (6) | 4 | 2336 (6) | 20 | — / TO | — / TO |

time limit. However, synthesised decoders might not be applicable to real processors, as the code size $|B|$ is fixed by the algorithm rather than by the processor (op)code specifications.

The *SAT-based encoding* produces decoders with an average overhead of $\simeq 7.4\%$ in comparison to Exhaustive decoders. The current implementation does not support scenarios in the form of CPOGs (see missing results — in the TI MSP430 rows). The runtime of the SAT-based and Single-literal approaches increase exponentially (exceeding the timeout) when $|S|$ grows.

On average, the *Proposed encoding* produces implementations with an area overhead of $\simeq 4.5\%$ in comparison to optimal solutions. It scales to higher number of scenarios (see Intel 8051 results), and supports scenarios in the form of CPOGs (see TI MSP430 results). The runtime is always within the timeout. As an example, Figure 8d shows the ARM system specifications obtained by composing its 11 constituent scenarios via the proposed encoding (left-hand side) and via the random search (right-hand side). The ‘proposed’ CPOG contains shorter conditions ϕ .

We also run the *Proposed encoding* by constraining $\left\lceil \frac{|S|}{2} \right\rceil$ scenarios of every processor specification randomly, using $\{0, 1, ?, X\}$. The resulting decoders always satisfy the composition constraints given, and have an overhead of $\simeq 12.4\%$, on average, in comparison to optimal implementations.

Finally, we used the behavioural synthesis approaches based on *Finite State Machines* (FSM) and *Signal Transition Graphs* (STG), in order to show that the proposed

methodology shows better results in comparison to established techniques in the field. The approach based on synchronous FSM and Design Compiler (known as *dc_shell* in the Synopsys tool-chain) is always able to synthesise controllers from the given specifications with the usage of the *sequential encoding*. Synthesised implementations show an average area overhead of $\simeq 56\%$ in comparison to the proposed unconstrained approach. The processing runtime is comparable.

On the other hand, the methodology based on STG is never able to synthesise implementations from the given specifications with the *sequential encoding*. The results shown on the table are derived with the *one-hot encoding*, which simplifies the specifications by replacing the *go* transitions and their dependencies with the codes, see [9]. However, even after this simplification, the methodology is not often successful. In most cases, Petrify returns the error “*support too big for minimisation*” (see missing results — on the left-hand side of the STG column), and MPSat does not find a solution within the given time limit (see TO entries on the right-hand side). MPSat is partially successful with the ARM Cortex M0+, whose scenarios include fewer datapath modules (6 compared to the 17 modules of the Intel 8051) and which does not include conditional scenarios (as the TI MSP430). On average, the methodology based on STG has an area overhead of $\simeq 51\%$ in comparison to the proposed unconstrained approach, and a much higher synthesis runtime.

D. Software output logs

The third set of benchmarks includes scenario specifications that describe a set of different software output logs [36].

TABLE IV: Three configurations of the proposed algorithm are compared with trivial CPOG composition techniques on 11 software output logs divided in (S)mall, (M)edium and (L)arge sizes. Bold results are the smallest controllers for each model. *Units of measure:* Area ($|B|$) = [μm^2] (number of bits), Runtime (RT) = [s].

| Model | $ S $ | Sequential | | Random search | | (a) Proposed #e = 1 | | (b) Proposed #e = 10 | | (c) Proposed #e = 1, SA $\times 10$ | |
|-------------------|-------|-------------------|---------|----------------|---------|------------------------|---------|-------------------------|---------|--|---------|
| | | Area ($ B $) | Runtime | Area ($ B $) | Runtime | Area ($ B $) | Runtime | Area ($ B $) | Runtime | Area ($ B $) | Runtime |
| BigLog1 | 16 | 503 (4) | 1 | 530 (4) | 1 | 495 (4) | 1 | 389 (4) | 2 | 447 (4) | 1 |
| (S) Purchasetopay | 20 | 581 (5) | 1 | 713 (5) | 1 | 502 (5) | 1 | 434 (5) | 3 | 546 (5) | 1 |
| BigLog2 | 26 | 953 (5) | 1 | 906 (5) | 1 | 699 (5) | 1 | 565 (5) | 3 | 568 (5) | 1 |
| Log2 | 32 | 1163 (5) | 1 | 1271 (5) | 1 | 969 (5) | 1 | 825 (5) | 3 | 815 (5) | 1 |
| Incidenttelco | 77 | 3531 (7) | 1 | 3490 (7) | 1 | 2953 (7) | 1 | 2927 (7) | 6 | 2613 (7) | 1 |
| (M) Svn_log | 92 | 3215 (7) | 1 | 3274 (7) | 1 | 2928 (7) | 1 | 2769 (7) | 5 | 2363 (7) | 2 |
| Telecom | 122 | 4417 (7) | 1 | 4644 (7) | 1 | 4123 (7) | 1 | 4237 (7) | 7 | 3938 (7) | 2 |
| Colibrilog | 167 | 6870 (8) | 2 | 7777 (8) | 2 | 7212 (8) | 3 | 6789 (8) | 11 | 6852 (8) | 6 |
| Caise2014 | 401 | 37314 (9) | 11 | 38339 (9) | 11 | 37234 (9) | 14 | 37180 (9) | 115 | 35616 (9) | 32 |
| (L) Log1-filtered | 402 | 13910 (9) | 3 | 20215 (9) | 4 | 18212 (9) | 6 | 18004 (9) | 46 | 14343 (9) | 31 |
| Documentflow | 651 | 21131 (10) | 8 | 25222 (10) | 8 | 24700 (10) | 12 | 24623 (10) | 112 | 22646 (10) | 57 |

