Scalable Formal Verification of Modular Concurrent Systems: A Survey of Techniques, Tools and Challenges

Sawsen Khlifa[1], Chiheb Ameur Abid[2], Asma ben Letaifa[3], Belhassen Zouari[4]
Higher School of Communications of Tunis, University of Carthage Ariana, Tunisia[1,3]
Faculty of Sciences of Tunis, University of Tunis El Manar Tunis, Tunisia[2]
LR11TIC05, Mediatron Lab, 2083, Ariana, Tunisia[1,2,3]
ICL-Junia, Université Catholique de Lille, LITL, F-59000 Lille, France[4]

***Abstract*—The increasing complexity of distributed and concurrent systems raises pressing challenges for ensuring correctness and reliability. Formal verification, and in particular model checking, offers a rigorous foundation to validate system properties, yet suffers from the well-known state space explosion problem. This difficulty is especially acute in modular architectures, where local behaviors intertwine with synchronization across components. This paper provides a structured survey of the main techniques designed to overcome these challenges, including state space reduction, abstraction, compositional reasoning, symbolic approaches, and distributed verification. We also review representative tools such as SPIN, NuSMV, LTSmin, DiVinE, and STORM, assessing their capabilities and limitations in handling modular and concurrent models. Building on this landscape, we position the Reduced Distributed State Space (RDSS) as a novel framework that addresses key scalability limits. RDSS reduces global complexity into module-specific meta-graphs, ensures stuttering equivalence, and enables local model checking without exploring the full global state space. Comparative experiments demonstrate significant gains over existing approaches, particularly for systems where modules are not all synchronized on the same transitions. We conclude by identifying open challenges and future research directions, including distributed implementations, AI-driven heuristics, and hybrid reductions. Our survey underscores the importance of structural awareness in modern verification workflows and establishes RDSS as a promising foundation for scalable verification of modular concurrent systems.**

***Keywords*—Distributed systems; state space; Modular Petri Net; formal verification; state explosion problem; model checking; temporal logic; reduction techniques; RDSS; ROS2; scalability; modularity; model checking**

I. Introduction

The escalating complexity of modern concurrent and distributed systems has significantly exacerbated the requirement for rigorous verification. From autonomous vehicle networks and smart home infrastructures to critical cloud services and industrial IoT, system components increasingly operate independently, communicate asynchronously, and must coordinate reliably across dynamic environments. This complexity makes ensuring system correctness a central challenge in software engineering.

Because control is distributed and interactions evolve dynamically over time, conventional testing struggles to provide meaningful guarantees about the correctness or robustness of such systems. One effective way to gain confidence in the behavior of such systems is through formal verification. Particularly, model checking stands out as a widely adopted technique. Its automatic, exhaustive nature provides a strong safety net against subtle concurrency bugs. It allows for the exploration of all possible system behaviors to detect design flaws before deployment. However, state space explosion remains a major obstacle [1], [2]. As system components interact, synchronize, and their architectures become more modular, the global state space can grow exponentially, rendering classical model checking intractable, even for moderately sized modular systems.

To mitigate this, a wide array of techniques has emerged, ranging from symbolic representations [3] to partial-order reductions [4], compositional reasoning [5], and on-the-fly verification [6]. These methods, implemented in tools, such as LTSmin [7], SPIN [8], PMC-SOG [9] and NuSMV [10], have significantly improved verification performance. Still, they often struggle when applied to Modular Petri net models, where both local behavior and inter-module synchronizations must be considered. However, the increasing scale and modularity of real-world systems demand approaches that are not only scalable but also structure-aware, able to exploit the modular decomposition of the system to control complexity. Moreover, existing surveys either lack depth, focus on outdated tools, or fail to address the nuances of modular verification in distributed contexts.

In this paper, we propose a structured and up-to-date survey of the techniques and tools used to address the state space explosion in modular and distributed systems.We examine the current landscape of verification tools, identifying those that can support modularity, distributed architectures, and scalable verification. By organizing the literature by technique, tool support, and target application domain, we aim to clarify the strengths and limitations of each class of solution.

We also explore how the formalism of our Reduced Distributed State Space (RDSS) allows for localized verification by isolating module-specific properties expressed in LTL/X. This approach enables efficient model checking that leverages system modularity without requiring global state enumeration.

This paper is intended to guide researchers and practitioners seeking to understand or implement scalable formal verification strategies for distributed systems. Our goal is

to highlight the growing need for structure-aware methods, evaluate the practical impact of recent techniques, and expose gaps in tool support and methodology that merit further investigation.

The structure of this paper is as follows. Section II introduces the background and key concepts underlying formal verification, temporal logics, and Modular Petri nets. Section III analyzes the scalability problem in modular concurrency, detailing the main dimensions of state space explosion. Section IV surveys the principal verification techniques that address these challenges, including reduction methods, abstraction, compositional reasoning, symbolic approaches, and distributed exploration. Section V reviews the tooling ecosystem, highlighting representative model checkers and their capabilities for modular and concurrent systems. Section VI presents the Reduced Distributed State Space (RDSS) framework, outlining its principles, key contributions, and positioning within the literature. Section VII discusses future directions and possible extensions, including distributed implementations, hybrid strategies, and AI-driven heuristics. Finally, Section VIII concludes the paper by summarizing our findings and emphasizing the role of RDSS in advancing scalable verification.

II. Essential Concepts and Preliminaries

Modern concurrent and distributed systems are characterized by the parallel execution of multiple components that often interact asynchronously. To rigorously verify such systems, formal methods —particularly model checking — have become indispensable. This section introduces foundational concepts, including transition systems, temporal logics, Modular Petri nets, and model checking, which underpin many of the techniques discussed in this survey. The above definitions are adapted from Clarke et al. [11] and Baier and Katoen [12], which constitute the classical references in the field of model checking.

A. Labeled Transition System

Labeled Transition Systems (LTS) constitute a fundamental formalism for modeling and analyzing concurrent and distributed systems. These mathematical structures extend the basic concept of transition systems by associating explicit actions with state transitions, thereby enabling a richer representation of dynamic system behavior. LTS finds applications across numerous domains in theoretical computer science, particularly in formal verification, protocol analysis, and reactive system specification. Their capability to distinguish between observable and unobservable actions makes them particularly well-suited for modeling systems where certain internal behaviors must be abstracted from the external interface.

Definition 1: A *Labeled Transition System* (LTS) is formally expressed as a quadruple:

$$LTS = \langle \Gamma, Act, \ \rightarrow, \ I \rangle$$

where:

- Γ denotes a finite ensemble of states,
- Act designates a finite repertoire of actions,
- $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ represents the labeled transition relation,
- $I \subseteq \Gamma$ constitutes the collection of initial states.

For any transition $(s, a, s') \in \rightarrow$, the concise notation

$$s \xrightarrow{a} s'$$

indicates the existence of a passage from state s toward state s', adorned with the label a.

Within the LTS framework, one draws a sharp distinction between:

- Observable actions, symbolized as $Obs \subseteq Act$,
- Unobservable actions, symbolized as $UnObs \subseteq Act$,

such that:

$$Obs \cup UnObs = Act, \qquad Obs \cap UnObs = \varnothing.$$

This partition allows the modeling of two distinct realms: the overtly perceptible behaviors of a system, and its latent, internal evolutions, those hidden pulsations that govern its subterranean logic yet remain invisible to an external observer.

Fig. 1 illustrates LTS, with s_0 serving as the initial state. $\{a, b\}$ make up the set of observed actions. Unlabeled edges are meant to be labeled by non-observed actions.

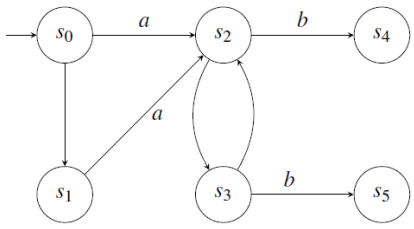

Fig. 1. Example of a labeled transition system.

B. Kripke Structure

In the domain of formal verification, Kripke Structures play a pivotal role as the semantic foundation for temporal logics such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). They extend the notion of transition systems by enriching states with atomic propositions that capture observable properties of the system. This labeling mechanism enables the evaluation of temporal formulas, thereby bridging the gap between system executions and logical specifications.

Definition 2: Let AP be a finite set of atomic propositions. A *Kripke Structure* (KS) over AP is a quadruple

$$KS \ = \ \langle \Gamma, \Gamma_0, R, L \rangle,$$

where

- Γ is a finite set of states,

- $\Gamma_0 \subseteq \Gamma$ is the set of initial states,
- $R \subseteq \Gamma \times \Gamma$ is a (total) transition relation,[1] and
- $L : \Gamma \to 2^{AP}$ is a labeling (interpretation) function that assigns to each state the set of atomic propositions true in that state, and each state of the KS is labeled with the values of these propositions.

A Kripke Structure example is shown in Fig. 2. There are two elements in the set of atomic propositions $\{a, b\}$.

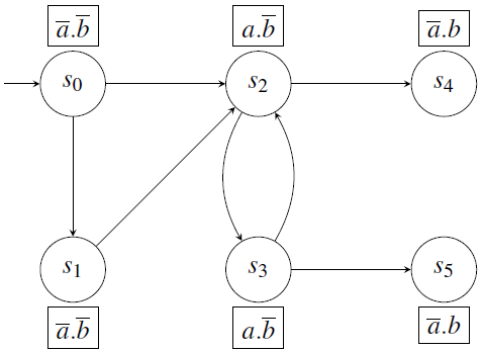

Fig. 2. Example of Kripke Structure.

Paths and reachability. A (finite or infinite) path is a sequence $p = s_0 s_1 s_2 \ldots$ with $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.

- A state $s' \in \Gamma$ is *reachable* from $s \in \Gamma$, written $s \Rightarrow s'$, iff there exists a finite path $s_0 \ldots s_k$ such that $s_0 = s$ and $s_k = s'$.
- A state s is *dead* (a sink) if there is no $s' \in \Gamma$ with $(s, s') \in R$.

Kripke Structures thus furnish the semantic substrate for model checking: a system satisfies a temporal specification precisely when all (or some, depending on the logic) paths of its Kripke Structure satisfy the corresponding formula.

C. Labeled Kripke Structure

While Kripke Structures provide the semantic foundation of temporal logics, in many verification settings it becomes necessary to make explicit the actions that drive state transitions. To address this issue, the notion of a *Labeled Kripke Structure* (LKS) has been introduced. An LKS integrates the action-labeled transitions of a Labeled Transition System (LTS) with the state-labeling of a Kripke Structure (KS). This dual enrichment bridges the gap between transition systems and Kripke semantics, yielding a versatile framework for reasoning simultaneously about state-based properties and action-driven behaviors in concurrent and distributed systems.

