IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 15, 2021, accepted May 8, 2021, date of publication May 19, 2021, date of current version June 2, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3081742

Handling State Space Explosion in Component-Based Software Verification: A Review

FARANAK NEJATI, ABDUL AZIM ABD GHANI, NG KENG YAP, (Member, IEEE), AND AZMI BIN JAFAAR
FSKTM, Department of Software Engineering and Information Systems, Universiti Putra Malaysia, Seri Kembangan 43400, Malaysia
Corresponding author: Abdul Azim Abd Ghani (azim@upm.edu.my)

This work was supported by the Ministry of Higher Education Malaysia and Universiti Putra Malaysia under Grant FRGS/1/2015/ICT01/UPM/02/6.

ABSTRACT Component-based software development (CBSD) is an alternative approach to constructing software systems that offers numerous benefits, particularly in decreasing the complexity of system design. However, deploying components into a system is a challenging and error-prone task. Model-checking is one of the reliable methods to systematically analyze the correctness of a system. Its brute-force checking of the system's state space assists to significantly expand the level of confidence in the system. Nevertheless, model-checking is limited by a critical problem called state space explosion (SSE). To benefit from model-checking, an appropriate method is required to reduce SSE. In the past two decades, a great number of SSE reduction methods have been proposed containing many similarities, dissimilarities, and unclear concepts in some cases. This research, firstly, plans to present a review of SSE handling methods and classify them based on their similarities, principle, and characteristics. Second, it investigates the methods for handling SSE problem in the verification process of CBSD and provides insight into the potential limitations, underlining the key challenges for future research efforts.

INDEX TERMS Component-based software development, verification of software components, model-checking, state space explosion.

I. INTRODUCTION

Component-based software development (CBSD) is a vital emerging topic in software engineering [1], [2]. CBSD is an alternative approach of constructing systems from prebuilt software units (components) which offers numerous benefits, particularly in decreasing the complexity of the system design. However, deploying components into a system is a challenging and error-prone task. Errors may lead to destructive results. A single error can lead to an overall system crash, as in the error that crashed the Arian-5 rocket. There was a small application in Arian-5 for the inertial reference system that was trying to assign a 64-bit floating-point number into a variable with 16-bit space [3]. This small mistake led to the catastrophic explosion of the aircraft. There are various other safety-critical systems similar to the Arian-5 that could have disastrous outcomes if such errors occur; like in nuclear power stations, avionic software, aircraft flight control, and traffic control [4].

Model-checking is one of the renowned approaches for verifying component-based software systems [5]. It is a brute-force verification method that is able to automatically and systematically analyze the specification and state space (SS) of a given system to demonstrate if its properties are satisfied completely or otherwise. This approach has been proposed independently by Clarke *et al.* [5] and Queille and Sifakis [6]. The brute-force check of SS in model-checking significantly expands the level of confidence in the system.

However, model-checking is limited by state space explosion (SSE). SSE occurs when a system's state space increases exponentially with the number of its components, thus rapidly surpasses the memory capacity of the computer. Subsequently, the amount of SS that can be checked by a model checker will be restricted. However, the promising advantages offered by model-checking have nevertheless encouraged the research community to tackle the SSE

The associate editor coordinating the review of this manuscript and approving it for publication was Antonio Piccinno.

obstacle and has spearheaded the major direction of model-checking research [7], [8]. As a result, a massive collection of methods to alleviate SSE problems in all domains of software development has been presented. However, it should be determined which of these methods and algorithms would be sufficient in CBSD for supporting component-wise software development and its justification in order to ensure its compatibility in component-wise software systems.

In that effort, we first summarized in subsequent sections all the proposed SSE reduction methods present in literature and provided a classification based on their principles and characteristics. The classification offers explanations pertaining to the key features and challenges in the literature that would aid in understanding model-checking and SSE reduction methods. The classification can then be utilized in the development of a new method that is suitable for a particular application domain.

Additionally, it can be deduced further to determine from the classification all component-wise software methods that are utilized the most in CBSD to rectify its suitability for CBSD along with a discussion pertaining to the key feature and challenges that are mentioned in the literature.

To this end, we set up the following objectives: 1- reviewing and briefly describing the methods presented in the literature to address SSE problem in model checking; 2- classifying them based on their principles and characteristics; 3- deducing the key features and challenges of the methods mentioned in the literature; 4- identifying and discussing the methods for tackling SSE problem used in CBSD in order to both analyze gaps and identify suitable methods for SSE reduction in CBSD.

To complete these steps sufficiently, six research questions (RQs) have been formulated which are defined in Section II. Answering these questions aids in enhancing comprehension, determining the suitability of SSE reduction methods, and underlying the key features and challenges for future researches.

The presented study shares similarities with surveys presented in [9]–[11] for dealing with SSE problem, however, this paper collects the wide range of SSE reduction methods, offers explanations for each method, their success factor, and identifies challenges. Additionally, a discussion of SSE problems in component-based systems is also covered. Other review papers alike [12], are based on the tool-sets which is beyond the scope of this paper.

The remainder of the paper is arranged as the following: in Section II, the research methodology is presented. Section III provides an explanation of the basic concepts about SSE used in this paper. Section IV defines, classifies, and explains the different methods for alleviating SSE problem. In addition, it also contains a tabled summary of the key factors and limitations of these methods. Section V contains a discussion about tackling SSE problem in CBSD and identifies the key challenges for future research. Section VI discusses and concludes the results.

II. RESEARCH METHOD

In this section, the conducted research method is described. In order to collect studies; we carried out the following steps: *(i)* formulating research questions, *(ii)* identifying keywords, databases and search strategy, *(iii)* obtaining inclusion and exclusion criteria to collect and analyse literatures, *(iv)* quality assessment and selected studies, and *(v)* information extraction.

A. RESEARCH QUESTIONS

The objective of this work is to classify and briefly describe SSE reduction methods and the underlying key features and challenges of the methods in both general and CBSD systems. The set of corresponding research questions are listed in Table 1. Answering these RQs establishes an effective overview of the most current SSE reduction method and fulfills the objectives of this research.

B. KEYWORDS, DATABASES, AND SEARCH STRATEGY

The keywords and search query for the RQ (1, 2, 3) were (‘‘model-checking’’ AND ‘‘State space explosion problem’’) or (‘‘model-checking’’ AND ‘‘the name of each mitigation methods for SSE, for example, assume-guarantee’’). For RQ (4, 5, 6), we added ‘‘component-based system’’ to the above search strings.

To shape the keywords and search query, this work relies on ‘‘Model Checking’’ by Clarke *et al.* [5], and ‘‘Specification and verification of concurrent systems in CESAR’’ by Queille and Sifakis *et al.* [6], as a basis together with other studies published by those studies like [13], [14]. One of the reasons that these books and studies have been selected is a credit to the authors as pioneering scientists in the model-checking.

The search queries have been executed in widely known electronic database/library resources such as ACM digital library, IEEExplore, Science Direct, Web of Science, Springer link, Google scholar, Citeceer.

C. INCLUSION AND EXCLUSION CRITERIA

In order to select the most important papers within the scope of this paper, a set of inclusion and exclusion criteria have been established. It is represented in Table 2.

D. QUALITY ASSESSMENT AND SELECTED STUDIES

After each iteration of the query, a preliminary review is carried out based on the inclusion and exclusion criteria to obtain an appropriate literature collection. Although specifying keywords and search queries, it has been observed that some results returned by the search engine are pertaining to model-checking concerning other domains or perspectives such as hardware verification. Some papers also use model-checking for specific programming languages or verify specific software. For example, in [15] a compositional approach has been presented and utilized for automatic verification of C programs. These papers are excluded as well.

TABLE 1. Research questions.

NO	Research questions	Motivation
RQ1	Which methods have been proposed in the literature for SSE problem?	To understand the current state-of-the-art methods for mitigating SSE problem.
RQ2	How the methods mitigate SSE problem?	To enhance understanding of the theories and concepts, as well as recognizing the different between SSE reduction methods easily. It also assists with classifying the methods.
RQ3	What are the key features and challenges have been obtained in the literature for SSE reduction methods?	To identify the success factors and potential challenges that could be encountered in using particular SSE reduction methods.
RQ4	Which SSE reduction methods are more frequently utilized in the literature for CBSD?	To determine the kinds of SSE reduction methods that can be apply to verify CBSD.
RQ5	How SSE has been tackled in CBSD?	To enhance comprehension of the theories, concepts, success factors, and challenges.
RQ6	What are the key features and challenges have been obtained in the literature for SSE reduction methods in CBSD?	To identify how challenges could fail or limit the verification process of CBSD.

TABLE 2. Inclusion and exclusion criteria.

General inclusion criteria	1) Research papers, white papers, conferences, technical reports, doctorate dissertation, books, hands books. 2) Studies the researches of well-known authors in model-checking and formal verification.
Specific inclusion criteria for RQs	1) The studies analyzes/addresses SSE in normal model-checking (RQ 1). 2) The studies provide discussion about mitigation methods of SSE, its key features and challenges (RQ 2 and 3). 3) The studies analyze/discuss/present verification in CBSD (RQ 4). 4) The studies provide discussion about mitigation methods of SSE in CBSD verification, the key features and challenges (RQ 5 and 6). 5) The studies or tool sets that can be used to make the description more clear such as by giving an example, or illustrating other aspects or domains of the presented method.
Exclusion criteria	1) Duplicate report of the same study. 2) Studies to utilize the already proposed methods rather than propose a new method to tackle SSE and do not fulfill the inclusion criterion No. 5. 3) Studies to present tool-sets based on the already proposed methods and do not fulfill the inclusion criterion No. 5. 4) Studies that do not introduced a method for SSE reduction. 5) Studies that utilize a SSE reduction method in a domain or programming language rather than introduce a new method. 6) Studies that are more focus on deductive verification such as theorem proving. Refer to II-D for more details. 7) Studies that do not contain the defined keywords.

Some of the studies found were based on deductive methods like theorem proving. However, model-checking is an algorithmic method for deciding whether a hardware or software design meets a formal specification [16]. Due to this, studies containing methods such as theorem proving are excluded as well. For more details on difference between deductive methods and model checking, refer to [17]. The emphasis of this paper is more on having insight into the SSE reduction methods and determine the methods that are suitable for verifying component-based systems.

E. INFORMATION EXTRACTION

We extract the following items from each selected paper:

1) SSE reduction methods: the methods to mitigate SSE problem.

2) Method's description: a brief description of SSE reduction methods.
3) SSE reduction methods in CBSD: the frequently used SSE reduction methods in CBSD. With the focus to show which SSE reduction method has been used frequently in (all domain of) CBSD.
4) Key features and potential challenges: the pros and cons of the methods that have been obtained in the literature.
5) Examples: Utilizing study examples/tools and illustrations of other aspects or domains of the methods to aid in the conceptual comprehension.

III. MAIN CONCEPTS

Formal methods have great potential to verify and ensure the system's correctness as early as possible. It removes ambiguity in the system specification and provides preciseness. One of the well-known formal methods is *Model-checking*. It is possible to describe model-checking as a tuple:

$$\mathbb{MODELCHECKING} =< \mathbb{M}, \mathbb{S} > \tag{1}$$

where $\mathbb{M}$ is a system model with m states and $\mathbb{S}$ is the formal specification as shown in Figure 1. Let m be a state of the system model, $m \in \mathbb{M}$, then model-checking searches all states m_i for $1 \leq i \leq n$ in $\mathbb{M}$ and returns "Yes" if $\forall m$ satisfy $\mathbb{S}$, $(\forall m_i \in \mathbb{M}) \models \mathbb{S}$ for $1 \leq i \leq n$. Otherwise, model-checking produces counterexample(s). Counterexample(s) is a declaration that defeats the specified properties on a given system by presenting it in at least one path in the SS.