They come from the process mining community: artificial logs derived from the simulation of a process model (BigLog1, Log2, Caise2014), and real-life traces in different other contexts (purchasetopay, incidenttelco, svn_log, telecom, documentflow).

Due to the size of these benchmarks (from 16 to 651 scenarios), we compare the *Proposed encoding* to the *Sequential encoding* and *Random search*, as the other CPOG algorithms always exceed the time limit. The proposed encoding is applied with three configurations: (a) with #e set to 1, (b) with #e set to 10, (c) and with #e = 1 and the Simulated Annealing parameters modified in such a way to allow $\times 10$ more iterations for the optimisation (SA $\times 10$).

Evaluation: On average, the area of the controllers found by the proposed encoding in configurations 1 and (2) are 4.7% (9.8%) more efficient, in terms of area, than sequential controllers, and 12.9% (18%) more efficient than random controllers. In turn, the results produced by the proposed encoding in configuration 3 are 13.2% and 21.7% more efficient, on average, than the sequential and random implementations, respectively.

On average, the Sequential encoding produces $\simeq 8.56\%$ smaller controllers in comparison to the Random search algorithm. Such a good result is due to a certain degree of similarity between pairs of subsequent scenarios, which are encoded naturally by pairs of subsequent and similar codes by the Sequential algorithm.

In Table IV, the benchmarks are divided in three sets of different sizes, from the bottom to the top: (S)mall ($10 < |S| < 30$), (M)edium ($30 < |S| < 400$) and (L)arge ($400 < |S| < 652$). See below consideration:

Small set, configuration 2 of the proposed encoding finds the best results, as the higher number of encodings inspected (#e = 10) provides a higher chance to produce a good result. The increased number of SA iterations of configuration 3 is not justified in this set due to the small $|S|$.

Medium set, configuration 3 finds the best results, as the higher $|S|$ justifies a longer optimisation time provided to the Simulated Annealing optimisation.

TABLE V: Features of the CPOGs compositional algorithms. **Max $|S|$:** maximum number of scenarios supported. **CPOG-scenarios:** support of scenarios in the form of CPOG. **Constraints:** support of composition constraints.

| | Exhaust. | SAT-based | Single-lit. | Proposed |
|-----------------------------|----------|----------------|----------------|--------------|
| Max S | 8 | $\simeq 10/15$ | $\simeq 10/15$ | $\simeq 650$ |
| CPOG-scenarios | ✓ | | ✓ | ✓ |
| Constraints | | | | ✓ |

Large set, configuration 3 finds smaller controllers in comparison to configurations 1 and 2. However, these benchmarks highlight the heuristic (inaccurate) component of the proposed approach, which may find worse controllers in comparison to trivial algorithms. For this set, a higher number of SA iterations would be justified for obtaining good results.

This set of benchmarks shows that the proposed approach can handle specifications of hundreds of scenarios. Also, it can be tuned as much as needed by modifying the time for the SA optimisation.

VII. DISCUSSION AND FUTURE RESEARCH

This paper presented a novel approach to scenario composition for the design methodology based on the Conditional Partial Order Graphs. The presented open-source SCENCO tool, embedded in the EDA toolsuite WORKCRAFT, implements this methodology. The algorithm and tool are evaluated on a set of benchmarks and compared to the state-of-the-art composition algorithms for CPOG, and to the behavioural synthesis techniques based on FSM and STG.

Table V summarises the comparison of all CPOG composition techniques, relying on the experimental results shown and evaluated in Section VI. The proposed algorithm, unlike previously published techniques, handles hundreds of scenarios with a good area/synthesis runtime trade-off, and supports composition constraints. It also supports conditional scenarios for modelling behaviours that contain dynamic branching. Also, the experimental results highlight that the

CPOG methodology produces more efficient implementations (in terms of area) than the approach based on the FSM and STG. The latter can be applied only to relatively compact models.

To further improve the CPOG methodology, a number of recommendations for future research are given. (1) Parallel implementation of the presented algorithm, which is important to further improve the efficiency of behavioural composition by exploring more solutions at no extra runtime cost. (2) Support for x-aware scenario encoding (with x being latency, power, energy, and other characteristics), which is important for making the methodology attractive to many practical domains.