Definition 3: Let AP be a finite set of atomic propositions and let Act be a finite set of actions.
A *Labeled Kripke Structure* (LKS) over AP is a quintuple

$$LKS = \langle \Gamma, Act, L, R, \Gamma_0 \rangle,$$

where:

- $\langle \Gamma, Act, R, \Gamma_0 \rangle$ is a Labeled Transition System (LTS),
- $\langle \Gamma, L, R, \Gamma_0 \rangle$ is a Kripke Structure (KS).

An illustration of LKS over $AP = \{a, b\}$ is shown in Fig. 3.

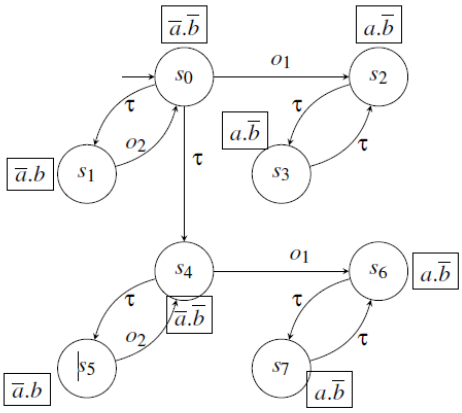

Fig. 3. Example of labeled Kripke Structure.

Remark. An LTS (resp. KS and LKS) can be represented in different ways: explicitly, where each state and arc is stored individually in memory; symbolically, where sets of states are encoded compactly using decision diagrams such as Binary Decision Diagrams (BDDs); or in a hybrid manner, with states represented symbolically while transitions are kept explicit. In addition, such models are often illustrated in an abstract form, for instance through state–transition graphs that emphasize their structural properties rather than their implementation details.

D. Modular Petri Net

Petri nets are a well-established formalism for modeling concurrent and distributed systems. However, when dealing with complex systems, a monolithic representation often becomes intractable. The notion of a Modular Petri Net (MPN) was introduced to address this limitation by decomposing a system into a finite collection of communicating modules, each being an ordinary Petri net. Communication between modules can occur in two ways: A synchronized mode through the fusion of places, or an asynchronous mode through the fusion of transitions. As shown by Christensen and Petrucci [13], any modular net with shared places can be transformed into an equivalent net with only shared transitions. For this reason, in our study we focus exclusively on modules that interact through shared transitions.

Definition 4: A **Modular Petri Net** (MPN) is a pair

$$MPN = (S, TF),$$

that satisfies the following conditions:

1) S is a finite set of **modules** where:
   a) Each module $s \in S$ is an ordinary Petri net

$$s = (P_s, T_s = T_{\text{sync},s} \cup T_{l,s}, W_s, M_{0_s}),$$

[1] Totality means $\forall s \in \Gamma,\ \exists s' \in \Gamma$ such that $(s, s') \in R$.

where, $T_{\text{sync},s}$ denotes the set of **synchronized transitions** of module s, and $T_{l,s}$ denotes the set of **local** (or **internal**) transitions of s.

b) The sets of states and transitions of distinct modules are pairwise disjoint:

$$\forall s_1, s_2 \in S,\ s_1 \neq s_2 \ \Rightarrow\ \begin{array}{l}(P_{s_1} \cup T_{s_1}) \\ \cap\,(P_{s_2} \cup T_{s_2}) = \varnothing.\end{array}$$

c) The global sets of places and transitions are given by

$$P = \bigcup_{s \in S} P_s, \qquad T = \bigcup_{s \in S} T_s.$$

2) $TF \subseteq 2^T \setminus \{\varnothing\}$ is a finite collection of non-empty **transition fusion sets**.

Fig. 4 displays a Modular Petri Net example. The three modules that make up this net are A, B, and C. Both modules A and B are in sync through transition $Sync1$, and both modules B and C are in sync through transition $Sync2$.

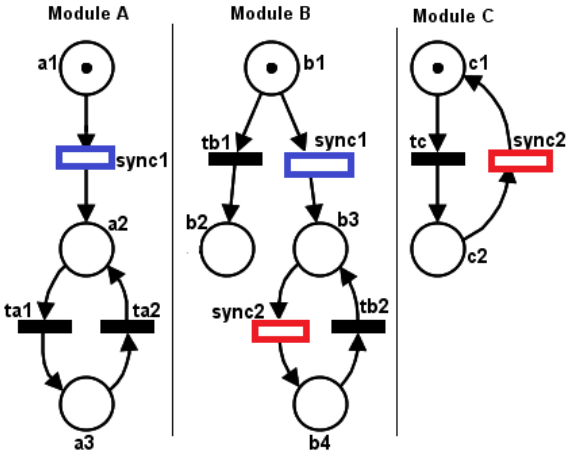

Fig. 4. Example of Modular Petri Net.

Thus, Modular Petri Nets establish a rigorous modeling framework in which complex systems can be decomposed into smaller communicating modules. By distinguishing between local and synchronized transitions, this formalism not only preserves clarity in system representation but also provides a solid foundation for subsequent works that favor modularity and compositional reasoning in the analysis and verification of distributed systems.

E. Temporal Logics: LTL and CTL

Temporal logics are central in the specification and verification of concurrent and distributed systems. They enrich propositional logic with temporal operators, making it possible to express properties not only about individual states but also about the evolution of executions over time. Among the most widely studied are Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), which differ in the way they interpret paths and quantify over possible system behaviors.

1) Linear Temporal Logic (LTL): Linear Temporal Logic (LTL) extends propositional logic with temporal operators to describe both static properties of states and the dynamic progression of system executions. It is widely used to specify properties such as safety (ensuring that undesirable events never occur) and liveness (ensuring that desired events eventually occur).

An LTL formula is built over a finite set AP of atomic propositions that capture the elementary properties of states. Boolean operators ($\neg$, $\wedge$, $\vee$) allow the expression of static conditions, while temporal operators such as X (next), U (until), R (release), F (eventually), and G (always) enable the specification of dynamic properties. Formally, an LTL formula φ is defined over AP, and its models are infinite words over 2^{AP}.

Definition 5: **(LTL Syntax)** The set of LTL formulas is defined inductively as follows:

1) Each $p \in AP$ is an LTL formula.
2) If φ and ψ are LTL formulas, then so are $\neg\varphi$, $\varphi \vee \psi$, $X\varphi$, and $\varphi\, U\, \psi$.

Additional operators are introduced as abbreviations:

$$F\varphi := \top\, U\, \varphi, \qquad G\varphi := \neg F \neg \varphi, \qquad \varphi\, R\, \psi := \neg(\neg\varphi\, U\, \neg\psi).$$

The semantics of LTL link logical formulas to system behaviors by determining whether an execution σ satisfies a formula φ.

Definition 6: **(LTL Semantics)** Let $u = u_0 u_1 u_2 \ldots$ be an infinite word with $u_i \subseteq AP$. For an LTL formula φ, the satisfaction relation $u \models \varphi$ is defined inductively:

- $u \models p$ iff $p \in u_0$,
- $u \models \neg\varphi$ iff $u \not\models \varphi$,
- $u \models \varphi \vee \psi$ iff $u \models \varphi$ or $u \models \psi$,
- $u \models X\varphi$ iff $u_1 u_2 \ldots \models \varphi$,
- $u \models \varphi\, U\, \psi$ iff there exists $k \geq 0$ such that $u_k u_{k+1} \ldots \models \psi$ and for all $0 \leq i < k$, $u_i u_{i+1} \ldots \models \varphi$.

Given a Kripke Structure $K = \langle \Gamma, L, R, s_0 \rangle$, an infinite path $\pi = s_0 \to s_1 \to \ldots$ induces the word $w = L(s_0)L(s_1)\ldots$ over 2^{AP}. We write $K \models \varphi$ if every infinite path π of K satisfies φ.

Remark. The fragment $LTL\backslash X$, which omits the "next" operator, is frequently considered in verification since it guarantees desirable properties such as *stuttering invariance*.

2) Computation Tree Logic (CTL): Computation Tree Logic (CTL) is a branching-time temporal logic, complementary to LTL, that introduces explicit path quantifiers. Whereas LTL formulas are evaluated along a single linear path, CTL formulas are interpreted over the computation tree unfolding from the initial state of a Kripke Structure. Path quantifiers A ("for all paths") and E ("there exists a path") combine with temporal operators X (next), F (eventually), G (globally), and U (until) to form operators such as AX, EF, AG, or $E[\varphi U \psi]$.

Definition 7: **(CTL Syntax)** Let AP be a finite set of atomic propositions. CTL formulas are defined inductively as follows:

1) Each $p \in AP$ is a CTL formula.
2) If φ and ψ are CTL formulas, then so are $\neg\varphi$, $\varphi \vee \psi$.
3) If φ and ψ are CTL formulas, then the following are also CTL formulas: $AX\varphi$, $EX\varphi$, $AF\varphi$, $EF\varphi$, $AG\varphi$, $EG\varphi$, $A[\varphi U \psi]$, and $E[\varphi U \psi]$.

The semantics of CTL formulas are defined with respect to states and their computation trees. Path quantifiers specify whether a property holds across all possible futures (A) or on at least one possible future (E).

Definition 8: **(CTL Semantics)** Given a Kripke Structure $K = \langle \Gamma, L, R, s_0 \rangle$, the semantics of CTL formulas are defined as follows:

- $K, s \models AX\varphi$ iff for all successors s' of s, $K, s' \models \varphi$,
- $K, s \models EX\varphi$ iff there exists a successor s' of s such that $K, s' \models \varphi$,
- $K, s \models AF\varphi$ iff along all paths starting at s, φ eventually holds,
- $K, s \models EG\varphi$ iff there exists a path from s where φ holds globally,
- and analogously for the other operators.

Remark. The essential distinction between LTL and CTL lies in their view of time: LTL formulas are interpreted along single linear paths without explicit quantifiers, while CTL relies on branching-time semantics, combining state formulas with the path quantifiers A and E (see Table I).

TABLE I. COMPARISON BETWEEN LTL AND CTL

Aspect	LTL	CTL
Time model	Linear time: formulas are interpreted along single infinite paths.	Branching time: formulas are interpreted over computation trees with multiple possible futures.
Quantification	Implicit: no explicit path quantifiers, properties are evaluated over all paths.	Explicit: path quantifiers A (all paths) and E (there exists a path) precede temporal operators.
Syntax	Operators: $\neg$, $\vee$, X, U (with derived F, G, R).	Operators: $\neg$, $\vee$, combined with AX, EX, AF, EF, AG, EG, $A[\varphi U \psi]$, $E[\varphi U \psi]$.
Semantics	Evaluated over infinite words induced by paths in a Kripke Structure.	Evaluated over states in a Kripke Structure with respect to their computation tree.
Expressiveness	Suitable for specifying properties of individual executions (e.g. safety, liveness).	Suitable for reasoning about branching choices and existence/universality of behaviors.
Verification	Model checking reduces to automata-theoretic techniques (Büchi automata).	Model checking often based on fixpoint computations over the state space.