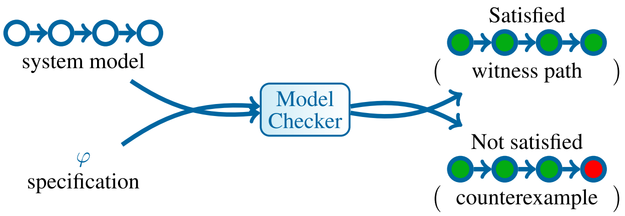

FIGURE 1. Model checking.

System model $\mathbb{M}$ is a conceptual model to represent and describe a system. In model-checking, a system model is originally represented by a *Kripke* structure, but can be represented as a state chart [18], Petri net [19], or other possible graph-like visualization of system's SS. Using graph-like representation by individual states is one of the main representation paradigms in model-checking called *explicit-state model-checking*. Figure 2 shows an explicit-state system model based on a *Kripke* structure. The nodes in Figure 2(a) (named by X, Y, Z) are the states of a system (or a process) modelled as nodes in the *Kripke* structure. Figure 2(b) shows the SS of the modelled system in Figure 2(a).

Another representation paradigm is the *implicit model-checking*. In this model, states checking are not individually represented, but a quantified propositional logic formula is used to represent the state space graph.

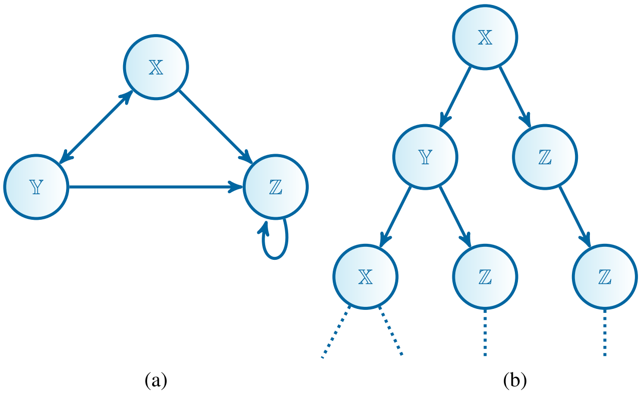

FIGURE 2. A kripke structure and its computation tree [5].

The formal specification in model-checking is represented by propositional temporal logic which is a kind of logic to specify and reason over the ongoing properties of the system being modelled in terms of time. There are two types of operations that temporal logic supports: The first are logical operators such as ¬, ∨, ∧, and the second refers to modalities like **F**inally, **U**ntil, **G**lobally. The set of operators which expresses properties in only a single future position for every point in running time is called *linear temporal logic (LTL)*.

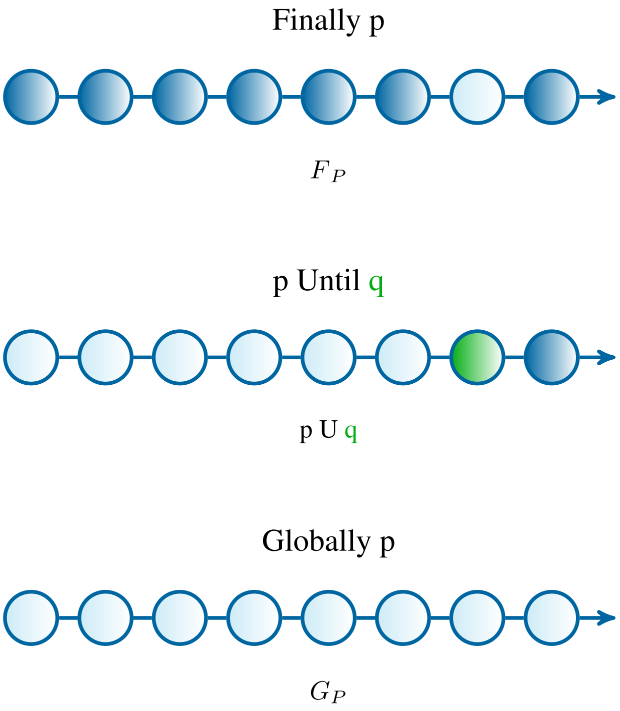

FIGURE 3. Example of LTL operators [14].

In Figure 3, three sequences of events of a system (or a process) is shown with two properties, p and q. LTL checks only one sequence of event(s) in a single run and does not switch to another run while checking. For example, in the first sequence of events in Figure 3, F_p (finally properties p happens) will be checked. The light blue state, p, will

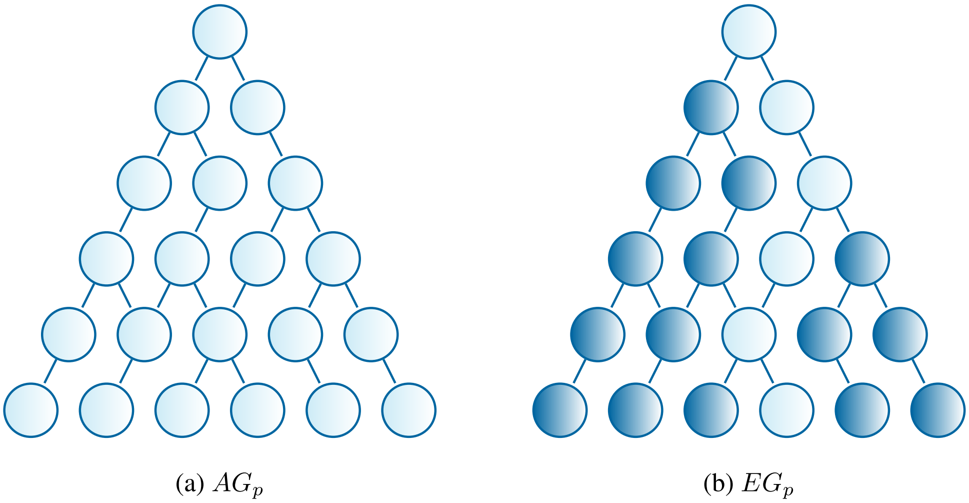
(a) AG_p (b) EG_p

FIGURE 4. Example of CTL operators [14].

finally happen after the other states (indicated by dark blue) is checked. For the second sequence of events in Figure 3, $p\ U\ q$ (p can continue to happen until q happens) will be checked. The light blue and green states are p and q accordingly. For the third sequence of events, Gp (globally p may happen in the future) among all of the states in the sequence, p may happen. Due to this, all the states are in light blue.

On the other hand, there is another type of temporal logic called *computational tree logic (CTL)* that is able to check all possible paths in a single run. In Figure 4(a), the light blue nodes are all possible paths, which CTL can switch between any of them during one single run. It checks by operator AG_p (in all paths, the property p may happen). CTL operators can support one sequence of events in one run as well. For example, in Figure 4(b), the light blue path is one sequence of events. This sequence will be checked by operator EG_p (eventually in one of the paths, the property p may happen) to indicate whether p eventually happens or not.

System property is defined by temporal logic. Some common system properties are *reachability properties* (some particular position in a system model that can be met), *Safety properties* (under particular circumstances, an event will not happen in any way, like without the key a car won't start.), *liveness* (under certain circumstances, some event will eventually happen, like if we press the button of an elevator, it is bound to arrive ultimately.), *fairness* (under certain circumstances an event will or will not happen infinitely often, like the gate will be raised infinitely often.) [5].

State space explosion is the most critical problem restricting model-checking. To define SSE, let a given system contains n processes and each process has m states (same states). The size of SS of these n processes might be m^n. Then, the amount of state space of a given system may increase exponentially with the size of its states (processes) and consequently exceed the memory capacity of the system.

IV. RESULTS ON SSE REDUCTION METHODS

This section elaborates the results of SSE reduction methods that have been found in the research literature. The results are represented as a classification of SSE reduction methods. The classification has five dimensions summarized as the following:

1) *Memory handling.* Identifies the methods that are directly engaged in memory expansion and management.
2) *Heuristics and probabilistic reasoning.* Identifies the methods that are able to find an approximation of the exact solution. They are the fastest way to find a close solution when the exact answer cannot be computed.
3) *Scaling down the state space.* Identifies the methods that try to reduce the size of states to be stored in the memory. The reduction can be based on compression, symmetry, and similarity omission, using binary decision diagram (BDD), or hash table.
4) *Bottom-up approach.* Identifies the methods that start verification as early as the whole state space is constructed.
5) *Divide-and-conquer approach.* Identifies the methods which decompose the state space into small parts and address each small part separately.

SSE problem is a bottleneck in model-checking. The amount of a system's SS (even a finite system) strongly depends on its components and is prone to increase in size exponentially. Consequently, it easily exceeds the computer's memory capacity and limits the size that can be verified by a model checker. Therefore, memory becomes the main concern for SSE reduction methods. Memory concern can be addressed from several aspects, for example, expanding memory capacity, reducing the states that need to be stored in memory, released memory from the redundant and repeated states, etc. Generally, a method cannot cover all the aspects, some methods find and omit redundancies, while others may focus on memory expansion. Therefore, this work simply

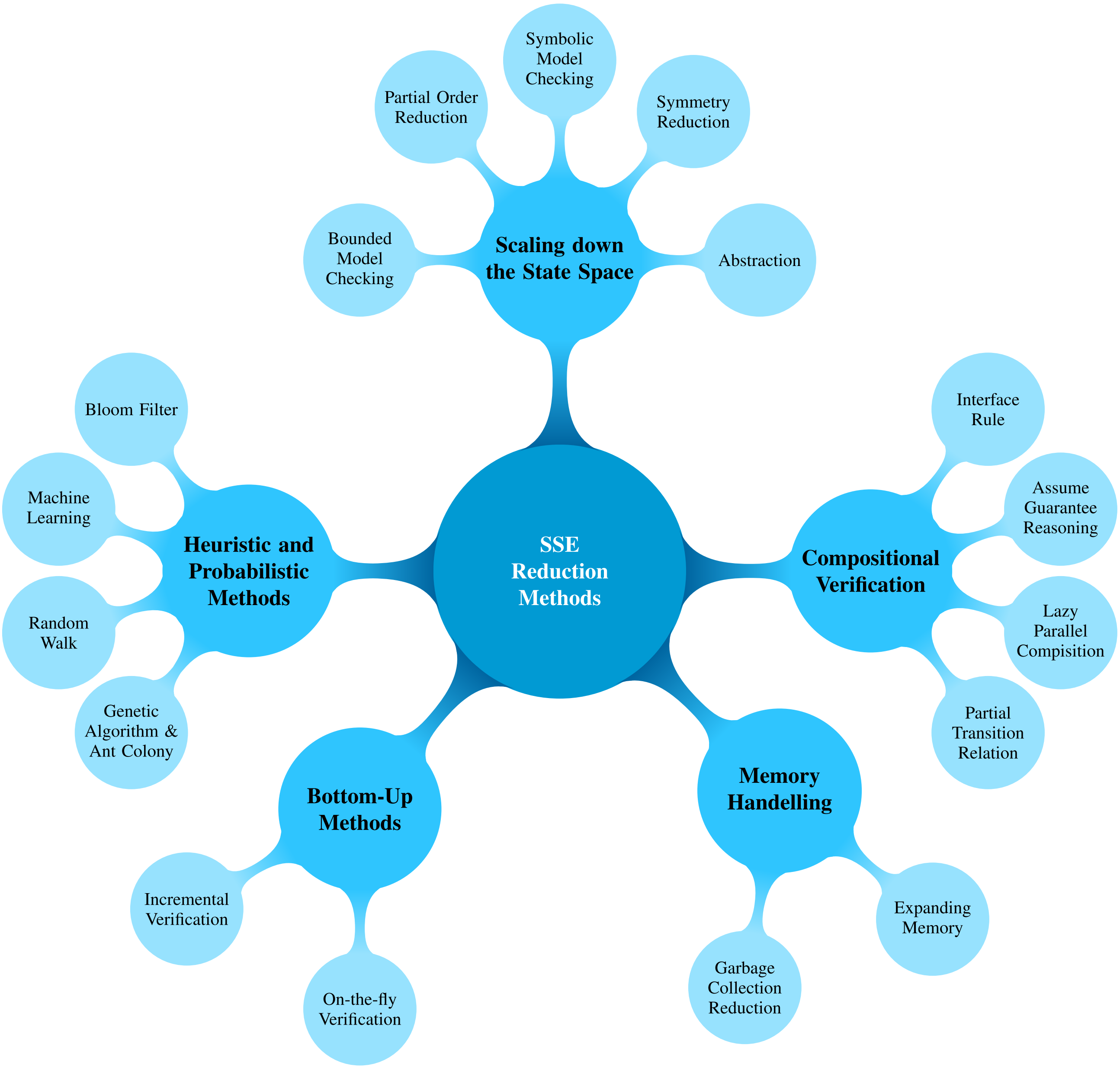

FIGURE 5. An overview of SSE reduction methods.

classifies these methods according to the aspects that each method can cover.