ACKNOWLEDGMENT

We would like to thank all members of the μ Systems Research Group at Newcastle University for supporting us and this research. In particular, we thank Alex Yakovlev, Danil Sokolov, Maxim Rykunov, Jonathan Beaumont and Ghaith Tarawneh. This research was supported by the Royal Society Research Grant “Computation Alive: Design of a Processor with Survival Instincts”.

REFERENCES

- [1] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 37–47. ACM, 2010.
- [2] D. Gajski, AC-H Wu, V. Chaiyakul, S. Mori, T. Nukiyama, and P. Bricaud. Essential issues for ip reuse. In *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pages 37–42. IEEE, 2000.
- [3] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009.
- [4] A. Reid. Trustworthy specifications of ARM v8-a and v8-m system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168, Oct 2016.
- [5] A. Fox and M.O. Myreen. *A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture*, pages 243–258. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [6] A. Mokhov and A. Yakovlev. Conditional partial order graphs: Model, synthesis, and application. *IEEE Transactions on Computers*, 59(11):1480–1493, Nov 2010.
- [7] A. Mokhov, A. Iliasov, D. Sokolov, M. Rykunov, A. Yakovlev, and A. Romanovsky. Synthesis of Processor Instruction Sets from High-Level ISA Specifications. *IEEE Transactions on Computers*, 63(6):1552–1566, June 2014.
- [8] A. Mokhov, A. Alekseyev, and A. Yakovlev. Encoding of processor instruction sets with explicit concurrency control. *IET Computers Digital Techniques*, 5(6):427–439, November 2011.
- [9] SCENCO code. GitHub repository: github.com/tuura/scenco.
- [10] WORKCRAFT. GitHub repository: github.com/workcraft/workcraft, WORKCRAFT website: www.workcraft.org.
- [11] A. de Gennaro, P. Stankaitis, and A. Mokhov. A heuristic algorithm for deriving compact models of processor instruction sets. In *2015 15th International Conference on Application of Concurrency to System Design*, pages 100–109, June 2015.
- [12] G. Birkhoff. *Lattice Theory*. Number v. 25, pt. 2 in American Mathematical Society colloquium publications. American Mathematical Society, 1940.
- [13] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-Formal. In *International Conference on Computer Aided Verification*, pages 42–58. Springer, 2016.
- [14] Kathryn E Gray, Peter Sewell, Christopher Pulte, Shaked Flur, and Robert Norton-Wright. The Sail instruction-set semantics specification language. 2017.
- [15] Georgy Lukyanov and Andrey Mokhov. Concurrency Oracles for Free. In *Proceedings of the Algorithms and Theories for the Analysis of Event Data 2018 Workshop*, 2018 (in print).
- [16] A. Mokhov. *Conditional Partial Order Graphs*. PhD thesis, Newcastle University, 2009.
- [17] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis for Asynchronous Controllers and Interfaces*. Springer Publishing Company, Incorporated, 2013.
- [18] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.
- [19] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers, 1996.
- [20] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state coding conflicts in stg unfoldings using sat. In *Third International Conference on Application of Concurrency to System Design, 2003. Proceedings.*, pages 51–60, June 2003.
- [21] Pran Kurup and Taher Abbasi. *Logic Synthesis Using Synopsys*. Springer Publishing Company, Incorporated, 2nd edition, 2011.
- [22] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [23] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52 – 89, 2012.
- [24] Niklas Eén and Niklas Sörensson. *An Extensible SAT-solver*, pages 502–518. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
- [25] R.W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950.
- [26] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.
- [27] P.C. McGeer, J.V. Sanghavi, R.K. Brayton, and A.L. Sangiovanni-Vicentelli. Espresso-signature: a new exact minimizer for logic functions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(4):432–440, Dec 1993.
- [28] Berkeley Logic Synthesis and Verification Group. ABC, a system for sequential synthesis and verification.
- [29] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vicentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.
- [30] SCENCO documentation. workcraft.org/help/encoding_plugin.
- [31] A. Mokhov, M. Rykunov, D. Sokolov, and A. Yakovlev. Design of processors with reconfigurable microarchitecture. *Journal of Low Power Electronics and Applications*, 4(1):26–43, 2014.
- [32] J. Sparso and S. Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Springer Publishing Company, Incorporated, 1st edition, 2010.
- [33] D. Sokolov, V. Dubikhin, V. Khomenko, D. Lloyd, A. Mokhov, and A. Yakovlev. Benefits of asynchronous control for analog electronics: Multiphase buck case study. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1751–1756, March 2017.
- [34] D. Sokolov, A. de Gennaro, and A. Mokhov. Reconfigurable asynchronous pipelines: from formal models to silicon. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2018*, March 2018.
- [35] M. Rykunov. *Design of Asynchronous Microprocessor for Power Proportionality*. PhD thesis, Newcastle University, 2013.
- [36] A. Mokhov, J. Carmona, and J. Beaumont. *Mining Conditional Partial Order Graphs from Event Logs*, pages 114–136. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.