F. Model Checking

Model checking is an automated verification technique that, given a finite-state model of a system and a formal property, determines whether the property holds in the model. In the event of a violation, the model checker generates a counterexample, i.e. an execution trace from the initial state to a state that violates the condition. This significantly enhances its practical value, as it helps designers identify and correct errors. Clarke, one of the pioneers of the field, underlined its significance [11]: "It is impossible to overestimate the importance of the counterexample feature. The counterexamples are invaluable in debugging complex systems. Some people use model checking just for this feature."

Based on state-space exploration, the model checking procedure can be described in four main phases [14] (see Fig. 5 [15]):

- State-space generation: Construct the state-space of the model M, representing all possible infinite executions of M. The generated state-space can be modeled as an ω-automaton A_M, whose language $\mathcal{L}(A_M)$ is given in the form of a Kripke Structure.
- Property translation: Transform the LTL property φ into an ω-automaton $A_{\neg\varphi}$, whose language $\mathcal{L}(A_{\neg\varphi})$ consists of all infinite executions that violate φ.
- Product construction: Build the synchronized product automaton $A_M \oplus A_{\neg\varphi}$, which accepts the language $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\varphi})$, representing all executions of M that invalidate φ.
- Emptiness check: Verify whether $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\varphi}) = \varnothing$. If the intersection is empty, the system satisfies φ. Otherwise, the product automaton accepts an infinite word that violates φ, which is then returned as a counterexample.

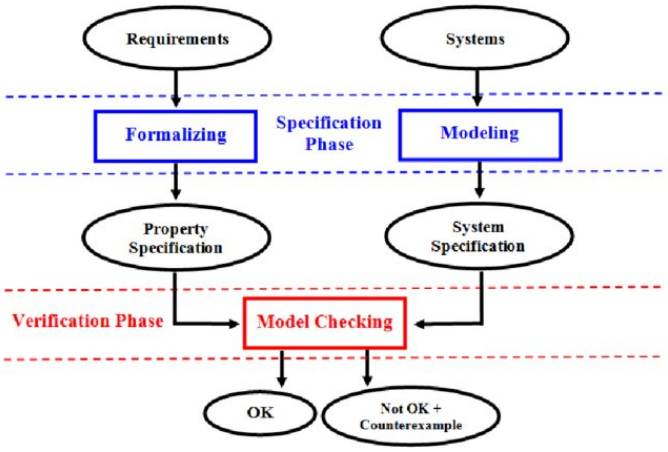

Fig. 5. Schematic view of model-checking approach.

The range of properties addressed by model checking is quite broad. For example, one can verify safety conditions, such as whether a system may reach a deadlock state, or quantitative aspects, such as whether every message will eventually be delivered with a probability of at least 0.99. In both cases, the verification is carried out automatically: the model checker systematically explores the system state space in order to detect any possible violation of the specification.

Several advantages explain the popularity of this technique. It requires little manual effort, as most of the process is

automated; it is versatile, covering a wide variety of correctness properties; and it remains comparatively efficient, even on complex designs. In addition, model checking can be applied even when only partial system specifications are available. While limitations persist, most notably the state explosion problem, the approach has proven to be one of the most practical and widely used methods in the field of formal verification.

However, the practical applicability of model checking is significantly challenged by the state-space explosion problem, which becomes particularly acute in the context of modular and concurrent systems. This limitation motivates the following discussion on scalability issues in modular concurrency.

III. The Scalability Problem in Modular Concurrency

Concurrent and distributed systems, especially when modeled in a modular fashion, pose intrinsic challenges to formal verification. Although model checking provides strong guarantees, its practical use is often hampered by the exponential growth of the state space. This phenomenon is commonly referred to as the scalability problem. In this section, we examine the fundamental dimensions of scalability in modular concurrency, highlighting how structural and semantic aspects of systems trigger combinatorial blow-ups that no straightforward verification method can overcome. These dimensions set the stage for the reduction and abstraction techniques reviewed in the subsequent sections.

A. Dimensions of the Scalability Problem in Modular Concurrency

The scalability problem in modular verification manifests along several complementary dimensions. Each of these dimensions reflects a specific source of combinatorial growth that challenges verification pipelines and limits the applicability of Naïve exploration strategies. By disentangling these dimensions, we gain a clearer view of where reduction and abstraction techniques must intervene to keep verification tractable. In the following, we outline four major dimensions that have been consistently identified in recent research: interleaving blow-up, synchronization hotspots, heterogeneous architectures, and property preservation requirements.

1) Interleaving blow-up: In modular concurrent systems, each independent action multiplies the interleaving possibilities. This leads to a combinatorial explosion of the state space. Even simple system topologies may generate billions of states. Such growth renders explicit exploration infeasible due to memory and time limitations. The issue worsens when components interleave freely, forming dense state graphs that are extremely hard to verify exhaustively.

This challenge has long been identified as a core scalability barrier in model checking [16]. Recent work shows that, even with optimized reductions, state counts can exceed 10^{12} in real-world concurrent systems [17].

2) Synchronization hotspots: Partial-Order Reduction (POR) aims to prune redundant interleavings by merging equivalent execution paths. However, when modules synchronize tightly on shared transitions, independence drops. This reduces the effectiveness of POR, increasing exploration cost. Recent complexity results show that finding near-optimal POR reductions is NP-hard. This underlines the need for hybrid strategies combining POR with abstraction or symmetry to regain scalability [18].

3) Heterogeneous architectures: Modern concurrent systems often execute across heterogeneous platforms such as multi-core CPUs, GPUs, and distributed clusters. Such platforms promise improved scalability. However, they require careful handling of memory consistency to maintain soundness and determinism across parallel or distributed memory. Mismatch in consistency models can cause nondeterministic and incorrect behaviors. Recent work highlights that enforcing cache-coherent shared memory across heterogeneous devices remains challenging, and formal verification must account for these architectural inconsistencies [19].

4) Property preservation: Effective reduction methods must guarantee preservation of critical behavioral properties, especially temporal logic specifications. This usually requires maintaining trace equivalence or stuttering equivalence, essential for logics like LTL without the next-time operator (LTL$\setminus X$). Recent formal work characterizes stuttering equivalence rigorously and reaffirms that LTL$\setminus X$ properties remain valid under such equivalence [20].

To provide a structured overview, Table II summarizes the four core dimensions that shape scalability limits in modular concurrent systems.

TABLE II. Dimensions of the Scalability Problem in Modular Concurrency

Aspect	Description
Interleaving Blow-Up	Exponential growth in interleavings due to concurrent, independent actions.
Synchronization Hotspots	Synchronous transitions undermine Partial-Order Reduction (POR); optimizing POR is NP-hard.
Heterogeneous Architectures	Multi-core, GPU, and cluster deployments challenge soundness and tool integration.
Property Preservation	Reductions must preserve trace or stuttering equivalence to ensure correctness in temporal logics (e.g. LTL\X, CTL*).

B. Real-World Failures Underscoring Scalability Challenges

The scalability issues discussed above are not only theoretical. They have had severe real-world consequences across hardware, avionics, medical systems, and cloud infrastructures.

- Intel Pentium II FDIV bug. A floating-point division error escaped formal analysis and led to costly hardware failures [21].
- Therac-25 radiation accidents. Concurrency-related defects in the control software caused lethal overdoses, revealing verification gaps in safety-critical systems [22].
- Boeing 737 MAX crashes (2018–2019). Insufficient verification of the MCAS flight-control logic under modular and concurrent interactions contributed to catastrophic accidents [23], [24].
- Autonomous vehicle accidents (Tesla, Uber, 2016–2021). Real-time decision modules misinterpreted sen-

sor data due to concurrency, leading to fatal outcomes [25].

- Spectre/Meltdown vulnerabilities (2018). These hardware flaws exposed the limits of verifying speculative execution and microarchitectural concurrency [26].
- Cloud outages (AWS, Azure, Google Cloud, 2020–2022). Complex interactions within distributed services triggered cascading failures that standard verification techniques struggled to predict [28]–[30].

These failures illustrate how state explosion, synchronization complexity, heterogeneous architectures, and property preservation remain key impediments to scalable and trustworthy verification.

IV. Landscape of Scalable Verification Techniques

A wide range of techniques has been proposed to mitigate the scalability challenges of model checking.

As shown in Fig. 6, these approaches can be organized into a taxonomy that highlights the main methodological categories, which will be detailed in the subsequent subsections.

These approaches range from algorithmic reductions that shrink the explored state space, to compositional reasoning frameworks that exploit modularity, and to strategies that harness high-performance computing platforms such as multi-core processors, GPUs, and distributed clusters. This section provides a structured overview of the methodological principles that underpin scalable verification. We highlight algorithmic strategies like state space reduction, abstraction, compositional reasoning, distributed exploration, and symbolic approaches, which form the foundation for advancing model checking in modular and concurrent systems.

A. State Space Reduction Methods

1) Partial-Order Reduction (POR): Partial-Order Reduction (POR) reduces redundant interleavings by exploiting the independence of concurrent actions. It has become one of the most effective techniques to alleviate the state explosion problem in modular and concurrent systems.

A quasi-optimal POR framework was formalized in [31], demonstrating how near-optimal reductions can drastically shrink the explored space in realistic models. Practical advances have followed, such as context-sensitive DPOR with observers, which improves scalability while preserving correctness [32]. Other recent work proposes interference-based DPOR to further prune unnecessary interleavings [33]. At the same time, theoretical results confirm the intrinsic hardness of optimizing POR, proving that even approximate solutions remain NP-hard [18].

These developments underline both the practical benefits of POR in real verification pipelines and the fundamental complexity limits that still motivate new heuristics and hybrid strategies.

2) Symmetry reduction: Symmetry reduction (SR) addresses the state explosion problem by identifying and merging states that are equivalent under permutations of identical components. For instance, in systems composed of many replicated processes, the order in which two identical processes execute is often irrelevant; exploring only one such case is sufficient.

This principle has been successfully applied in the context of Communicating Sequential Processes (CSP), a process algebra for modeling concurrent systems. Using the Failures–Divergences Refinement (FDR) model checker, groups of symmetric states are collapsed so that only a single representative is explored. This technique has enabled the verification of models with otherwise extremely large state spaces [34].

More recently, for distributed round-based protocols, partition symmetry reduction and message-order reduction dramatically shrink the model size by exploiting template symmetries [35].

Another promising direction is the use of counter abstraction, where systems with many identical components are represented by counting processes instead of enumerating each explicitly. The recent work of Eichler et al. [36] introduces a precise $(0, 1)$-counter abstraction for parameterized verification. This approach leverages process symmetry while preserving essential behavioral distinctions, thus enabling scalable verification of modular concurrent systems with large numbers of replicated components.