A discussion about the classification is provided in detail in the first subsection and an illustration of it is represented in Figure 5. Each class is divided into multiple sub-classes which is in accordance with the current methods for tackling SSE problem. The RQs (1, 2) are answered in this section. The second part of this section pertains to RQ 3. The characteristics, key features, and challenges of SSE reduction methods have been summarized in tables.

A. CLASSIFICATION

The classified SSE reduction methods are described below based on evaluating the selected studies:

1) MEMORY HANDLING

As previously described, memory is the main concern of SSE reduction methods that can be addressed from several aspects. Some of them are specifically involved in memory and are contradictory to the other methods like compressing the state space. These kinds of methods are based on the following principles: 1- proper memory management to increase performance. Memory management is a process of controlling and incorporating programs by using a sufficient methodology to fragment, allocate, monitor, and release memory. 2- increase memory capacity to provide more space for data storage. It can be achieved by expanding the external memory. The aforementioned two principles can be achieved through the following:

a: EXPANDING EXTERNAL MEMORY

The idea of using an external memory with a proper algorithm when the RAM cannot handle all data, might be one of the solutions to overcome SSE problem. External memory is able to provide a much larger space. Currently, the capacity of magnetic disks is increasing enormously at a relative cost. This fact motivates researchers to utilize external memories in model-checking. Due to the fact that external memory cannot be accessed rapidly like internal memory, providing an efficient external memory algorithm is the main concern in using this method. The algorithm must organize disk access carefully and precisely. The efficiency of the algorithm is determined via the amount of I/Os. In other words, between the amount of I/Os and time efficiency, there is a relationship in the sense that the time efficiency will be improved if the I/O actions are reduced.

Lamborn and Hansen [20] proposed layered duplicate detection to improve duplicate elimination in external memory model-checking. This approach determines which states, while searching in the state space, should be stored in the RAM, and which of them should be stored on disk. As a result, it increases the efficiency of the run time and decreases the amount of disk storage.

Wu *et al.* [21] proposed an I/O (input/output) efficient methodology to provide a model checker based on extending memory. Their methodology is generally based on nested depth-first (NDF). By combining the following three methods the authors have achieved a significant improvement in time efficiency of I/Os. The first method is sorting a hash table in linear time. In this method, the already visited states sorted in a hash table will be merged into a hash table which is saved and sorted into the external memory. The second method is detecting duplicates in a cache. Finally, the third method refers to the managing of the dynamic path. This methodology gives performance guarantees for I/O efficiency.

b: GARBAGE COLLECTION REDUCTION

Garbage collection reduction is a memory management policy that is inspired by utilizing garbage collection in real-life software systems to improve model-checking methods. It deletes information about already visited-states and reclaims allocated memory while model-checking is performed. On the contrary, when garbage collection and memory reclaiming for idle memory is not utilized, the state space may grow and limit the verification. Garbage collection has some advantages, for example, it does not suffer from inefficient memory fragmentation and complex pointer analysis.

One of the classic collection algorithms is by Mark and Sweep. In this algorithm, a state can be marked as garbage when no more transition to it is available [22]. In other words, it is based on reachability [23] and uses graph-search algorithms like depth-first search to indicate any state that must be marked as garbage.

The other collection algorithm is reference counting collection [24]. It discovers the garbage directly by monitoring and counting the pointers that point to each state. Disabling to reclaim the cycles of garbage is the major difficulty of this algorithm [23]. Additional algorithm for garbage collections is based on finding usability [25]. The most widely used garbage collections are Java based programs [26].

2) HEURISTIC AND PROBABILISTIC REASONING

In many problems that do not have an exact solution, the hope is to have at least an approximate answer. In this case, probabilistic reasoning can handle the situation. It is able to find approximate solution faster than other methods. The solution may not be optimum, however, is still valuable because it helps us in cases that the exact solution could not be achieved. Utilizing this kind of method is a possible way to reduce the model-checking effort. The rest of this section introduces a few SSE reduction methods based on probabilistic reasoning.

a: GENETIC ALGORITHMS

A genetic algorithm (GA) in computer science is a metaheuristic optimizer that is based on population (a set of chromosomes) and encouraged by biological evolution. GA is a subset of a larger category called evolutionary algorithms (EA) and has been used in model-checking [27]. In some cases, an existing exact method (the methods which try to find an exact solution, not an approximate solution) can fail to detect an exact or complete solution for a given problem or there is no solution with lower complexity to them. Thus, it may be sufficient to find solutions approximately or to provide faster coverage approaching solutions by using heuristics such as GA. In analyzing the use of GA or any EAs in model-checking, one must consider to target finding any solution (any error) not only the optimum solution. In addition, every reachable state of the entire system must also be checked.

P. Godefroid and S. Khurshid investigated the utilization of GA in order to explore very large state space for finding errors such as deadlock and assertions violations [28]. They combined model-checking and GA to guide searching during the verification problem of a concurrent reactive system. When there are more than one enabled transitions in the current state, GA tries to explore them and find the transition which is the most fitted to be selected by using a fitness function.

Yousefian *et al.* [29] explored the use of GA for model-checking of graph transformation systems. In their work, an incomplete SS is created instead of creating an entire SS to detect deadlock. Model checker only checks paths with a low outgoing transition. Another article found in the conducted search [30] presented a mixed way of genetic algorithm and assume-guarantee reasoning (this type of model-checking is introduced later in this paper) to mitigate SSE.

To enhance the results of using EAs in model-checking, other state-of-the-art algorithms like the Imperialist competitive algorithm [31], Grey-wolf optimization algorithm [32], or Raccoon optimization algorithm [33] may help.

b: RANDOM WALK

Random walk defines a path that includes a sequence of random steps to find errors in model-checking [34]. For a certain type of graph like the Markov chain, random walk is able to decide reachability and predict error traces by polynomial algorithms. The complexity of the algorithms is not better than some other methods for alleviating SSE problem, instead, it is worst. However, some advantages exist and are of interest to researchers to use in the verification processes. For instance, its need for memory space is minimal which is an advantage. Secondly, parallel random walk is easy to implement and reduces execution time. Despite the advantages, it does not guarantee the exploration of all global states.

A tool for randomization search in SS has been proposed by Owen *et al.* [35] called LURCH. The researchers compared the tool with other tools and concluded that LURCH cannot be preferable as much as the others that have a complete search feature. Nevertheless, futuristics and random search can be useful to some system models that are massive and cannot be explored completely. A random walk based heuristic algorithms are presented in [36], [37]

c: BLOOM FILTER

The two main schemes for probabilistic verification are hash compaction and bit state hashing, which utilizes the data structure of Bloom filter. Bloom filter is an explicit and probabilistic method for verification activities. It stores compressed values in a hash table rather than storing full state descriptors. During the verification process, the states with a non-zero probability will be deleted. Therefore, some reachable states are never checked during the verification process and may result in false-positive outcomes.

An improved probabilistic method based on this method has been proposed by U. Stern and D. L. Dill in [38]. The researchers reduce the probability of deleting states by using a specific hashing design. The design requires a lower number of probes required in the hash table. Another work of them for the probabilistic method is presented in [39].

Dillinger and Manolios [40] proposed a method based on Bloom filter which is more accurate that shows the Bloom filter can play an important role in model-checking.

d: ANT COLONY

Ant colony method is another probabilistic method to optimize the problems which is mostly used to find the optimum paths through graph-like models. It can be also applied in model-checking and verification. Duarte *et al.* [41] combined model-checking and the ant colony method to solve the traveling salesman problem.

e: MACHINE LEARNING

Machine learning (ML) is another method used to train the data set of a verification process. The data set could be the system modeled through graph-like visualization such as *Kripke* structures, CTL or LTL formulas, and the results obtained from a model-checking tool. Subsequently, ML trains the data set to predict the results. The related articles based on the conducted search in this paper for machine learning in model-checking are [42] and [43].

Another kind of heuristic model-checking method to mitigate SSE found through the search result is the continuous-time Markov chain [44], [45]. A hybrid metaheuristic approach is also presented in [46]. Heuristic model checking could be utilized by other SSE reduction methods to get more optimum results. For example in [47] a heuristics-based incremental model checking at runtime has been introduced.

3) SCALING DOWN THE STATE SPACE

Scaling down the size of SS to be checked by model checkers is another way to alleviate SSE problem. One can begin by representing the SS in another way (implicitly) which consumes less memory, like symbolic representation instead of defining them in the original shape (explicitly), which truly compresses the SS. In the comparison part of this section, in Table 3, the implicit and explicit methods have been indicated. Furthermore, capturing a critical part of the system [48] and ignoring irrelevant or useless variables and information in a system leads to a decrease in the size of the SS. Additionally, discovering duplicate states and avoiding regenerating them is another way of reducing SS.

The following briefly discusses some methods for alleviating SSE problem for the state space scaling down class.

a: SYMBOLIC MODEL CHECKING

This method is utilized to compress the SS of a system by symbolically (implicitly) representing the SS. It considers a large number of states in a single step and represents them as formulas instead of enumerating them one at a time. As a result, representing them in such a way, reduces the size of the SS. It was introduced by Burch *et al.* [49] based on Bryant's binary decision diagram (BDD) for Mu-Calculus. BDD is a data structure used to canonically represent a Boolean formula that is essentially compressed even more than other data structures, and Mu-Calculus falls into a kind of logic called modal logic (a type of logic that is able to express modalities like a possibility and impossibility) which is able to define the properties in terms of graph-like patterns.

The methods have been used successfully for many problems such as to derive efficient decision procedures for CTL, and satisfiability of LTL. For example, a verification tool-set called ITS-tools by Thierry-Mieg [50] has been developed based on symbolic model-checking which supports reachability property and two kinds of temporal logic, CTL and LTL of the concurrent specification. Symbolic model-checking also has been used in a diverse range of systems like distributed control systems [51].

However, the BDD that is a substantial part of symbolic model-checking extremely relies upon the variable's ordering which limits the use of symbolic model-checking. To illustrate it more precisely, we use the following example:

Let two orders of a Boolean functions with 6 variables [52], [53]:

$$1 - (a.b) + (c.d) + (e.f)$$
$$2 - (a.d) + (b.e) + (c.f)$$

These two Boolean functions have the same number of variables, but they are different in order. The constructed BDD for both, as indicated in Figure 6, are not the same because BDD is sensitive to ordering. For the first function, the BDD has fewer nodes while for the second function it has more nodes.

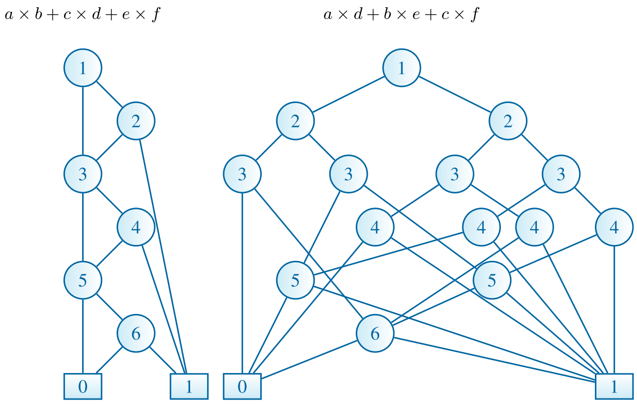
$a \times b + c \times d + e \times f$
$a \times d + b \times e + c \times f$
1
2
3
4
5
6
0
1
1
2
2
3
3
3
3
4
4
4
4
5
5
6
0
1

FIGURE 6. Ordering dependency [52].