Together, these results illustrate not just how leveraging symmetry enables scalability in modular settings, but also how such reductions can be integrated into modern verification frameworks.

3) Stubborn sets: Stubborn sets are a state space reduction method that, like partial-order reduction, exploits the independence of concurrent actions. The idea is to identify a subset of transitions as the stubborn set that suffices to represent all relevant behaviors from a given state. By exploring only these transitions, large portions of equivalent interleavings can be safely omitted while preserving essential properties, like reachability and stuttering-invariant temporal logic formulas [LTL\X).

The core principle of this method continues to be refined and adapted to new and complex verification scenarios. For example, it has been extended to handle two-player reachability games, providing efficient reduction in adversarial settings where the outcome depends on the choices of multiple agents [37].

Furthermore, the versatility of stubborn sets is demonstrated by their application beyond standard Petri nets; they have been successfully integrated into frameworks for verifying Time Petri Nets, helping to mitigate state explosion in real-time systems by reducing redundant interleavings of temporal actions [38].

These developments underscore that stubborn sets remain a fundamental and adaptable technique for tackling state space explosion in diverse types of concurrent systems.

B. Abstract State Space Representations

Abstraction is a fundamental method to tackle the state explosion problem by collapsing sets of concrete states into

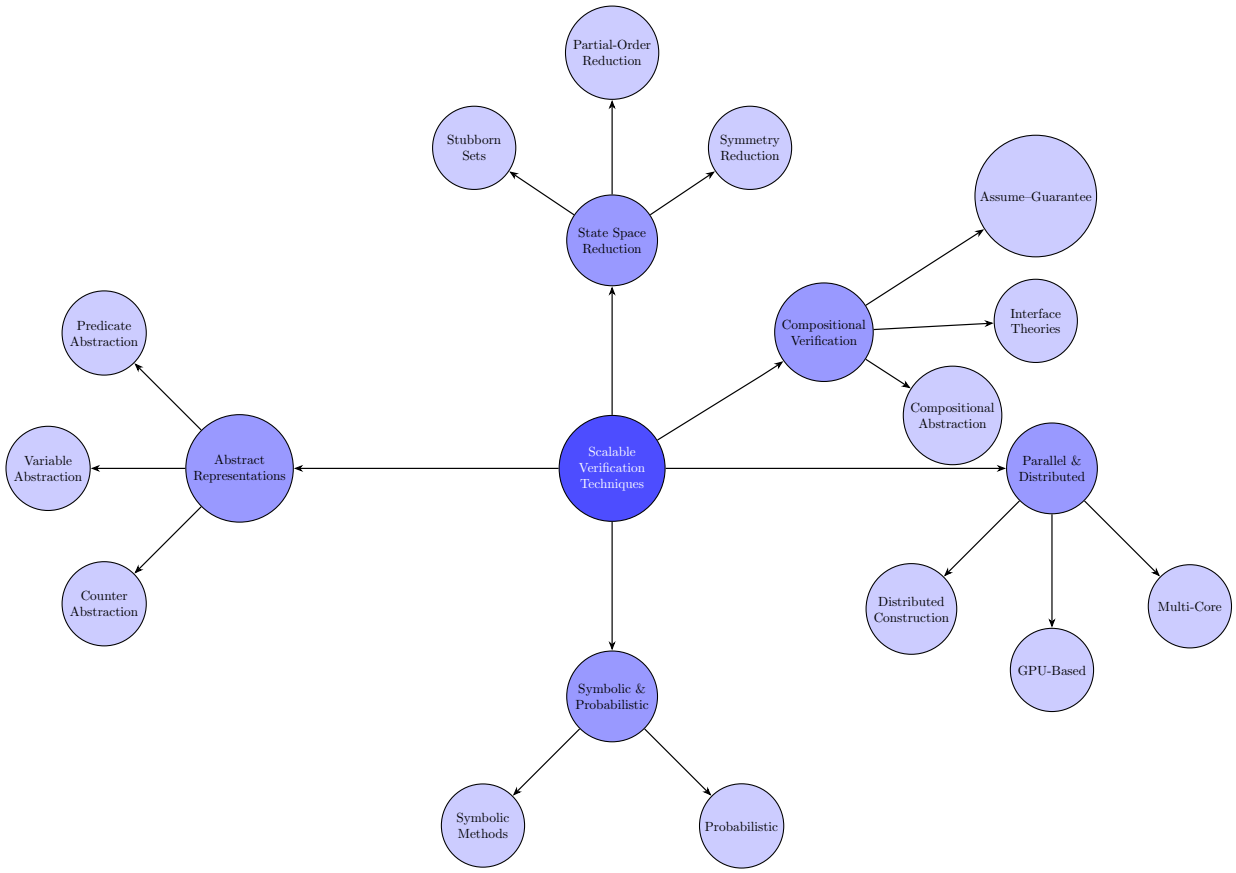

Fig. 6. Overview of scalable verification techniques, categorized into state space reduction, abstraction, compositional reasoning, parallel and distributed approaches, and symbolic/probabilistic methods.

abstract representatives. In contrast to reductions such as POR or symmetry, which preserve the concrete structure but prune interleavings, abstraction constructs a higher-level model that approximates the original system. Three categories of abstraction are commonly distinguished in the literature:

1) Predicate abstraction: Predicate abstraction replaces concrete data domains with Boolean predicates that capture properties relevant to the verification task. This technique underpins the well-known CEGAR (Counterexample-Guided Abstraction Refinement) framework, where spurious counterexamples trigger automatic refinement. Recent work applies predicate abstraction to real-time multi-agent systems, producing finite abstract models that remain correct for temporal logics, such as TCTL [39].

2) Data / Variable abstraction: Variable abstraction reduces large or infinite data domains into coarser partitions such as intervals, symbolic ranges, or sets. This approach is particularly useful in systems with timing or quantitative data, where abstraction preserves the control structure but simplifies the analysis. Examples include abstractions used for timed automata and hybrid systems, where compact representations allow tractable model checking [40].

3) Counter abstraction: Counter abstraction (also known as symmetry-based abstraction) aggregates identical processes by recording how many are in each local state, rather than tracking each process individually. This method has been widely used in the parameterized verification of concurrent protocols, significantly reducing the state space of replicated-component systems. Recent advances refine counter abstraction with precise $(0, 1)$-counters for parameterized systems, ensuring scalability without losing critical behaviors [36].

These three categories form the canonical core of abstraction techniques in model checking, offering scalable verification through higher-level representations of system behavior.

C. Compositional Verification

Compositional verification is a fundamental strategy to address state explosion by leveraging the modular structure of a system. Instead of verifying the system monolithically, this approach decomposes the verification task by analyzing individual components in isolation. The results of these local analyses are then composed to infer global properties of the entire system. The core challenge lies in managing the interactions between components, typically through the use of assumptions, contracts, or abstract interfaces.

1) Assume-guarantee reasoning: Assume-Guarantee Reasoning (AGR) is a cornerstone compositional technique that verifies systems by analyzing their parts separately. Instead of building a single large model for the entire system, AGR checks each component under specific assumptions about how

others will behave. The core idea is captured by a simple rule: a component C guarantees a property G provided its environment meets an assumption A, written as $\langle A \rangle C \langle G \rangle$.

To verify a system made of two components, suitable assumptions must be found so that each component's guarantee holds when the other acts as assumed. This avoids constructing the global state space altogether. The main challenge is automatically discovering these assumptions.

Recent advances are expanding the scope and expressiveness of AGR along two key dimensions: by developing novel logical frameworks to specify complex compositional properties, such as using epistemic logic to reason about knowledge in concurrent systems [5], and by creating powerful symbolic techniques that apply this reasoning to verify intricate systems, such as multi-language security protocols [27].

2) Interface theories: Interface Theories provide a formal foundation for modeling component interactions through contracts. A contract explicitly defines the assumptions a component makes on its environment and the guarantees it provides under those conditions. This formalism enables rigorous compositional reasoning by ensuring that each component's guarantees satisfy the assumptions of others in the system, thereby guaranteeing overall correctness [41].

Contract-Based Design (CBD) elevates this concept to a systems engineering methodology. It promotes modularity, reuse, and incremental development by allowing components to be specified, verified, and refined independently based on their contracts. This approach is particularly effective for large-scale concurrent systems, as it localizes verification efforts and simplifies integration. Recent research has focused on overcoming the practical challenge of manual contract specification by developing methods for the automated composition and refinement of contracts, making the methodology more scalable and accessible for complex system-on-chip designs [42]. The practical impact of CBD is significant, with successful applications spanning critical domains such as cyber-physical systems (CPS), where it helps manage the complexity of interactions between computational and physical elements [43]. This also extends to the automotive and aerospace industries.

3) Compositional abstraction and minimization: This technique involves generating a simplified, abstract model for each component that is conservative with respect to the properties of interest. The key idea is to reduce the state space of individual components before composition, making the subsequent global analysis more tractable. The abstraction must be precise enough to preserve relevant behaviors when components interact.

Methods range from traditional abstraction refinement (CEGAR) applied compositionally to techniques that exploit the modular structure to minimize each component while preserving equivalence (e.g. bisimulation minimization).

A recent advancement in this area focuses on compositional model checking for multi-properties, which aims to verify multiple system properties simultaneously and efficiently by leveraging the compositional structure [44]. The final verification step is performed on the composition of the abstract models, which is typically much smaller than the composition of the original components.

D. Distributed and Parallel Verification

The exponential growth in state space size often exceeds the capabilities of sequential verification, necessitating approaches that leverage parallel and distributed computing architectures. This subsection explores three key strategies that exploit modern hardware to distribute the computational and memory load of verification. These techniques, which target multi-core CPUs, GPUs, and distributed clusters respectively, form an essential toolkit for scaling verification to extremely large and complex modular systems.

1) Multi-core verification: A key strategy to combat state explosion leverages the parallel processing power of multi-core CPUs. This involves designing model checking algorithms and data structures that can efficiently distribute the workload across multiple cores, aiming for near-linear speedups. The challenge lies not only in parallelizing state space exploration but also in ensuring that the underlying data structures scale without becoming a bottleneck due to memory contention or fragmentation.

Recent innovations address this core challenge head-on. The EDDY framework exemplifies this progress, introducing a novel multi-core Binary Decision Diagram (BDD) package specifically engineered for modern shared-memory architectures [45]. Its main contributions include advanced dynamic memory management and a fragmentation-reducing allocator, which together mitigate critical performance overheads that plague traditional implementations. By optimizing memory access patterns and locality, EDDY enables more efficient parallel symbolic verification, directly increasing the scale of systems that can be checked. This focus on low-level memory optimization complements higher-level concurrency strategies, such as work stealing and lock-free data structures, forming a robust and scalable foundation for multi-core model checking.

a) GPU-based model checking: The massive parallelism offered by GPUs is exploited to accelerate demanding verification tasks, particularly for complex system models. This approach moves beyond traditional CPU-bound algorithms to harness the thousands of cores available on a GPU for parallel state space exploration and analysis.