Thus, reduced ordered binary decision diagrams (ROBDD) are used to reduce decision graphs and provide a more concise canonical representation for Boolean propositions. An improved variable ordering of BDD was introduced by Prasad *et al.* [54] that is based on graph topology. They demonstrated that using a graph representation of a given Boolean function and computing the shortest path among the variables can improve ROBDD. Further work to improve ROBDD was presented by Sharma and Singh *et al.* [55] to get the most optimum size of ROBDD. However, computing an optimum order for ROBDD generally falls into the NP-Complete problem category proved by B. Bolling *et al.* in [56] and B. Bolling in [57]. A parallel version of symbolic model-checking was also presented to improve the sequential version [58] and [59].

b: BOUNDED MODEL-CHECKING

Bounded model-checking (BMC) has been proposed in [60], [61] to deal with the complexity of model-checking and provide error traces. In this method, the length of the trace to be explored is limited via a fixed amount of states which will be indicated by $2^k \in int$, as illustrated in Figure 7. Then, it checks through it to reveal error states. If an error location could not be reached inside the bound, the amount of k will be increased and the process will be repeated until one error is found. The selected k has to be large enough, otherwise, the method is not able to be completed [62]. However, if k is small enough, it outperforms BDD based model-checking [63]. Furthermore, discovering the k involves minimal hands-on manipulations. On the other hand BDD requires a great deal of hands-on effort to find an optimum ordering. In addition, BMC can handle much more clauses and variables than BDD methods [62].

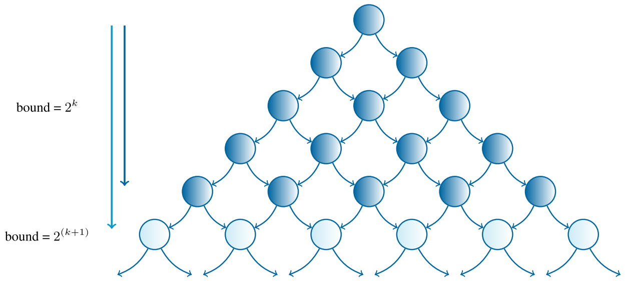

FIGURE 7. Bounded model-checking [64].

Bounded model-checking is the most important industrial application for the Boolean satisfiability (SAT) solver [65]. SAT solver provides a platform for searching and reasoning based on propositional logic that is able to solve complex problems with millions of variables and constraints [66]. SAT testing falls into NP-complete problems [63] and some recent works shows that using BMC in SAT leads to successfully verify security critical systems [67], concurrent systems [68], multi agent systems [69], generating functional tests [70], signal temporal logic [71], and parallel and distributed systems [72].

BMC can also be based on satisfiability modulo theories (SMT) [73]. SMT allows to compress the formula when arrays and vectors are involved [74]. MBC was developed and used in a wide range of communities and domains [68], [72], [75], [76]. An explicit version of BMC has been proposed in the literature in [77].

c: PARTIAL ORDER REDUCTION

This method attempts to cut down the state space of concurrent asynchronous systems [78]. In asynchronous processes, interleaving models of executions must consider all possible orders of events for the sake of preventing the omission of important ones. Some of these ordering results in the same state, as shown in Figure 8. To avoid this, partial order reduction avoids analyzing all sequences and considers only an incomplete set of events. Its methodology does not distinguish between traces that only differ by their orders. For example, in Figure 8 to reach s' from s, it does not matter if α is run first or β. The set of events includes only representatives of enabled transition. Some approaches of partial order reduction is introduced in stubborn sets [79], ample sets [80], persistent sets [81], unfolding methods [82], and sleep sets [83].

Normally, the reduced model of partial order reduction is explicit and is produced by utilizing methods based on modified depth-first search [85] or breadth-first search. It can be combined with other methods such as the on-the-fly model-checking [86] or symbolic model-checking [80]. This method

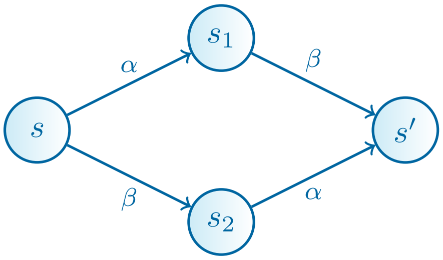

FIGURE 8. Some ordering results in a same state [84].

reduces the memory usage and time requirements. One of the key factors affecting the efficiency of these methods is the number of enabled transitions, which may change a predicate in the verified property [87]. In complex systems, the number of these transitions increase, and fewer reductions can be constructed.

d: ABSTRACTION

Abstraction is a method to handle more state space by abstracting away the entire state space. It is based on the fact that states indicate computations in series of objects and obvious relationships that normally many similar behaviors are between them. The abstraction methods can interpret these objects in another universe of abstraction to avoid exploring all of them. However, the result of its execution must be the same as the original one [88]. In [5], abstraction is defined as the following: let S_i is i_{th} state over the set of all states $S_1 \dots S_n$. Abstraction will give a surjection $h = (h_1, \dots, h_n)$ that groups and maps each state to corresponding abstract states. In [89], E. M. Clarke *et al.* indicated that the abstraction can be done based on the following aspects: an equivalence modulo of an integer to address mathematical operations, symbolic abstraction, and single bit abstraction to address bit-by-bit logical operations.

As an example for the above kinds of abstraction, consider the following arithmetic modulo [89]:

$$(x\ mod\ i) + (y\ mod\ i)modi \equiv x + y\ (mod\ i)$$
$$(x\ mod\ i) - (y\ mod\ i)modi \equiv x - y\ (mod\ i)$$
$$(x\ mod\ i)(y\ mod\ i)\ mod\ i \equiv x\ y\ (mod\ i)$$

To abstract the above modulo and determine the value of modulo i, we can use the values of modulo i from the sub-expressions.

Another type of abstraction strategy based on storage reduction of states has been studied by Holzman *et al.* [90] that intends to minimize the size of the used memory during the construction of the state space. Additionally, another algorithm called *cone of influence* is considered as an abstract method that removes all variables from the system model. The variables are idle or do not have any influence on the system properties [5]. Abstraction method has been successfully used to verify in many domains [91], [92].

e: SYMMETRY REDUCTION METHODS

Symmetry reduction method attempts to reduce the amount of state space. It is based on replacing sets of symmetrically similar states in a given model via a single representative class. Consider Figure 9 of a mutual-exclusion for two components a, b modeled by a *Kripke* structure. There are many obvious symmetries between the components. For example, when component a is in the critical section, component b is waiting, equivalently, when the state in b is in the critical section, a is waiting. It is an adequate way of verification if we could find such equivalent states and check only one state from each class instead of checking all individual states.

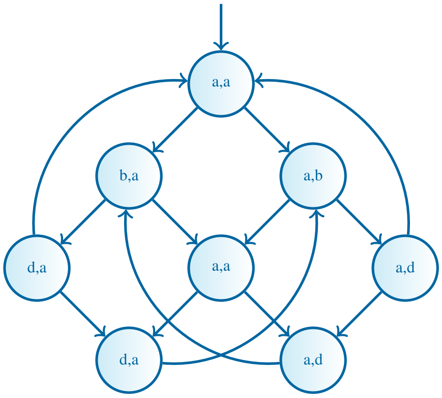

FIGURE 9. Mutual exclusion.

A constructed model M' of system M under symmetry methods is called a quotient structure and a given property *fi* holds for $M(fi \models M)$ if and only if, *fi* holds for $M'(fi \models M')$. Symmetry reduction methods have two difficulties: orbit problem, and constructive orbit problem. Orbit problem seeks to find if the two states a and $\bar{a}$ are in the equal orbit [93]. It is not known as NP-complete, but it is harder than isomorphism problems.

Constructive orbit problem (COP) is a representative function that replaces a set of symmetrically similar states in a given model by a single representative which is minimal. This problem falls into NP-hard problems [94].

f: HASH TABLES

A set of effective reachable states in contrast to the amount of possibly reachable states of a given state space is few. All effective states are stored somewhere in the system's memory. One way to retrieve these sparse states, which can be visited before and will be needed several times during checking, is by using hash tables. A hash table is an alternative to direct addressing into an ordinary array. This property of the hash table provides a simple way to examine an arbitrary state in a given array in O(1) time [95]. It can be considered as a yes-no method that is able to improve reachability analysis in verification processes.

G. J. Holzman in [96] speeds up the process of generating an exhaustive list of all visited states during a search for errors and check new generations of states against all pre-analyzed states by a hash table. Through a hash table, the states can be accessed quickly and decrease the amount of state for checking. To compress information more, they ignored storing the hash key itself (states) and only used the hash value (the address computed) to identify a state. The hashing discipline has been used to improve state storage and state comparison. Another example is the SPIN model checker [97] that utilizes hash functions.

4) BOTTOM-UP APPROACH

A verification process that can be done in a bottom-up manner prior to the entire state space construction. The state space will be checked bit by bit and the global properties can be deduced by combining their results. In the bottom-up approach, it deletes the states that are already checked from memory and can be re-verified when needed. On the other hand, the verification information from the verified-states can be reused without the need to re-verify. In contrast to compositional verification, bottom-up approaches do not involve system decomposition. The following two bottom-up approaches in alleviating SSE problem is discussed:

a: ON-THE-FLY METHOD

This method is an explicit method that is able to verify a system without storing the complete construction of the state space in memory. On-the-fly model-checking starts checking from an initial state and searches adjacent states to gain local knowledge about the state space in a stepwise manner. The key factor here is storing only the current path and verification along with the construction of the system state space. In other words, it does not postpone the verification until the state space construction is completed. Therefore, the counterexample of the properties that do not hold can be found and generated as early as possible. This property is the most important advantage of the on-the-fly method [98].

Another advantage of the method is that it reduces the memory requirements substantially because it already eliminates visited states from memory. On the other hand, eliminating already verified states may increase the run time during error searching. Since this method does not store the already-visited states in memory and may need to regenerate them over and over, then the time of exploration grows dramatically.

The methods themselves often employ depth-first-search (DFS) algorithms for searching through the state space. The run time of this method relies upon the number of states and the number of transitions. The DFS algorithm is divided into two categories: Nested DFS and strongly connected component (SCC). Nested DFS, firstly searches for accepting states. Secondly, it searches for cycles around accepted states. Despite the memory efficiency of this algorithm, it may lead to finding a very long trace of a counterexample. In [99], the authors proposed a method to achieve a minimal counterexample.

SCC-based on-the-fly methods find a strongly connected component from an initial state to a given state. If any violation is found, then it produces a counterexample trace that is strongly connected, and it includes at least one component. In comparison with nested DFS, it utilizes more memory, a larger stack, and finally a longer counterexample. J. Geldenhuys introduced Tarjan's algorithm [100] to improve this kind of model-checking. Furthermore, Geldenhuys and Valmari [101] improved Tarjan's algorithm in terms of finding an accepting cycle sooner and producing a shorter counterexample. On-the-fly methods have been combined with other methods such as symmetry reduction [102].

b: INCREMENTAL VERIFICATION

Incremental verification is one of those approaches that iteratively generates the SS of the system and verifies them until the overall properties of the system are satisfied. The key concept here is twofold: 1- preservation of the system properties when new increments have been added; 2- providing an appropriate way to avoid re-verifying the system when new increments in the higher level of verification have been added. Consequently, it reduces the whole verification effort. Incremental verification is discussed in detail in section V. [103] is a research falling under this class and is discussed in section V.