GPU acceleration effectively scales Statistical Model Checking (SMC), a key method for verifying stochastic and timed systems. For instance, executing simulations of Extended Timed Automata on the GPU significantly accelerates the verification of temporal and probabilistic properties [46]. This approach efficiently tackles state space explosion in complex real-time models.

2) Distributed state space construction: Distributed verification decomposes the global state space across a network of machines, each responsible for exploring a distinct partition. This approach directly addresses the memory limitations of a single node, enabling the analysis of exceptionally large systems. The performance and scalability of this strategy critically depend on the underlying network technology and the efficiency of state management.

Cutting-edge research now leverages high-speed RDMA networks to minimize communication overhead. For instance, recent algorithms combine distributed exploration with sophisticated state reconstruction techniques, dramatically reducing

the amount of data that needs to be transmitted between nodes. An evaluation of such an approach shows significant improvements in scalability and efficiency for explicit state model checking, demonstrating the potential of modern networking technologies for distributed verification [47]. This line of work is directly applicable to large-scale modular systems, including those using RDSS-style construction.

E. Symbolic and Probabilistic Verification

While the previous techniques often deal with the state space explicitly or by exploiting its structure, another family of approaches uses fundamentally different representations. This part covers two such powerful paradigms: symbolic verification, which uses compact encodings to represent vast state spaces implicitly, and probabilistic verification, which extends classical model checking to reason about stochastic behaviors and quantitative properties. The integration of symbolic and probabilistic methods opens new frontiers for scalable verification.

1) Symbolic methods: Symbolic model checking techniques avoid explicit state enumeration by using compact logical representations. These methods employ data structures such as Binary Decision Diagrams (BDDs) and satisfiability solvers (SAT/SMT) to encode state sets and transition relations as formulas. This approach enables efficient property checking by exploring large state spaces symbolically rather than explicitly.

The practical effectiveness of these techniques depends fundamentally on the performance of their underlying reasoning engines. Modern SAT/SMT solvers have demonstrated remarkable capability in solving complex constraint problems across various domains. As shown in recent evaluations of large-scale puzzle solving, these solvers can efficiently handle problems with millions of constraints and variables, showcasing their scalability and robustness [48]. In formal verification, this computational power enables the analysis of complex system properties by reducing verification tasks to satisfiability problems, making symbolic methods essential for handling large-scale industrial systems.

2) Probabilistic verification: Probabilistic model checking extends classical verification to systems that exhibit random or unpredictable behavior. This approach analyzes quantitative properties of models such as Markov chains, which describe systems evolving probabilistically over time. Unlike traditional verification that provides yes/no answers, probabilistic methods compute likelihoods, for example, determining the probability that a communication protocol successfully delivers a message or calculating the expected energy consumption of a distributed algorithm.

The theoretical foundation for efficiently analyzing these models relies heavily on notions of behavioral equivalence. Recent advances have focused on developing a spectrum of approximate probabilistic bisimulations, which provide a formal framework for quantifying the similarity between states in probabilistic systems [49]. This allows for the construction of simplified models that preserve quantitative properties within precise error bounds, enabling more efficient analysis.

To consolidate this discussion, Table III provides a structured summary of the main categories of scalable verification techniques, their key methodological principles, typical domains of application, and their suitability for modular concurrency.

Remark. Some references appear in multiple sections of this survey. This reflects their dual relevance: for instance, a contribution may introduce a novel reduction technique while also providing its implementation in a widely used tool. In such cases, the same source is cited in both the methodological and tooling contexts to accurately represent its impact across the landscape.

V. Tooling Ecosystem: Capabilities for Modular Systems

In parallel with theoretical advances, a rich ecosystem of verification tools has been developed. These platforms implement the reduction, abstraction, and compositional techniques surveyed in the previous section, thus bridging academic research with industrial applications. This section reviews the most influential verification tools and evaluates their relevance for modular and concurrent systems.

A. Classical Explicit-State Tools

Classical explicit-state tools represent the first generation of practical model checkers. They operate by directly constructing and exploring the state space. They are often supported by sophisticated search strategies and reduction techniques to manage the explosion of states. Although more recent symbolic and distributed approaches have emerged, these tools remain highly influential. They are still actively used in both academia and industry. The following subsections briefly review three representative explicit-state platforms.

1) SPIN: The SPIN model checker [8] remains one of the most widely used explicit-state verification engines. It targets systems specified in the Promela modeling language and implements techniques such as partial-order reduction and bitstate hashing. SPIN has been applied to a broad spectrum of communication protocols and distributed algorithms, demonstrating scalability up to millions of states. While not natively modular, its support for asynchronous processes makes it relevant for analyzing concurrent behaviors in modular settings.

2) NuSMV: NuSMV [10] is a symbolic model checker supporting both CTL and LTL properties. It implements BDD-based symbolic methods and SAT-based bounded model checking, providing a balance between explicit and symbolic exploration. Although its architecture is not specifically modular, NuSMV has served as a foundation for contract-based verification frameworks, notably in embedded and safety-critical software.

3) UPPAAL: UPPAAL [50], [51] focuses on timed automata and real-time systems. It integrates symbolic techniques for handling clock variables, making it particularly suited for modular cyber-physical systems. Compositional extensions of UPPAAL support verification of large networks of timed components, where timing constraints play a central role.

B. High-Performance Verification Frameworks

High-performance frameworks emerged to overcome the scalability limitations of classical explicit-state tools. These

TABLE III. COMPARATIVE SUMMARY OF SCALABLE VERIFICATION TECHNIQUES, THEIR DOMAINS, AND SUITABILITY FOR MODULAR CONCURRENCY

Category	Key Methods / References	Domain of Application	Suitability for Modular Concurrency
State Space Reduction	POR [18], [31]–[33], Stubborn Sets [37], [38], Symmetry [35], [36]	Protocols, Hardware verification, Petri nets	High — directly addresses interleaving and replication in concurrent modules
Abstract Representations	Predicate [39], Variable/Data [40], Counter [36]	Hybrid systems, Real-time models, Parameterized protocols	Medium to High — effective when abstraction is precise, but risk of spurious behaviors
Compositional Verification	AGR [5], [27], Interface Theories [41]–[43], Compositional Abstraction [44]	Large-scale software, Cyber-physical systems, Security protocols	High — inherently modular, though assumption discovery and contract design remain challenging
Parallel & Distributed, Symbolic & Probabilistic Verification	Multi-core [45], GPU [46], Distributed [47], Symbolic/Probabilistic [48], [49]	Industrial-scale models, Stochastic protocols, Cloud infrastructures	Medium to High — excellent for scaling exploration, less natural for modular decomposition

platforms exploit optimized data structures, parallel architectures, and language-independent interfaces to accelerate state space exploration. They provide the backbone for verifying larger and more complex models, while often serving as integration points for new reduction techniques and symbolic back-ends. The following subsections review representative frameworks in this category.

1) LTSmin: LTSmin [52] is a high-performance language-independent model checker that provides interfaces to multiple modeling languages (e.g. Promela, DVE, PNML). It integrates symbolic techniques (decision diagrams), parallel reachability, and partial-order reductions. Its modular architecture and native support for Petri nets make it highly relevant for modular concurrency.

2) DiVinE: DiVinE [53] is a distributed explicit-state model checker designed for large-scale verification on clusters and high-performance computing platforms. It implements parallel reachability, on-the-fly LTL model checking, and advanced partial-order reductions. DiVinE demonstrates strong scalability for concurrent modular models distributed across thousands of cores.

3) EDDY (multi-core BDD): The EDDY framework [45] provides a multi-core BDD package optimized for shared-memory architectures. By redesigning memory allocators and dynamic variable reordering, EDDY improves symbolic model checking on multi-core CPUs. This directly addresses the need for efficient symbolic verification in modular systems with large state spaces.

C. Probabilistic and Symbolic Engines

Another important branch of verification frameworks focuses on symbolic and probabilistic approaches. Instead of relying solely on explicit state enumeration, these engines use compact logical representations or stochastic models to capture system behaviors. Symbolic methods such as BDD- or SAT/SMT-based encodings enable implicit exploration of vast state spaces, while probabilistic model checkers extend classical verification to reason about uncertainty and quantitative properties. Together, these tools greatly expand the range of systems and properties that can be analyzed. The following subsections highlight representative engines in this category.

1) STORM: STORM is a modern probabilistic model checker that integrates symbolic, explicit, and numerical methods into a unified framework. It supports a wide range of probabilistic models, including Markov decision processes (MDPs), continuous-time Markov chains (CTMCs), and stochastic Petri nets, making it applicable to both academic benchmarks and industrial-scale case studies. Unlike earlier tools that specialized in either symbolic or explicit techniques, STORM provides a flexible architecture that combines multiple engines to balance scalability and precision. Recent evaluations confirm its ability to analyze large and complex quantitative models while maintaining soundness and efficiency [54]. This makes STORM particularly relevant for modular and concurrent systems where stochastic behaviors interact with structural concurrency.

2) PRISM: PRISM [55] pioneered probabilistic model checking and remains a widely used tool for analyzing stochastic protocols and cyber-physical systems. It supports a wide range of probabilistic logics and has influenced the design of STORM. While not primarily modular, its modeling language facilitates component-based specifications.

3) SAT/SMT-based frameworks: Modern SAT/SMT-based model checkers leverage the efficiency of constraint solvers to perform bounded or unbounded verification. Tools such as CBMC and IC3-based solvers integrate abstraction-refinement and compositional reasoning, making them suitable for modular concurrent software verification. Recent advances demonstrate their scalability for verifying security protocols and hardware blocks.

D. Domain-Specific and Hybrid Tools

Beyond general-purpose model checkers, a variety of domain-specific and hybrid tools have been developed to address the needs of particular application areas. These platforms often combine multiple verification paradigms (explicit, symbolic, probabilistic) and tailor them to specialized modeling languages or industrial domains. They play a crucial role in bridging the gap between theoretical advances and real-world deployments, offering verification capabilities that are more directly aligned with practitioner requirements. The following parts survey representative tools in this category.

1) PAT (Process analysis toolkit): PAT [56] supports verification of process-algebraic models, combining CSP-style semantics with state space reductions and refinement checking. It provides explicit support for modular specifications and has been applied to distributed protocols and workflow systems.

2) SDV (Static driver verifier): Microsoft's Static Driver Verifier (SDV) integrates symbolic model checking with static analysis to automatically verify Windows device drivers [57]. It

builds on the SLAM project and uses predicate abstraction with refinement. Although domain-specific, SDV exemplifies how verification pipelines can scale to real-world modular software components.