5) COMPOSITIONAL VERIFICATION

Compositional verification is a kind of divide-and-conquer approach which deals with SSE problem. It divides a large and complex problem into sub-problems and verifies each part separately. Contrary to its name, this method decomposes a given system into small components and then verifies the local properties of each component. One of the articles in this direction is [104]. The verifying of local properties of the sub-systems contributes to the deduction of the entire system property. Obviously, by using this approach the whole SS does not need to be constructed and the sub-systems are not as big as the system itself. Consequently, the state-space volume will be significantly reduced. In addition, it provides more insight into the system interactions. Compositional verification has several alternative methods which are explained below:

a: INTERFACE RULE

It makes an abstraction interface of component constraints and then proves the preservation of each local component by an interface rule. The idea behind using the interface abstraction is that in the composition process, only the properties are observable for other components should be checked. By hiding the rest of the properties, a huge amount of states will be reduced. Interface theory provides strong logical operations which are sound. The soundness of that is proved in [105]. Figure 10 shows a general schema for the interface rule method. P_1 and P_2 are two processes or two components which is equipped with their interface rules A_1 and A_2.

b: PARTITIONED TRANSITION RELATIONS

This method is based on the image of the states that produces a set of all successors of states A and pre-image which produces the predecessor of the set of states A' with a transition relation T [106]. Let the sets A and T which are given by

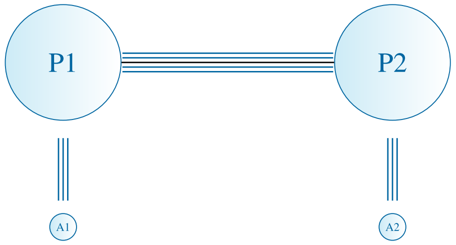

FIGURE 10. **Abstraction.**

a Boolean formula, then the image of A can be computed by the following formula:

$$\exists m[A(m) \wedge T(m, m')] \tag{2}$$

which existential m ($\exists m$) determines the quantification over all variables in the set of variables m.

In the first step, it constructs the transition relation N_i of each component i when exploring the system model and then composes all individual results to produce a global transition relation. In this way, the global transition relation will never be constructed explicitly. The formulas and steps for the synchronous systems are as the following, (3), as shown at the bottom of the page.

Clearly, each step depends on the previous step and the final partitioning strongly depends on the order that the variables are out qualified. It can be computed by optimum ordering of BDD (OBDD), however, finding an OBDD is complex and it needs special algorithms to find an optimum ordering. In [105], [107], the authors have presented an algorithm to compute an OBDD and improve the partition transition relations.

c: LAZY PARALLEL COMPOSITION

For all processes in this method, a restricted transition relation will be created. The restricted transition is more concise than the global transition relation itself [105]. Let R be a global transition relation and S a set of states. R' can be a restricted transition relation for the image s if it always satisfies the condition:

$$R'|_s = R|_s \tag{4}$$

The formula indicates that R and R' concur on transitions starting from the state s, however, R' has fewer nodes than R. This method simplifies the transition relation of each component by using the constraint operators before constructing the global transition relation.

$$R' = \bigwedge_{i...n} constraint(R_i, S) \tag{5}$$

R' must concur with the global relation transition R in the set of states S. As a result, producing successors of S by using the restricted transition R' produces the same result as using R. The total formula and steps that they take in comparison to the partial transition relation method is as the following:

$$\exists m'[A(m') \wedge \underbrace{(T_1(m, m')|s}_{step1} \wedge \underbrace{T_2(m, m')|s)}_{step2}] \tag{6}$$

In the formula, it is obvious that every step is independent, and it can be considered as an improvement of partition transition relations.

d: ASSUME-GUARANTEE REASONING

It has been proposed by Pnueli [108] where it verifies a single component of the system at a time. However, during the verification, a component needs to be associated with the assumption that the environment has a certain behavior, then if the other components of the system guarantee the behavior, it can be deduced that the behavior holds true for the entire system. Thus, two kinds of properties should be checked: firstly, the specific assumptions about the environment behavior. Secondly, guarantees that the assumptions hold. The basic rule of assume-guarantee can be formulated as the following:

$$\frac{\langle true\rangle M'\langle g\rangle \qquad \langle M\langle f\rangle}{\langle true\rangle M \quad \| \quad M'\langle f\rangle} \tag{7}$$

This method is discussed in detail in section V.

B. COMPARISON

These methods for mitigating SSE problem so far have involved addressing the memory concern. For example, by using a specific data structure like BDD, using heuristics, adding external memory, or dividing the problem into subproblems. These methods are completely different and the only shared characteristics between them are the way the SS is represented. The SS representation can be divided into two main paradigms in model-checking, *explicit and implicit*. Table 3 compares the reviewed methods based on them.

The success factor and challenges of the reviewed methods are summarized in two separate tables. Table 4 and 5 contains a list of the key features and challenges of each method that have been obtained in the literature and described in this section respectively.

$$\underbrace{\exists m_{\rho(n-1)}[\cdots \underbrace{\exists m_{\rho(1)}[\underbrace{\exists m_{\rho(0)}[A(m) \wedge T_{\rho(0)}(m, m')]}_{A_1} \wedge T_{\rho(1)}(m, m')]}_{A_2 \; \vdots} \wedge \cdots \wedge T_{\rho(n-1)}(m, m')]}_{A_n} \tag{3}$$

TABLE 3. Explicit and implicit methods.

State Space Explosion Reduction	explicit	implicit	State Space Explosion Reduction	explicit	implicit
Symbolic model-checking		✓	Genetic algorithm	✓	
Bounded model-checking		✓	Random walk	✓	
Garbage collection	✓		Ant colony	✓	
Partial order reduction	✓		Bloom filter	✓	
On-the-fly method	✓		Incremental verification	✓	✓
Expanding memory	✓	✓	Interface rules		✓
Symmetry reduction methods	✓		Partition transition relation		✓
Abstraction	✓		Lazy parallel composition		✓
Hash table	✓		Assume-guarantee reasoning	✓	✓

TABLE 4. Success factors of SSE reduction methods.

State Space Explosion Reduction Method	Specific features
Symbolic model-checking	by choosing a sufficient encoding of the system model, symbolic model checking can support verification of 10^{20} states in practice [107].
Bounded model-checking	it requires less by hand manipulation than other approaches like BDDs [63].
Garbage collection	its memory management and fragmentation is on the fly that leads to reduce SSE problem; it avoids suspending pointers being created [110].
Partial order reduction	it limits the searching of redundant interleaving that leads to search more states [79].
On-the-fly method	it is able to verify individual traces rather than the whole state space; it is able to produce counterexamples as early as possible [99].
Expanding memory	external memory capacity is infinite and not expensive.
Symmetry reduction methods	it replaces a set of symmetrically similar states in a given model by a single representative that is minimal that the similar states and traces present only once [94].
Abstraction	it handles more states by removing states with similar behaviors [5].
Compositional verification	the entire state space of the system will not be built completely, therefore by defining sufficient assumptions about the system and computational environment, SSE problem can reduce [106].
Hash table	it does not need to store the whole states instead a hash table is used; it empowered nested depth-first-search to search and check more states [111]
Genetic algorithm	it uses an intelligent search instead of searching exhaustively into the entire state space [29].
Random walk	its need for space is extremely low; it is able to be parallel [34], [36], [37].
Ant colony	coverage of all states is guarantee [112].
Incremental verification	it can be done before the system construction completes; the counterexamples can be found as early as possible [refer to section 4].

V. RESULTS ON HANDLING SSE IN CBSD

This section elaborates on the results of handling SSE in CBSD that have been found in the research literature. The search is conducted on all domains of CBSD as the focus of this research is to find the SSE reduction methods. The results indicate that despite the fact that deciding properties like liveness and deadlock-freeness in CBSD is NP-hard [120], [121]; model-checking have been successfully utilized to evaluate this kind of properties of CBSD. A particular subset of CBSD verification is concerned with addressing SSE problem. The methods that have been explained in the previous section could be used in CBSD as well. However, regarding the selected studies in this research, the frequently used methods in CBSD are assume-guarantee reasoning, interface rule, and incremental verification. The component-wise representation of the SS of these kinds of methods is one of the reasons for their popularity in CBSD. Assume-guarantee and interface rule are subsets of compositional verification which falls into the divide and conquer category. The philosophy behind these two methods is dividing a system into some sub-components, addressing the local properties of a subset of components independently, and then deducing the entire properties of the system properties. Incremental verification falls into a bottom-up category that is able to exploit

TABLE 5. Challenges of SSE reduction methods.

State Space Explosion Reduction Methods	Challenges
Symbolic model-checking	it uses BDDs which strongly depends on the ordering of its input variables [56], [57].
Bounded model-checking	it is capable to find only trivial properties and unable to check systems contains deep loops [113]; it can find long counterexample while short counterexamples are easier to understand [114].
Garbage collection	it consuming computer resources [115], [116]; it is hard to predict the pauses that garbage collection has done [117].
Partial order reduction	it does not sensitive in the ordering of traces [85].
On-the-fly method	it needs to regenerate already visited states which consequently lead to increase runtime [118].
Expanding memory	it leads to slower memory access, therefore it needs a proper Input/Output algorithm [21].
Symmetry reduction methods	it falls into NP-hard problems [95].
Abstraction	it needs a proper mapping function [118].
Compositional verification	breaking down a system is hard to do [119], [120]; verifying the systems with long chain circularity is difficult [118].
Hash table	it assigns a unique index number which may lead to a hash collision and missing the error states [111].
Genetic algorithm, Random walk, Ant colony	these three methods do not guarantee the exploring all global states, therefore it does not guarantee to find global properties; time to coverage is uncertain [112].
Bloom filter	there is no guarantee to find global properties, it may lead to false-positive results [38] .
Incremental verification	it needs to generate rules to guarantee the preservation of properties and generating rules to avoid re-verifying the previous levels [refer to section V].

TABLE 6. Challenges of SSE reduction methods in component-based verification.

State Space Explosion Reduction Methods	Challenges
Assume-guarantee Reasoning	decomposing system, detecting appropriate assumption, developing new rules to determine the correctness of assumptions, circularity
Interface-based verification	decomposing system, generating abstract constraint, refine the interface
Incremental verification	preserving of system properties when new increments are added, avoiding of re-verifying in the high level of verification

lower level verification information when small changes are applied, or components are added. Table 6 represents the characteristics of these three methods. In this section, these methods are described followed by the key features and potential challenges.

Assume-guarantee reasoning- involves three steps starting with D (*Three-D*): 1- Decomposing a given system S into its sub components $C_1, C_2, \ldots, C_n$, 2- Deriving assumption A_i about the environment for each C_i, 3- Defining rules to prove that properties of C_i guarantees the requirements φ of system S under assumption A_i.

Step 1 breaks up the entire system into sub-components. Applying the divide and conquer methods over CBSD which is already composed of multiple components may facilitate the decomposition step, but it is still a tedious task. Cobleigh *et al.* [118], [119] determined that finding an appropriate decomposition of a given system to verify by such methods is challenging.

Step 2 is the process of capturing the behavior that a given component C_i collects about its environment A_i. The most important key point resulting in a successful assumption check is detecting the appropriate assumptions for every component. Thus, a challenging question arises here: How to detect an appropriate assumption? The assumptions have been traditionally generated by users that have hard limited the assume-guarantee reasoning practically. Some proposals have been proposed to develop the assumptions automatically such as using learning assumptions [122], [123].

Step 3 is defining the rules to prove that the sub-component C_i guarantees its correct behavior under assumption A_i. Having rules to decide about the correctness of assumption is necessary. Thus, this question must be answered precisely: How to develop this kind of proof rules? The rules can be provided based on a set of theoretic operations [124].

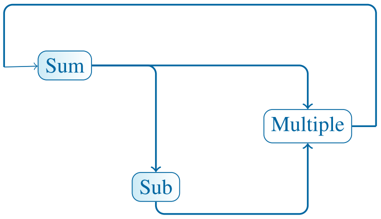

FIGURE 11. Interdependency between components.