3) Recent hybrid platforms (GPU, cloud): Recent tools leverage GPUs and cloud infrastructures to accelerate verification. Notably, distributed model checkers such as DiVinE-cloud and GPU-accelerated statistical model checking frameworks demonstrate promising results [46], [47]. These platforms highlight the emerging trend of integrating heterogeneous computing into verification workflows.

4) SPORE: SPORE [58] is a recent stateless model checker that combines partial-order reduction with symmetry reduction. Its hybrid algorithm significantly reduces explored interleavings, demonstrating orders-of-magnitude improvements over classical POR. SPORE directly targets concurrency and is particularly relevant for modular distributed protocols. Although some of these tools were introduced decades ago, they continue to be actively maintained and widely applied in recent studies.

Table IV summarizes their core verification techniques, supported models, relevance for modular/concurrent systems, and illustrates their continued impact with recent references and application domains.

While many of these tools were originally introduced more than a decade ago, their sustained use in recent research highlights the robustness of their foundational techniques. As shown in Table IV, extensions such as GPU acceleration, symbolic encodings, and domain-specific adaptations (e.g. robotics, IoT, autonomous driving) ensure that these platforms remain relevant for contemporary verification challenges. This consolidated landscape provides the backdrop against which we now situate our own contribution, focusing on the Reduced Distributed State Space (RDSS) framework and its role in addressing scalability in modular concurrent systems.

VI. RDSS: Contribution, Positioning, and Future Directions

Building upon the surveyed techniques and tools, this section introduces the Reduced Distributed State Space (RDSS) framework as a novel approach to scalable verification of modular and concurrent systems. RDSS directly addresses the limitations identified in existing methods by providing a modular structure that integrates reduction principles while enabling verification at the level of individual modules. We first outline the gaps in current approaches, then present the RDSS framework and its key contributions, followed by a discussion of its positioning within the literature and possible future extensions.

A. Scope and Claims

Before detailing our framework, we explicitly clarify its scope and novelty. The novel contribution of RDSS lies in its ability to perform local model checking directly on per-module meta-graphs, rather than requiring construction of the entire global state space. This is achieved through two key mechanisms: (*i*) τ-hiding of irrelevant synchronizations, and (*ii*) meta-state fusion that preserves *stuttering equivalence*. These innovations enable verification of module-specific properties with strong theoretical guarantees. RDSS does not reinvent prior reduction strategies; instead, it *integrates* and benefits from established methods such as partial-order reduction, symmetry reduction, and abstraction, while remaining compatible with distributed exploration. Finally, the property classes supported are those expressible in the stutter-invariant fragment of LTL (LTL $\setminus$ $\mathbf{X}$), including safety, liveness, and local properties such as deadlock-freedom, home-space reachability, and module-level liveness. By stating these boundaries, we aim to separate our claims from the survey of existing work and set clear expectations for the reader.

B. Identified Gaps in Existing Approaches

Despite the advancements in model checking and state space reduction methods, current verification techniques for modular and concurrent systems face significant scalability challenges. Traditional approaches, such as POR, symmetry reduction, and abstraction, address specific aspects of the state space explosion problem. Unfortunately, they do not provide comprehensive solutions for the complex nature of modern systems.

First, POR helps reduce redundant interleavings by exploiting the independence of actions. However, its effectiveness is limited when the system's concurrency is highly interdependent or exhibits complex synchronization. Symmetry reduction identifies equivalent states that result from symmetric components. Nevertheless, it fails to address cases where the symmetry is less evident or dynamic. Similarly, abstraction techniques can collapse large portions of the state space. However, they risk oversimplifying behaviors, potentially missing critical verification information. This is particularly problematic when applied to modular systems with complex interactions between components.

Moreover, these techniques, while effective individually, fail to handle the full scope of challenges posed by highly dynamic and modular systems, where components can interact in unpredictable ways. These limitations motivate the need for more integrated and scalable approaches, such as Reduced Distributed State Space (RDSS), which tackles the scalability problem in a modular and distributed context.

C. The RDSS Framework

The RDSS framework introduces a novel and scalable solution to the state explosion problem in modular and concurrent systems. It builds upon the distributed state space structure originally proposed in [80], extending it with formal guarantees such as stuttering equivalence, τ-hiding of irrelevant synchronizations, and fusion of equivalent meta-states. This part first presents the core principles of RDSS, followed by its key contributions to scalable verification.

1) The RDSS principle: The RDSS framework addresses the challenge of state explosion in modular systems by reducing the global state space into smaller, manageable meta-graphs formed of meta-states that represent the state space of individual modules. The input of the RDSS consists of the concurrent modular system specification being studied, formalized in a Modular Petri Net structure. Each module in the system constructs its local meta-graph, capturing only the relevant transitions based on local and synchronized events.

TABLE IV. SUMMARY OF REPRESENTATIVE VERIFICATION TOOLS, THEIR CORE TECHNIQUES, SUPPORTED MODELS, RELEVANCE FOR MODULAR/CONCURRENT SYSTEMS, AND EVIDENCE OF CONTINUED USE IN RECENT DOMAINS

Tool	Core Techniques	Model Supported	Relevance to Modular/Concurrent Systems	Recent Use (Refs.)	Application Domain (Recent)
SPIN	Explicit-state, POR, Bitstate hashing	Promela processes	Asynchronous processes, distributed protocols	[59], [60]	IoT-based transport, abstracted IoT protocols
NuSMV	Symbolic (BDD, SAT/SMT), Bounded MC	SMV language, CTL/LTL	Contract-based reasoning in modular embedded systems	[61], [62]	Multi-agent systems, software product lines
UPPAAL	Symbolic (timed automata, DBMs)	Networks of Timed Automata	Real-time CPS and IoT verification	[63], [64]	Drone access control, contract automata runtime
LTSmin	Symbolic DDs, POR, Distributed exploration	PNML, Promela, mCRL2	Modular Petri Nets and parameterized models	[65], [66]	GPU-based LTL MC, knowledge-based reductions
DiVinE	Explicit distributed MC, Parallel LTL checking	DVE, PNML, Promela	Cluster-based concurrency analysis	[67], [68]	C++ program verification, GPU POR
EDDY	Multi-core symbolic BDDs	Symbolic encodings (BDD-based)	Shared-memory multi-core verification	[69], [70]	HPC memory optimization, GPU model training
STORM	Symbolic + numerical probabilistic engines	MDPs, CTMCs, Stochastic PN	Quantitative verification of stochastic modular systems	[71], [72]	Robotic systems, MDP algorithm selection
PRISM	Symbolic (BDD, MTBDD), Probabilistic checking	Probabilistic automata, MDP, CTMC	Stochastic CPS, security protocols	[73], [74]	Symmetry reduction, model-based frameworks
SPORE	Stateless MC, POR + Symmetry reduction	Concurrent C programs, message-passing protocols	Concurrency, modular distributed protocols	[58]	Benchmarks of concurrency programs
PAT	Explicit + refinement checking, POR	CSP-style models, workflows	Modular process-based verification	[75], [76]	Federated learning, blockchain smart contracts
SDV	Predicate abstraction + CEGAR (SLAM)	C code (Windows drivers)	Modular verification of device drivers	[77], [78]	Distributed BMC, autonomous driving (ADS)

The interactions between modules in RDSS are captured through synchronization points, which align the meta-graphs of different modules during the construction process. This modular approach avoids the need to explore redundant state space information from non-relevant modules.

A key concept in RDSS is the use of τ transitions, which are used to hide irrelevant synchronizations between modules. These transitions allow the system to focus on the core behaviors, ignoring unnecessary synchronization events that do not affect the properties being verified. During the construction of the RDSS, equivalent meta-states are identified and merged to reduce the overall state space. This process ensures that only unique behaviors are captured, and redundant states that exhibit identical or analogous transitions are eliminated.

This fusion is based on the principle that multiple states, which share the same output behavior and synchronization properties, can be represented by a single meta-state. By merging these equivalent meta-states, RDSS maintains the correctness of the verification while significantly reducing the number of states that need to be explored [81].

The local meta-graphs are then used for verification, enabling the system to scale by performing module-specific verification without requiring access to the entire global state space [79].

Furthermore, RDSS ensures the stuttering equivalence of the reduced state space. It has been formally proven that the projection of the language generated by the graph of a module on its own transitions is the same as the language generated by the entire system's global state space. This guarantees that no critical behaviors are lost during the reduction process.

Fig. 7 demonstrates the modular decomposition and the use of synchronization points in the RDSS framework.

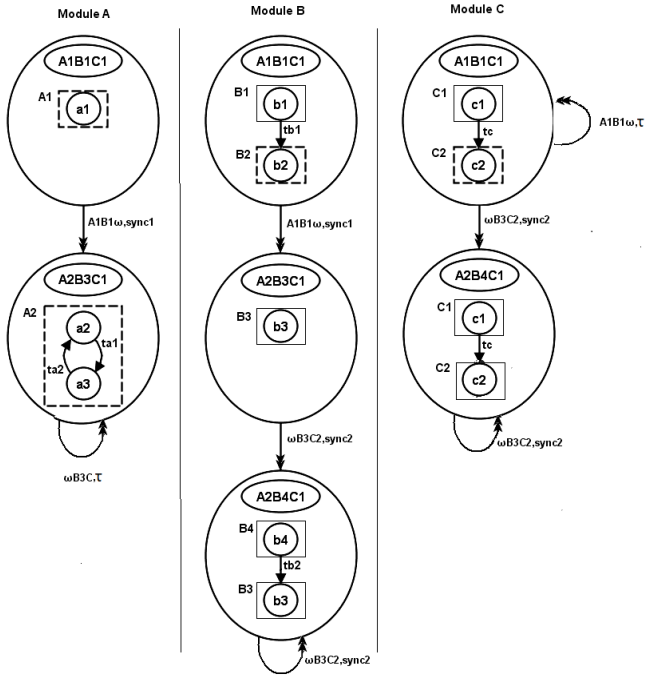

Fig. 7. Illustration of RDSS corresponding to the Modular Petri Net of Fig. 4.

D. Soundness of Local Model Checking in RDSS

A central question for the Reduced Distributed State Space (RDSS) framework is whether properties verified locally on module-level meta-graphs are sound with respect to the original Modular Petri net (MPnet). This subsection establishes the formal basis of this claim by proving that RDSS preserves stuttering equivalence with the original system, and that the projection of behaviors on a given module coincides in both

structures. As a result, $LTL \setminus \mathbf{X}$ properties can be checked locally without loss of correctness.

1) Projection argument: Let Σ be the language generated by the MPnet $\mathcal{P}$ over its set of transitions, and $\Sigma_{/s}$ the projection of Σ onto the transitions specific to module s. Let Σ_s be the language generated by the meta-graph of module s, and $\Sigma_{s/s}$ its projection onto the transitions of s. We prove that $\Sigma_{/s} = \Sigma_{s/s}$.

Indeed, any infinite execution sequence $\sigma \in \Sigma$ can be written as

$$\sigma = \left((t_l)^*(t_{sync})^*\tau^*\right)^*,$$

where, t_l are local transitions, t_{sync} are synchronized transitions, and τ denotes hidden synchronizations. Projecting σ onto module s eliminates irrelevant synchronizations, yielding

$$\sigma^{\Sigma_{/s}} = \left((t_{l,s})^*(t_{sync,s})^*\right)^* \in \Sigma_s.$$

Hence $\Sigma_{/s} \subseteq \Sigma_{s/s}$.

Conversely, every execution $\sigma \in \Sigma_s$ is built from the local, synchronized, and τ transitions of module s, so

$$\sigma = \left((t_{l,s})^*(t_{sync,s})^*\tau^*\right)^*.$$

Projecting σ on s removes the τ transitions, giving

$$\sigma^{\Sigma_{s/s}} = \left((t_{l,s})^*(t_{sync,s})^*\right)^* \in \Sigma_{/s}.$$

Thus $\Sigma_{s/s} \subseteq \Sigma_{/s}$. Combining both inclusions yields:

$$\Sigma_{/s} = \Sigma_{s/s}.$$

2) Stuttering equivalence: From the above projection result, we conclude that the maximal paths of the MPnet and those of the RDSS coincide modulo τ-hiding. This implies that the two structures are stuttering-equivalent, and therefore preserve all properties expressible in the stutter-invariant fragment $LTL \setminus \mathbf{X}$.

Theorem 1 (Stuttering Equivalence for RDSS): Let $\mathcal{P}$ be an MPnet with $\mathcal{P} = (S, TF)$, and let $\mathcal{R}$ be its corresponding RDSS defined as

$$\mathcal{R} = \{\, RG_s = (\widehat{\mathcal{N}}_s, \widehat{\mathcal{A}}_s) \mid s \in S \,\}.$$

For any $LTL \setminus \mathbf{X}$ formula φ defined over T_s, the set of local and synchronized transitions of a module s, we have:

$$\mathcal{P} \vDash \varphi \iff \mathcal{R} \vDash \varphi.$$

3) Proof sketch: The projection argument shows that every path of $\mathcal{P}$ restricted to module s corresponds to a path in $\mathcal{R}$ and vice versa. Since $LTL \setminus \mathbf{X}$ properties are preserved under stuttering equivalence, verifying φ on the module-level meta-graph is sound and complete with respect to verifying it on the global reachability graph. Detailed proofs and demonstrations are available in our GitHub repository[2].

4) Key contributions of RDSS: The RDSS framework offers several groundbreaking contributions to the field of scalable verification for modular concurrent systems:

5) Verification of individual modules: The core contribution of RDSS is its ability to verify properties of individual modules using only the meta-graph of that specific module. Unlike traditional methods that require access to the global state space, RDSS is a pioneering approach that enables local model checking. This approach drastically improves scalability, as it allows the modules of large systems to be verified efficiently. The benefits of this modular verification strategy are further illustrated in Fig. 8, which compares the verification times of RDSS with those of LTSmin. The results show that RDSS achieves competitive or even superior performance, particularly in systems where modules are not all synchronized on the same transitions.

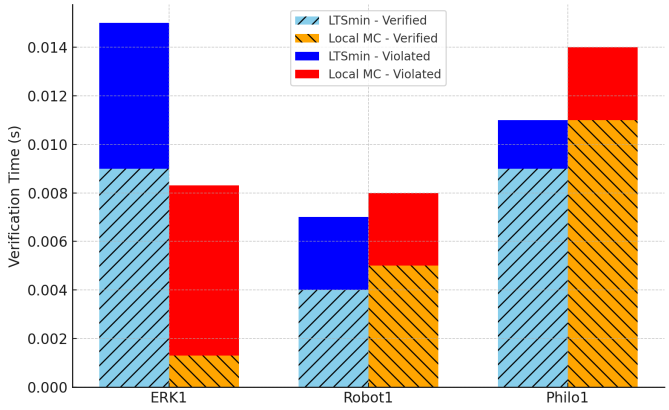

Fig. 8. Comparative performance of LTSmin and Local Model Checking in verifying modular systems.

6) Integration of state space reduction techniques: Another significant contribution of RDSS is its integration of various state space reduction techniques into a modular framework. Methods such as partial-order reduction (POR), symmetry reduction, and abstraction are applied both at the global and local levels. By combining these reductions, RDSS is a hybrid method that optimizes the verification process, ensuring the correctness of global properties while managing the complexity of individual modules.

7) Flexibility in modular verification: RDSS introduces a new level of flexibility in the verification of modular systems, particularly for systems with dynamic interactions. Unlike traditional verification methods that may struggle with complex interdependencies between modules, RDSS can efficiently handle systems where interactions evolve, leveraging its module-specific meta-graphs. Moreover, the modular structure

[2]https://github.com/chihebabid/DSS-Checker

of RDSS is well-suited for distributed concurrent systems, as it allows for efficient verification by focusing on individual modules and their interactions, rather than constructing the entire global state space.

8) State space size reduction: When compared with the flat Petri net state space, RDSS delivers better results, particularly for systems where not all modules are synchronized on the same transitions. This reduction in state space is particularly beneficial for verifying large-scale modular systems, as shown in Fig. 9, which illustrates the performance improvement of RDSS over traditional methods. The comparison between the flat Petri net and RDSS in terms of transitions and markings demonstrates the scalability advantages of the RDSS approach.

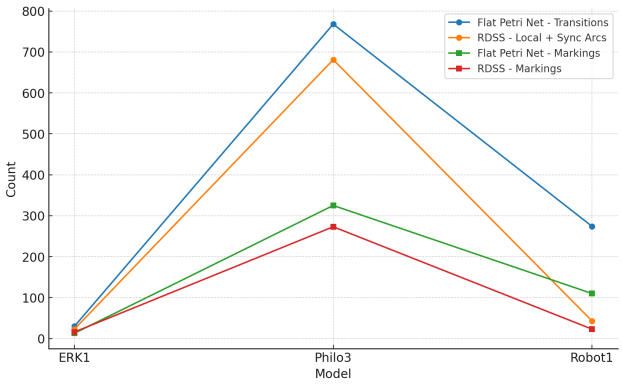

Fig. 9. Comparison of Flat Petri Net and RDSS. The figure illustrates the performance improvement of RDSS in terms of state space size reduction, particularly in systems with unsynchronized modules.

9) Model checking focus: A key strength of RDSS lies in its model checking capabilities. By focusing on module-level verification, RDSS allows for the verification of properties such as deadlock, liveness, and reachability directly within each module, using its associated meta-graph. This localized model checking significantly reduces the computational burden compared to traditional approaches that require the construction and exploration of the entire global state space.

Fig. 10 illustrates the RDSS model checking process, highlighting how the meta-graph of each module is used for verification. This diagram emphasizes the modular decomposition of the verification process, showcasing the RDSS's ability to verify individual components without needing to consider the full system's state space.

10) Worked example: To illustrate the process, consider the Modular Petri Net (MPN) in Fig. 4 and its corresponding RDSS in Fig. 7.

We aim to verify the following $LTL \setminus \mathbf{X}$ property on module A:

$$\varphi = \mathbf{G}(sync_1 \rightarrow \mathbf{F}\, t_{a1}) \ \wedge \ \mathbf{G}\big(\neg(t_{a1} \rightarrow \mathbf{F}\, t_{a2})\big).$$

Instead of constructing the global state space, verification is restricted to the meta-graph RG_A of module A (Fig. 11). The procedure follows the automata-theoretic model checking approach:

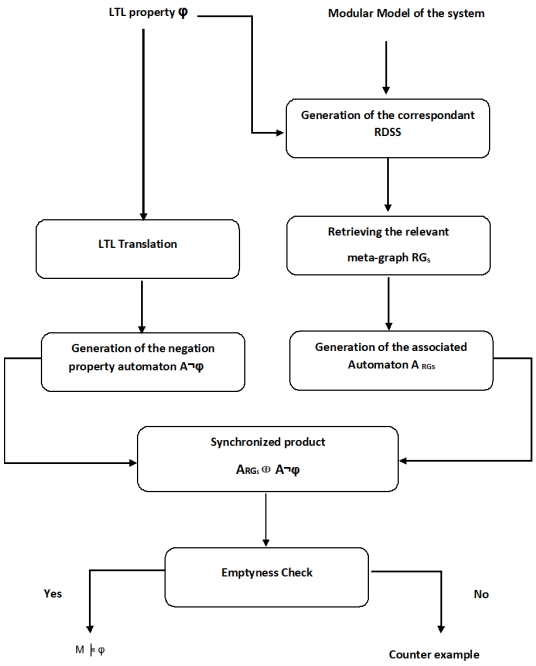

Fig. 10. The RDSS local model checking workflow.

- Negate the property φ to obtain $\neg\varphi$.
- Construct the Büchi automaton $A_{\neg\varphi}$ (Fig. 12).
- Build the automaton A_{RG_A} from the module's meta-graph.
- Compute the synchronous product $A_{RG_A} \otimes A_{\neg\varphi}$ (Fig. 13 and Fig. 14.

Initially, both automata start in their respective initial states. During the first synchronization step (Fig. 13), when the transition $sync1$ is activated in the meta-graph of module A, the automaton of $\neg\varphi$ fires $!t_{a1}$, moving to state 1. Subsequently, as illustrated in Fig. 14, the cycle in the component's strongly connected component (SCC) allows repeated firings of $t_{a2}.t_{a3}$, which are matched by the "true" loop in the automaton of $\neg\varphi$.

In this case, the product automaton is not empty. The counterexample execution is:

$$sync_1 \cdot (t_{a1}\, t_{a2})^*,$$

showing that φ is violated.

This example demonstrates concretely how RDSS enables local model checking by verifying properties using only the meta-graph of the concerned module, thus avoiding the explosion of the global state space.

These contributions, particularly the ability to perform local model checking and reduce the global state space, mark RDSS as a highly effective and scalable solution for verifying large, modular systems.

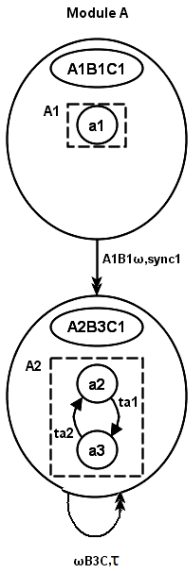

Fig. 11. Meta-graph of module A.