Another challenge of this method is shown in Figure 11. Suppose a system with three components *Sum*, *Sub*, and *Multiple*. It is compulsory to proof satisfaction of component *Sum* to verify component *Sub*. Likewise, verifying component *Multiple* depends on *Sum* and *Sub*. This represents the problem of interdependent assumptions between components. Mutual interdependency between the components, like the dependency between component *Sum* and *Multiple* is another problem that is called *circularity*. Tackling this problem needs a set of sound and complete rules. In [117], the researchers prove that solving a long chain of circularity is difficult.

Other works based on assume-guarantee reasoning involve an algorithm based on a prefix-closed set of traces [124], an algebraic theory [125], [126], an algorithm named AGMC [127], an assume-guarantee verification for SOFA component model [128], interconnected systems [129] and SSE reduction for time systems [130]–[132].

Interface rule reasoning- Interface rule which has been presented in [5] is a set of abstract constraints for each single component in the systems. It restricts the behavior of components and assures the protection of local properties. The first key challenge is generating abstract constraints. This method can be used in compositional verification strategies by decomposing the interface of the system into sub-parts which represent the global properties. Then, the composition of individual components should satisfy the global properties of the interface. The second challenge of this method is decomposition.

The interface rules must include appropriate information to fulfill the compositional verification goal. It can be either traditionally prepared by users manually or generated automatically. Some works based on this method are discussed below.

Jin [117] proposed a formal framework for specifying and verifying component-based systems based on *interface automata (IA)*. IA has been used to describe interaction protocols of components and preserve the local properties of them to verify them independently. Another work on verifying component-based systems via interface rule is presented by Isazadeh and Karimpour [133]. They have proposed a formal model to specify the interface rule for communication protocols of components and then verify components according to this interface rule.

Ben-Hafaiedh *et al.* [134] have developed a framework for interface rule reasoning (in term of contract-based reasoning) for component-based design. In order to introduce such interface rule, they make benefit from a notation of I/A automata proposed by Henzinger and take into account sets of notations of constraints about composability and compatibility.

Another work presented in [135] on compositional verification of component-based in X-MAN. This work consists of two steps: 1- Vertical verification and Horizontal verification. Vertical verification guarantees that each atomic component satisfies its constraints (presented by an interface). 2- Horizontal verification which uses component constraint to verify the entire system. In this work, they suppose that the interface rules have been already attached to each component in the repository. Other works in interface based verification are [136]–[138].

Incremental verification- as its name implies, incremental verification incrementally checks the system behaviour. It can be done during design process. If a CBSD is constructing incrementally, verification can be done during the construction and iteratively check properties of unfinished system until the system is completed. Thus, for incremental verification of CBSD, it is vital to support incremental construction. In [139], property feasibility of incremental construction have been proved. Let system S be constructed by inc_i increments from initial inc_0 where $inc_1 \subseteq inc_2 \subseteq inc_3 \ldots$ and $inc_i \subseteq inc_{i+1}$ which means inc_{i+1} contains the behaviours of inc_i [139]. We say φ is the properties of the system S such that $\varphi_{inc_1} \subseteq \varphi_{inc_2} \subseteq \varphi_{inc_3} \subset \ldots$ where $\varphi_{inc_1} \subseteq \varphi_{inc_2}$ means the properties that is satisfied in inc_i is preserved in the next step of construction inc_{i+1}. Thus, preservation of the system properties when new increments are added is one of the key challenges in incremental verification.

Second, providing an appropriate way to avoid re-verifying the system when new increments in the higher level of verification are added is required. Verifying the whole system after every small change is not efficient. Therefore, providing a rule to deal with this issue can be considered as an effective improvement in formal verification, because it leads to reduce the whole verification effort significantly.

An incremental verification based on interaction invariants for component-based design has been proposed in [140]. It is an invariant-based method for verification in the BIP component-based model. The interaction invariants involve locations of multiple components and express constraints on global SS induced via interactions. A method called binary behavioral constraints (BBC) has been proposed to symbolically compute the interaction invariants. The methods completely define the influence of interactions of a composite component on the other component's behavior To reuse those invariants when new increments have been added they decompose the BBC and use two new methods to enhance scalability. Finally, the constraints and computations are represented by BDD. As discussed in Section 4, BDD strongly depends on input ordering and without it, the BDD may grow and quickly exceed the memory capacity. Despite proposing a sufficient technique to reduce the BDD, ordering BDD falls into NP-problems and does not have exact solutions [56], [57].

Other work concerning incremental verification for dynamic CBSD is defined in [141]. This work can verify CBSD whose components and structure change dynamically at runtime. The method, called INVEST, improved compositional verification by adding an incremental strategy to reverify a system after any removal, modification, and addition of components. Initially, the system will be verified by typical compositional verification and assume-guarantee reasoning. The incremental verification executes when any changes occur in the system.

VI. DISCUSSION AND CONCLUSION

This work on one hand reviews, briefly discusses, characterizes, and classifies existing methods of SSE reduction methods into five categories. On the other hand, it investigates the methods for alleviating SSE problem that have been utilized in CBSD. In section 3, RQ (1, 2, 3) have been answered. The state-of-the-art mitigation methods for SSE problem have been identified and explained. The key features and challenges of them are summarized as well. All these information have led to setting up a classification for common SSE reduction methods. RQs (4, 5) have been discussed in section 4. We demonstrated the common SSE reduction methods in CBSD and the potential challenges that have been obtained in the literature.

The general clue for this research is that despite proposing many methods for solving the bottleneck of model-checking, SSE problem still remains an obstacle in the worst case and has not been solved completely yet. This research provides a basis for many stakeholders such as component-based developers that need to select the most appropriate method for verifying their system, organizations that desire to create model checkers, and researchers seeking to set their research directions.

Having all the aforementioned information about SSE reduction methods, now we are in the position to discuss the proper method to be utilized in CBSD. Among the common methods for alleviating SSE problem in CBSD, is by using compositional verification in the form of either the assume-guarantee or interface rules. However, in such methods, the entire verification problem should be decomposed into the smaller task of its components and checked individually. Then, after decomposition, some difficulties such as interdependency between components, circularity, finding assumptions will arise. Utilizing such methods in order to verify CBSD obviously is limited by several issues.

On the other hand, applying incremental verification in CBSD may have some advantages. To begin with, such methods, omit the tedious task of breaking up the verification problem. Verifying a system in a bottom-up manner or bitwise during incremental construction reduces the verification effort, rather than decomposing the system after the entire construction is finished and then verifying each part individually. The implementation of incremental construction and verification by [139], [140] has determined the possibility of this.

Another advantage of incremental verification is that counterexamples and error traces can be found as early as possible. In other words, the counterexamples can be created before the whole system is constructed. It is very useful to reveal error states before going through the higher levels of construction. However, all component-based models have yet to support incremental construction. Thus, providing a way to incrementally construct and verify component-based systems is a major direction for our future work. It might be possible by utilizing a component-based model with encapsulation mechanisms like what is presented in [142]–[145].

REFERENCES

[1] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 709–724, Oct. 2007.

[2] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, and R. Seacord, "Volume II: Technical concepts of component-based software engineering," Carnegie Mellon Softw. Eng. Inst., USA, Tech. Rep. CMU/SEI-2000-TR-008, 2000, pp. 26–29.

[3] W. K. Ehrlich, A. Iannino, B. Prasanna, J. P. Stampfel, and J. R. Wu, "How faults cause software failures: Implications for software reliability engineering," in *Proc. Int. Symp. Softw. Rel. Eng.*, 1991, pp. 233–241.

[4] E. E. Ogheneovo, "Software dysfunction: Why do software fail?" *J. Comput. Commun.*, vol. 2, no. 6, pp. 25–35, 2014.

[5] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[6] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *Proc. Int. Symp. Program.* Berlin, Germany: Springer, 1982, pp. 337–351.

[7] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 1–54, 2009.

[8] C. Tian, S. Liu, and Z. Duan, "Abstract model checking with SOFL hierarchy," in *Proc. Int. Work. Struct. Object-Oriented Formal Lang. Method.* Berlin, Germany: Springer, 2012, pp. 71–86.

[9] R. Pelánek, "Fighting state space explosion: Review and evaluation," in *Proc. Int. Work. Formal Methods Ind. Crit. Syst.* Berlin, Germany: Springer, 2008, pp. 37–52.

[10] V. Rafe, M. Rahmani, and K. Rashidi, "A survey on coping with the state space explosion problem in model checking," *Int. Res. J. Appl. Basic Sci.*, vol. 4, no. 6, pp. 1379–1384, 2013.

[11] N. Bertrand, "Model checking randomized distributed algorithms," *ACM SIGLOG News*, vol. 7, no. 1, pp. 35–45, Feb. 2020.

[12] S. Gabmeyer, P. Brosch, and M. Seidl, "A classification of model checking-based verification approaches for software models," in *Proc. 2nd Int. Work. Verification Model Transformation (VOLT)*, 2013.

[13] Y.-F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang, "Automated assume-guarantee reasoning through implicit learning," in *Proc. Int. Conf. Comput. Aided Verification*, 2010, pp. 511–526.

[14] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School Software Engineering*. Berlin, Germany: Springer, 2011, pp. 1–30.

[15] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, ''Modular verification of software components in c,'' *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 388–402, Jun. 2004.

[16] E. M. Clarke, E. A. Emerson, and J. Sifakis, ''Model checking: Algorithmic verification and debugging,'' *Commun. ACM*, vol. 52, no. 11, pp. 74–84, Nov. 2009.

[17] R. A. B. Silva, J. M. P. D. Oliveira, and J. S. Pinto, ''A case study on model checking and deductive verification techniques of safety-critical software,'' Universidade Federal de Campina Grande, Campina Grande, Brazil, Tech. Rep., 2012.

[18] T. Schäfer, A. Knapp, and S. Merz, ''Model checking UML state machines and collaborations,'' *Electron. Notes Theor. Comput. Sci.*, vol. 55, no. 3, pp. 357–369, Oct. 2001.

[19] G. Liu and C. Jiang, ''Petri net based model checking for the collaborative-ness of multiple processes systems,'' in *Proc. IEEE 13th Int. Conf. Netw., Sens., Control (ICNSC)*, Apr. 2016, pp. 1–6.

[20] P. Lamborn and E. A. Hansen, ''Layered duplicate detection in external-memory model checking,'' in *Proc. Int. SPIN Work. Model Checking Softw.* Berlin, Germany: Springer, 2008, pp. 160–175.

[21] L. Wu, H. Huang, K. Su, S. Cai, and X. Zhang, ''An I/O efficient model checking algorithm for large-scale systems,'' *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 5, pp. 905–915, May 2015.

[22] F. Lerda and W. Visser, ''Addressing dynamic issues of program model checking,'' in *Proc. Int. SPIN Work. Model Checking Softw.* Berlin, Germany: Springer, 2001, pp. 80–102.

[23] R. Iosif and R. Sisto, ''Using garbage collection in model checking,'' in *Proc. Int. SPIN Work. Model Checking Softw.* Berlin, Germany: Springer, 2000, pp. 20–33.

[24] A. W. Appel, *Modern Compiler Implementation*. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[25] M. Might, B. Chambers, and O. Shivers, ''Model checking via γCFA,'' in *Proc. Int. Work. Verification, Model Checking, Abstract Interpretation*. Berlin, Germany: Springer, 2007, pp. 59–73.

[26] P. Lengauer and H. Mössenböck, ''The taming of the shrew: Increasing performance by automatic parameter tuning for java garbage collectors,'' in *Proc. 5th ACM/SPEC Int. Conf. Perform. Eng.*, Mar. 2014, pp. 111–122.

[27] T. Zheng and Y. Liu, ''Genetic algorithm for generating counterexample in stochastic model checking,'' in *Proc. VII Int. Conf. Netw., Commun. Comput.*, 2018, pp. 92–96.

[28] P. Godefroid and S. Khurshid, ''Exploring very large state spaces using genetic algorithms,'' in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2002, pp. 266–280.

[29] R. Yousefian, V. Rafe, and M. Rahmani, ''A heuristic solution for model checking graph transformation systems,'' *Appl. Soft Comput.*, vol. 24, pp. 169–180, Nov. 2014.

[30] Y. Ma, Z. Cao, and Y. Liu, ''A probabilistic assume-guarantee reasoning framework based on genetic algorithm,'' *IEEE Access*, vol. 7, pp. 83839–83851, 2019.

[31] E. Atashpaz-Gargari and C. Lucas, ''Imperialist competitive algorithm: An algorithm for optimization inspired by imperialistic competition,'' in *Proc. IEEE Congr. Evol. Comput.*, May 2007, pp. 4661–4667.

[32] S. Mirjalili, S. M. Mirjalili, and A. Lewis, ''Grey wolf optimizer,'' *Adv. Eng. Softw.*, vol. 69, pp. 46–61, Mar. 2014.

[33] S. Zangbari Koohi, N. A. W. Abdul Hamid, M. Othman, and G. Ibragimov, ''Raccoon optimization algorithm,'' *IEEE Access*, vol. 7, pp. 5383–5399, 2019.

[34] R. Pelánek, T. Hanžl, I. Černá, and L. Brim, ''Enhancing random walk state space exploration,'' in *Proc. 10th Int. Workshop Formal Methods Ind. Crit. Syst.*, 2005, pp. 98–105.

[35] D. Owen, T. Menzies, M. Heimdahl, and J. Gao, ''On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate,'' in *Proc. 28th Annual NASA Goddard Softw. Eng. Work.*, 2003, pp. 75–81.

[36] H. Sivaraj and G. Gopalakrishnan, ''Random walk based heuristic algorithms for distributed memory model checking,'' *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 1, pp. 51–67, Sep. 2003.

[37] T. H. Bui and A. Nymeyer, ''Heuristic sensitivity in guided random-walk based model checking,'' in *Proc. 7th IEEE Int. Conf. Softw. Eng. Formal Methods*, 2009, pp. 125–134.

[38] U. Stern and D. L. Dill, ''Improved probabilistic verification by hash compaction,'' in *Proc. Adv. Res. Work. Conf. Correct Hardw. Design Verification Methods*. Berlin, Germany: Springer, 1995, pp. 206–224.

[39] U. Stern and D. L. Dill, ''A new scheme for memory-efficient probabilistic verification,'' *Formal Description Techn.*, vol. 4, pp. 333–348, May 1996.

[40] P. C. Dillinger and P. Manolios, ''Bloom filters in probabilistic verification,'' in *Proc. Int. Conf. Formal Methods Comput.-Aided Design*. Berlin, Germany: Springer, 2004, pp. 367–381.

[41] L. M. Duarte, L. Foss, F. R. Wagner, and T. Heimfarth, ''Model checking the ant colony optimisation,'' in *Distributed, Parallel and Biologically Inspired Systems*. Berlin, Germany: Springer, pp. 221–232, 2010.

[42] W. Zhu, P. Feng, and M. Deng, ''An approximate CTL model checking approach,'' in *Proc. IEEE 10th Int. Conf. Softw. Eng. Service Sci. (ICSESS)*, May 2019, pp. 646–648.

[43] W. Zhu, H. Wu, and M. Deng, ''LTL model checking based on binary classification of machine learning,'' *IEEE Access*, vol. 7, pp. 135703–135719, 2019.

[44] L. Bortolussi, L. Cardelli, M. Kwiatkowska, and L. Laurenti, ''Central limit model checking,'' *ACM Trans. Comput. Log.*, vol. 20, no. 4, pp. 1–35, Sep. 2019.

[45] S. Donatelli, ''Markov regenerative processes solution and stochastic model checking: An on-the-fly approach,'' in *Proc. 12th EAI Int. Conf. Perform. Eval. Methodol. Tools*, Mar. 2019, pp. 1–5.

[46] N. Rezaee and H. Momeni, ''A hybrid meta-heuristic approach to cope with state space explosion in model checking technique for deadlock freeness,'' *J. AI Data Mining*, vol. 8, no. 2, pp. 189–199, 2020.

[47] Y. Liu and C. He, ''A heuristics-based incremental probabilistic model checking at runtime,'' in *Proc. IEEE 11th Int. Conf. Softw. Eng. Service Sci. (ICSESS)*, Oct. 2020, pp. 355–358.

[48] A. Khemiri, M. E. A. Hamri, C. Frydman, and J. Pinaton, ''Limiting state space explosion of model checking using discrete event simulation: Combining DEVS and PROMELA,'' in *Proc. Summer Simul. Conf.*, 2019, pp. 1–12.

[49] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill, and L. J. Hwang, ''Symbolic model checking: 1020 states and beyond,'' *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, Jun. 1992.

[50] Y. Thierry-Mieg, ''Symbolic model-checking using ITS-tools,'' in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2015, pp. 231–237.

[51] S. Guellouz, A. Benzina, M. Khalgui, G. Frey, Z. Li, and V. Vyatkin, ''Designing efficient reconfigurable control systems using IEC61499 and symbolic model checking,'' *IEEE Trans. Autom. Sci. Eng.*, vol. 16, no. 3, pp. 1110–1124, Jul. 2019.

[52] P. Kissmann and J. Hoffmann, ''BDD ordering heuristics for classical planning,'' *J. Artif. Intell. Res.*, vol. 51, pp. 779–804, Dec. 2014.

[53] R. E. Bryant, ''Symbolic Boolean manipulation with ordered binary-decision diagrams,'' *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, Sep. 1992.

[54] P. Prasad, A. Assi, A. Harb, and V. Prasad, ''Binary decision diagrams: An improved variable ordering using graph representation of Boolean functions,'' *Int. J. Comput. Sci.*, vol. 1, no. 1, pp. 1–7, 2006.

[55] P. K. Sharma and N. Kumar Singh, ''Improved BDD compression by combination of variable ordering techniques,'' in *Proc. Int. Conf. Commun. Signal Process.*, Apr. 2014, pp. 617–621.

[56] B. Bollig and I. Wegener, ''Improving the variable ordering of OBDDs is NP-complete,'' *IEEE Trans. Comput.*, vol. 45, no. 9, pp. 993–1002, Aug. 1996.

[57] B. Bollig, ''On the width of ordered binary decision diagrams,'' in *Proc. Int. Conf. Combinat. Optim. Appl.* Cham, Switzerland: Springer, 2014, pp. 444–458.

[58] H. Ouni, K. Klai, C. A. Abid, and B. Zouari, ''Parallel symbolic observation graph,'' in *Proc. IEEE Int. Symp. Parallel Distrib. Process. with Appl. IEEE Int. Conf. Ubiquitous Comput. Commun. (ISPA/IUCC)*, Dec. 2017, pp. 770–777.

[59] H. Ouni, K. Klai, C. A. Abid, and B. Zouari, ''Towards parallel verification of concurrent systems using the symbolic observation graph,'' in *Proc. 19th Int. Conf. Appl. Concurrency Syst. Design (ACSD)*, Jun. 2019, pp. 23–32.

[60] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, ''Symbolic model checking without BDDs,'' in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 1999, pp. 193–207.

[61] A. Biere, E. Clarke, R. Raimi, and Y. Zhu, ''Verifying safety properties of a powerpc-microprocessor using symbolic model checking without BDDs,'' in *Proc. Int. Conf. Comput. Aided Verification*. Springer, 1999, pp. 60–71.

[62] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, ''Bounded model checking,'' *Adv. Comput.*, vol. 58, pp. 117–148, May 2003.

[63] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, ''Bounded model checking using satisfiability solving,'' *Formal Methods Syst. Des.*, vol. 19, no. 1, pp. 7–34, 2001.

[64] W. Spijkerman, "Marking pocket states for bounded on-the-fly model checking," Model Checking Softw., Tech. Rep. 2008.

[65] H. A. Kautz, "Planning as satisfiability," in *Proc. ECAI*, vol. 92, 1992, pp. 359–363.

[66] F. Van Harmelen, V. Lifschitz, and B. Porter, *Handbook of Knowledge Representation.*, vol. 1. Amsterdam, The Netherlands: Elsevier, 2008.

[67] A. Armando, R. Carbone, and L. Compagna, "SATMC: A sat-based model checker for security-critical systems," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2014, pp. 31–45.

[68] Q.-S. Phan, P. Malacaria, and C. S. Pęžsreanu, "Concurrent bounded model checking," *ACM SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 1–5, Feb. 2015.

[69] B. Woåna-Szczeániak, "SAT-based bounded model checking for weighted deontic interpreted systems," *Fundam. Inf.*, vol. 143, nos. 1–2, pp. 173–205, Feb. 2016.

[70] Y. Zhang, K. Chakrabarty, Z. Peng, A. Rezine, H. Li, P. Eles, and J. Jiang, "Software-based self-testing using bounded model checking for Out-of-Order superscalar processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 3, pp. 714–727, Mar. 2020.

[71] K. Bae and J. Lee, "Bounded model checking of signal temporal logic properties using syntactic separation," in *Proc. ACM Program. Lang.*, vol. 3, Jan. 2019, pp. 1–30.

[72] O. Inverso and C. Trubiani, "Parallel and distributed bounded model checking of multi-threaded programs," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 202–216.

[73] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 69–83, Feb. 2009.

[74] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," *Handbook satisfiability*, vol. 185, pp. 825–885, Oct. 2009.

[75] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, pp. 957–974, Jul. 2012.

[76] M. Amin Alipour and A. Groce, "Bounded model checking and feature omission diversity," 2016, *arXiv:1610.08020*. [Online]. Available: http://arxiv.org/abs/1610.08020

[77] M. G. Meulen, F. P. Stappers, and T. A. Willemse, "Breadth-bounded model checking," Dept. Comput. Sci., Technische Univ. Eindhoven, Eindhoven, The Netherlands, Tech. Rep. 903, 2009.

[78] P. Godefroid, "Using partial orders to improve automatic verification methods," in *Proc. Int. Conf. Comput. Aided Verification*, 1990, pp. 176–185.

[79] A. Valmari, "Stubborn sets for reduced state space generation," in *Proc. Int. Conf. Appl. Theory Petri Nets*. Berlin, Germany: Springer, 1989, pp. 491–515.

[80] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-order reduction in symbolic state space exploration," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 1997, pp. 340–351.

[81] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 110–121, Jan. 2005.

[82] K. McMillan, "An approach to the state explosion problem," Ph.D. dissertation, Dept. Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, 1992.

[83] E. Clarke, "The birth of model checking," in *Proc. Model Checking*, 2008, pp. 1–26.

[84] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *Int. J. Softw. Tools Technol. Transf.*, vol. 2, no. 3, pp. 279–287, Nov. 1999.

[85] A. Valmari, "A stubborn attack on state explosion,"in *Proc. Int. Conf. Comput. Aided Verification*, 1990, pp. 156–165.

[86] D. Peled, "Combining partial order reductions with on-the-fly model-checking," in *Proc. Int. Conf. Comput. Aided Verification* 1994, pp. 377–390.

[87] A. J. Robinson and A. Voronkov, *Handbook Automated Reasoning*, vol. 1. Cambridge, MA, USA: MIT Press, 2001.

[88] P. Cousot, "Abstract interpretation," *ACM Comput. Surv.*, vol. 28, no. 2, pp. 324–328, 1996.

[89] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[90] G. J. Holzmann, P. Godefroid, and D. Pirottin, "Coverage preserving reduction strategies for reachability analysis," in *Proc. 12th IFIP WG*, vol. 6, 2013, pp. 349–363.

[91] W. Oortwijn, D. Gurov, and M. Huisman, "An abstraction technique for verifying shared-memory concurrency," *Appl. Sci.*, vol. 10, no. 11, p. 3928, Jun. 2020.

[92] E. André, L. Fribourg, J.-M. Mota, and R. Soulat, "Verification of an industrial asynchronous leader election algorithm using abstractions and parametric model checking," in *Proc. Int. Conf. Verification, Model Checking, Abstract Interpretation*, 2019, pp. 409–424.

[93] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla, "Symmetry reductions in model checking," *Int. Conf. Comput. Aided Verification*, 1998, pp. 147–158.

[94] L. Babai and E. M. Luks, "Canonical labeling of graphs," in *Proc. 15th Annu. ACM Symp. Theory Comput.*, 1983, pp. 171–183.

[95] T. H. Cormen, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.

[96] G. J. Holzmann, "An improved protocol reachability analysis technique," *Softw., Pract. Exper.*, vol. 18, no. 2, pp. 137–161, Feb. 1988.

[97] G. J. Holzmann, "An analysis of bitstate hashing," in *Formal Methods System Design*, vol. 13. Cham, Switzerland: Springer, 1998, pp. 289–307.

[98] S. Schwoon and J. Esparza, "A note on on-the-fly verification algorithms," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2005, pp. 174–190.

[99] P. Gastin, P. Moro, and M. Zeitoun, "Minimization of counterexamples in spin," in *Proc. Int. SPIN Work. Model Checking Softw.* Berlin, Germany: Springer, 2004, pp. 92–108.

[100] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, Jun. 1972.

[101] J. Geldenhuys and A. Valmari, "Tarjan's algorithm makes on-the-fly LTL verification more efficient," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Springer, 2004, pp. 205–219.

[102] R. Patel, K. Patel, and D. Patel, "On-the-fly symmetry reduction of explicitly represented probabilistic models," in *Proc. Int. Conf. Distrib. Comput. Internet Technol.*, 2015, pp. 203–206.

[103] S. Bensalem, M. Bozga, B. Boyer, and A. Legay, "Incremental generation of linear invariants for component-based systems," in *Proc. 13th Int. Conf. Appl. Concurrency Syst. Design*, Jul. 2013, pp. 80–89.

[104] Y. Phyo, C. M. Do, and K. Ogata, "Toward development of a tool supporting a 2-layer divide & conquer approach to leads-to model checking," in *Proc. Int. Conf. Adv. Inf. Technol.*, 2019, pp. 250–255.

[105] S. Berezin, S. Campos, and E. M. Clarke, *Compositional reasoning in model checking*. Berlin, Germany: Springer, 1998, pp. 81–102.

[106] J. Burch, E. M. Clarke, and D. Long, "Symbolic model checking with partitioned transition relations," *Citeseer*, vol. 7, pp. 49–58, Aug. 1991.

[107] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using model checking," in *Proc. 27th ACM/IEEEDesign Autom. Processing*, Oct. 1990, pp. 46–51.

[108] A. Pnueli, *In Transition From Global to Modular Temporal Reasoning About Programs*. Berlin, Germany: Springer, 1985, pp. 123–144.

[109] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: Art Automation Memory Management*. Boca Raton, FL, USA: CRC Press, 2016.

[110] G. J. Holzmann, "State compression in spin: Recursive indexing and compression training runs," in *Proc. 3rd Int. Spin workshop*, 1997, pp. 1–9.

[111] V. Selvi and D. R. Umarani, "Comparative analysis of ant colony and particle swarm optimization techniques," *Int. J. Comput. Appl.*, vol. 5, no. 4, pp. 1–6, Aug. 2010.

[112] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.

[113] V. Schuppan and A. Biere, "Shortest counterexamples for symbolic model checking of LTL with past," in *Proc. Int. Conf. Tools Algorithms for Construction Anal. Syst.*, 2005, pp. 493–509.

[114] B. Zorn, "The measured cost of conservative garbage collection," *Softw., Pract. Exper.*, vol. 23, no. 7, pp. 733–756, Jul. 1993.

[115] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. Explicit memory management," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 313–326, Oct. 2005.

[116] F. Siebert, "Constant-time root scanning for deterministic garbage collection," in *Proc. Int. Conf. Compiler Construct.*, 2001, pp. 304–318.

[117] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "Compositional verification for component-based systems and application," *IET Softw.*, vol. 4, no. 3, p. 181, 2010.

[118] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke, "Breaking up is hard to do: An investigation of decomposition for assume-guarantee reasoning," in *Proc. Int. Symp. Softw. Test. Anal.*, 2006, pp. 97–108.

[119] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke, "Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, p. 7, 2008.

[120] M. Martens, C. Minnameier, and M. Majster-Cederbaum, "Deciding liveness in component-based systems is NP-hard," *Manuskripte/Reihe Informatik*, vol. 6, p. 45, May 2006.

[121] C. Minnameier, *Deadlock-detection component-based System is NP-hard*. Baden-Wärttemberg, Germany: Univ. Mannheim/Institut für Informatik, 2006.

[122] F. He, S. Mao, and B.-Y. Wang, "Learning-based assume-guarantee regression verification," in *Proc. Int. Conf. Comput. Aided Verification*, 2016, pp. 310–328.

[123] K. Abd Elkader, O. Grumberg, C. S. Päreanu, and S. Shoham, "Automated circular assume-guarantee reasoning," *Formal Aspects Comput.*, vol. 30, no. 5, pp. 571–595, Sep. 2018.

[124] C. Chilton, B. Jonsson, and M. Kwiatkowska, "Assume-guarantee reasoning for safe component behaviours," in *Proc. Int. Work. Formal Aspects Compon. Softw.*, 2012, pp. 92–109.

[125] A. Müller, S. Mitsch, W. Retschitzegger, W. Schwinger, and A. Platzer, "A component-based approach to hybrid systems safety verification," in *Proc. Int. Conf. Integr. Formal Methods*, 2016, pp. 441–456.

[126] C. J. Chilton, "An algebraic theory of componentised interaction," Ph.D. dissertation, Dept. Comput. Sci., Univ. Oxford Wolfson Building, Oxford, U.K., 2013.

[127] D. Hoang-Minh, T. Le-Khanh, and P. Ngoc Hung, "An assume-guarantee model checker for component-based systems," in *Proc. RIVF Int. Conf. Comput. Commun. Technol.-Res., Innov., Vis. Future (RIVF)*, Nov. 2013, pp. 22–26.

[128] P. Parizek and F. Plasil, "Assume-guarantee verification of software components in SOFA 2 framework," *IET Softw.*, vol. 4, no. 3, pp. 210–211, 2010.

[129] M. Al Khatib and M. Zamani, "Controller synthesis for interconnected systems using parametric assume-guarantee contracts," in *Proc. Amer. Control Conf. (ACC)*, Jul. 2020, pp. 5419–5424.

[130] H.-V. Tran, P. N. Hung, and D. Van Hung, "On improvement of assume-guarantee verification method for timed component-based software," in *Proc. 10th Int. Conf. Knowl. Syst. Eng. (KSE)*, Nov. 2018, pp. 270–275.

[131] H.-V. Tran, Q.-T. Nguyen, and P. N. Hung, "On implementation of the improved assume-guarantee verification method for timed systems," in *Proc. 10th Int. Symp. Inf. Commun. Technol.*, 2019, pp. 457–464.

[132] K. Ghasemi, S. Sadraddini, and C. Belta, "Compositional synthesis via a convex parameterization of assume-guarantee contracts," in *Proc. 23rd Int. Conf. Hybrid Syst., Comput. Control*, Apr. 2020, pp. 1–10.

[133] A. Isazadeh and J. Karimpour, "A new formalism for mathematical description and verification of component-based systems," *J. Supercomput.*, vol. 49, no. 3, pp. 334–353, Sep. 2009.

[134] I. Ben-Hafaiedh, S. Graf, and S. Quinton, "Reasoning about safety and progress using contracts," in Proc. *Int. Conf. Formal Eng. Methods*, 2010, pp. 436–451.

[135] N. He, D. Kroening, T. Wahl, K.-K. Lau, F. Taweel, C. Tran, P. Rümmer, and S. Sharma, "Component-based design and verification in X-MAN," in *Proc. Embedded Real Time Softw. Syst.*, 2012, pp. 1–5.

[136] C. Sun, N. Xi, J. Li, Q. Yao, and J. Ma, "Verifying secure interface composition for component-based system designs," in *Proc. 21st Asia–Pacific Softw. Eng. Conf.*, vol. 1, 2014, pp. 359–366.

[137] A. Cimatti and S. Tonetta, "Contracts-refinement proof system for component-based embedded systems," *Sci. Comput. Program.*, vol. 97, pp. 333–348, Jan. 2015.

[138] F. Howar, T. Kahsai, A. Gurfinkel, and C. Tinelli, "Trusting outsourced components in flight critical systems," *AIAA Infotech, Aerosp.*, vol. 7, p. 1868, May 2015.

[139] K.-K. Lau, K.-Y. Ng, T. Rana, and C. M. Tran, "Incremental construction of component-based systems," in *Proc. 15th ACM SIGSOFT Symp. Compon. Based Softw. Eng.*, 2012, pp. 41–50.

[140] S. Bensalem, M. Bozga, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, "Component-based verification using incremental design and invariants," *Softw. Syst. Model.*, vol. 15, no. 2, pp. 427–451, May 2016.

[141] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *Proc. 16th Int. ACM Sigsoft Symp. Component-Based Softw. Eng.*, 2013, pp. 33–42.

[142] F. Nejati, A. A. Abd Ghani, N. K. Yap, and A. B. Jafaar, "PUTRACOM: A concurrent component model with exogenous connectors," *IEEE Access*, vol. 6, pp. 15446–15456, 2018.

[143] F. Nejati, N. K. Yap, A. A. Abd Ghani, and A. Jaffar, "PUTRACOM: A formalism of a novel component model," *Int. J. Adv. Sci., Eng. Inf. Technol.*, vol. 10, no. 4, p. 1444, Aug. 2020.

[144] T. Rana and A. Baz, "A generalised coordination design pattern for the exman component model," *IEEE Access*, vol. 8, pp. 115461–115475, 2020.

[145] K.-K. Lau, M. Ornaghi, and Z. Wang, "A software component model and its preliminary formalisation," *Formal Methods Compon. Objects*, vol. 7, pp. 1–21, May 2005.

FARANAK NEJATI received the M.Sc. degree in software engineering from Tabriz University, Iran, and the Ph.D. degree from the Department of Software Engineering, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia (UPM), Malaysia. Her research interests include component-based software development, software composition, software connectors, formal modeling, formal verification, model checking, evolutionary algorithms, and artificial intelligence.

ABDUL AZIM ABD GHANI received the B.Sc. degree in mathematics/computer science from Indiana State University, in 1984, the M.Sc. degree in computer science from the University of Miami, in 1985, and the Ph.D. degree in software engineering from the University of Strathclyde, in 1993. He is currently a Professor with the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia. His research interests include software engineering, software measurement, software quality, and security in computing.

NG KENG YAP (Member, IEEE) received the B.Sc. degree in Computer Science and the M.Sc. degree from Universiti Putra Malaysia (UPM), and the Ph.D. degree from Manchester. He is currently a Senior Lecturer with the Department of Software Engineering and Information System, Universiti Putra Malaysia. His research interests include software architecture, software engineering such as component-based, incremental software composition, and software connectors, software metric, data science, and business analytics.

AZMI BIN JAFAAR received the B.Sc. degree in mathematics and computer science and the M.Sc. degree in mathematics from Indiana University, USA, and the Ph.D. degree in mathematical programming from Universiti Putra Malaysia, in 1997. He is currently an Associate Professor with the Department of Software Engineering and Information System, Universiti Putra Malaysia. His research interests include software engineering, discrete structures, mathematical programming, and empirical methods in computer science. He is also a member of International Association of Engineers (IAENGI) and has been a member of the Malaysian Mathematical Society (PERSAMA), since 1988.