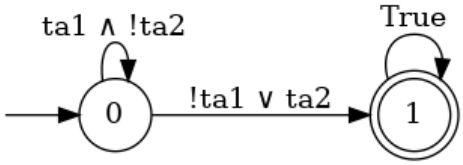

Fig. 12. Automaton of $\neg\varphi$.

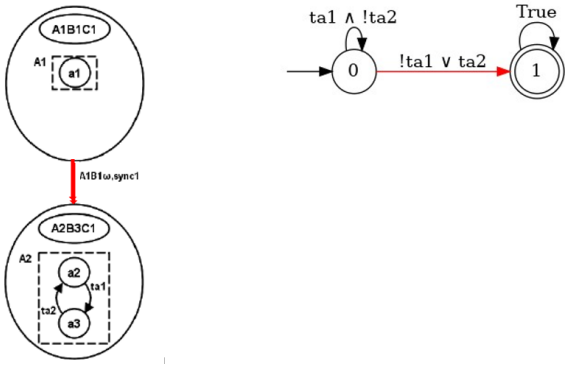

Fig. 13. First step of the synchronized product construction.

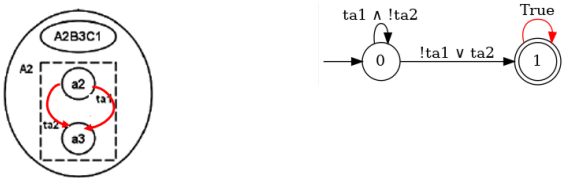

Fig. 14. Second step of the synchronized product construction.

11) Experimental setup and reproducibility: To ensure reproducibility of the reported results, we now provide additional details on the benchmarks, environment, and properties considered.

a) Benchmarks: The experiments were carried out on Modular Petri Net models representative of the case studies discussed in this paper. These include 1) an IoT-based monitoring scenario with modular communication between devices and gateways, and 2) a multi-robot patrolling system with synchronized alert transitions. For each model, different parameterizations were used to vary the number of modules and the degree of synchronization between them.

b) Verification properties: The properties evaluated include locally generic ones such as deadlock-freedom, home-space reachability, and liveness, as well as temporal properties expressed in the stutter-invariant fragment LTL$\setminus\mathbf{X}$ (e.g. $\mathbf{G}\,\neg error$ and $\mathbf{GF}\,request \rightarrow \mathbf{F}\,grant$). These properties were selected to cover both safety and liveness aspects.

c) Environment: All experiments were executed on a workstation running Ubuntu 22.04 LTS, equipped with an Intel Core i7 processor and 16 GB of RAM. The RDSS prototype was implemented in C++/ROS2, while LTSmin and SPIN were used as comparative tools. Each verification run was subject to a timeout of 30 minutes.

d) Replication: The RDSS implementation, benchmark models, and property specifications are publicly available at https://github.com/chihebabid/ros2dss, enabling replication of the results presented in this paper.

E. Positioning RDSS within the Literature

The Reduced Distributed State Space (RDSS) framework introduces a modular approach to overcome state explosion, going beyond traditional reductions by exploiting meta-graphs of individual modules. We briefly position RDSS against classical techniques and widely used tools, and highlight its application domains.

1) Reduction techniques: Compared to Partial-Order Reduction (POR), RDSS avoids redundant global interleavings by applying reductions locally at the module level. Unlike Symmetry Reduction (SR), which assumes static symmetries, RDSS flexibly combines SR with τ-hiding and meta-state fusion. In contrast to coarse-grained abstractions, RDSS preserves stuttering equivalence, ensuring correctness of LTL$\setminus X$ properties.

2) Verification tools: NuSMV is effective for small symbolic systems but does not scale to modular decomposition. LTSmin supports large state spaces yet lacks modular reduction; RDSS achieves competitive performance with lower memory needs (Fig. 8). SPIN remains powerful for explicit-state checking but does not provide modular verification like RDSS, which verifies module-specific properties directly.

3) Application domains: RDSS is particularly well-suited for distributed concurrent systems, where modules interact through synchronization but maintain partial independence. This includes multi-robot systems, IoT platforms, and cyber-physical infrastructures, where global state construction is infeasible. For example, in [82], RDSS was successfully applied to an IoT-based case study, showing that local model checking can verify safety and liveness properties efficiently without exploring the full global state space.

4) Summary: Overall, RDSS integrates the principles of POR, SR, and abstraction while overcoming their modularity limitations. By focusing on module-specific properties, RDSS positions itself as a scalable solution for distributed concurrent systems, with promising applications in robotics, IoT, and large-scale modular infrastructures. As highlighted in Table V, RDSS complements existing techniques and tools by enabling local model checking with reduced state space construction, offering clear advantages in modular and dynamic settings.

F. Limits Under Dense Synchronization

While RDSS demonstrates clear advantages when modules retain partial independence, a pathological case arises when the system exhibits dense synchronization, i.e. when most or all transitions are shared among modules (case of ERK1). In this scenario, the benefits of modular decomposition diminish, since local meta-graphs must frequently synchronize, reducing the reduction opportunities from τ-hiding and meta-state fusion.

Compared to classical techniques, RDSS in this setting behaves closer to a flat global exploration, as synchronization dominates the structure of the meta-graphs. Partial-order reduction (POR) or stubborn sets may then achieve comparable or even better pruning, since they exploit independence at the transition level rather than structural modularity. Symmetry reduction may also remain beneficial if replicated modules synchronize in a uniform manner, collapsing symmetric synchronization patterns.

These observations suggest that RDSS is most advantageous in systems where:

- Modules contain a significant proportion of local transitions relative to their synchronized ones,
- Synchronization points are sparse or limited to subsets of modules, and
- Property checking targets module-specific behaviors (e.g. local deadlock, local liveness, LTL$\setminus X$ properties tied to one module).

In contrast, in systems with dense synchronization, hybrid approaches that combine RDSS with stubborn sets or symbolic encodings may be more appropriate, ensuring that both structural and transition-level redundancies are exploited.

G. Future Directions and Extensions

While the RDSS framework already provides a scalable and modular verification approach, several promising research directions remain open. These extensions aim to further enhance scalability, applicability, and integration with modern verification practices.

1) Distributed implementations with ROS2: One natural extension of RDSS is its deployment in distributed verification environments. A first step in this direction has been demonstrated through a prototype implementation based on ROS2, where modules act as independent nodes communicating through publish/subscribe services. This distributed construction of the RDSS allows verification tasks to be executed concurrently across multiple workers, offering increased scalability for large systems with heterogeneous components.[3] Future work can explore optimizations in communication overhead and dynamic load balancing.

2) AI-driven heuristics for reduction: The construction of RDSS relies on reduction techniques, such as partial-order reduction, symmetry, and abstraction. Integrating artificial intelligence, particularly reinforcement learning and machine learning heuristics, could guide the selection of reduction strategies dynamically. For instance, AI agents may learn to prioritize transitions, identify symmetries, or detect redundant synchronizations, thereby reducing exploration costs while preserving correctness. Such integration would pave the way for adaptive and context-aware verification pipelines.

3) Hybrid reduction strategies: Another promising direction is the design of hybrid approaches that combine RDSS with other established reduction frameworks. This includes integrating counterexample-guided abstraction refinement (CEGAR), advanced stubborn sets, or symbolic encodings with the modular meta-graph construction of RDSS. Such hybrids would allow leveraging the strengths of different techniques to achieve stronger reductions without sacrificing property preservation, particularly for LTL$\setminus X$ and branching-time logics.

4) Application to emerging domains: RDSS is naturally suited for concurrent and modular architectures such as multi-robot coordination, IoT ecosystems, and cloud-native distributed systems. Extending RDSS to these domains can provide realistic benchmarks that highlight its advantages over state-of-the-art tools. For example, in the IoT-based agriculture use case [82], RDSS enabled local model checking without requiring the full global state. Similar extensions to autonomous vehicles, cyber-physical systems, and large-scale cloud infrastructures would consolidate RDSS as a practical solution to real-world scalability challenges.

5) From local to global verification: While RDSS focuses primarily on enabling local model checking at the module level, an important extension is to move toward global verification within the same modular framework. The key idea is that a global property can often be decomposed into a set of local properties, each verifiable on the corresponding module's meta-graph. By invoking local model checking for each component property and then combining the results, one can infer the truth of the global property.

This modular synthesis of verification outcomes effectively reconstructs global correctness from local evidence, avoiding the need to explore the full global state space. Such an approach bridges local and global reasoning, and can be viewed as a form of modular controller synthesis, where global guarantees emerge from the coordinated satisfaction of local obligations.

The verification of modular concurrent systems is no longer an abstract academic pursuit but a practical necessity for the safe deployment of autonomous platforms, IoT infrastructures, and large-scale distributed software. Ensuring correctness at scale requires techniques that tame the state explosion problem without sacrificing soundness.

[3]Prototype implementation available at: https://github.com/chihebabid/ros2dss

TABLE V. POSITIONING RDSS WITH RESPECT TO CLASSICAL TECHNIQUES AND TOOLS

Approach / Tool	Main Principle	Limitations	RDSS Perspective
Partial-Order Reduction (POR)	Prunes redundant interleavings by exploiting independence of concurrent transitions	Less effective with synchronous transitions across modules	Applied locally at module level to avoid global blow-up
Symmetry Reduction (SR)	Collapses symmetric states from replicated components	Assumes static symmetry; weak for dynamic or partial symmetries	Combined with τ-hiding and meta-state fusion to preserve behaviors
Abstraction	Groups concrete states into abstract representatives (predicate/counter abstraction)	Risk of oversimplification; may lose critical distinctions	Balanced with stuttering equivalence, ensuring correctness of LTL$\setminus X$
NuSMV	Symbolic model checking with BDD/SAT	Limited scalability for modular decomposition	RDSS avoids global symbolic encoding, scales better for Modular Petri Nets
LTSmin	Symbolic state-space exploration	Handles large systems, but lacks modular reductions	RDSS achieves competitive verification times with lower memory usage (see Fig. 8)
SPIN	Explicit-state temporal logic model checking	No modular decomposition; explores entire global space	RDSS verifies module-specific properties directly using local meta-graphs

In this survey, we have examined the landscape of techniques for scalable verification, including state space reduction, abstraction, compositional reasoning, symbolic methods, and distributed verification. Within this context, we have introduced and positioned the Reduced Distributed State Space (RDSS) framework as a novel and effective contribution. RDSS reduces the global state space into modular meta-graphs, enforces stuttering equivalence, and supports verification at the module level. This enables properties to be checked locally, a capability not offered by existing approaches. Our experimental comparisons against powerful tools such as LTSmin demonstrate the scalability gains of RDSS, particularly when modules are not all synchronized on the same transitions. Furthermore, we have shown that RDSS integrates naturally with techniques such as partial-order reduction, symmetry reduction, and abstraction, achieving strong reductions while maintaining correctness.

Beyond these contributions, the RDSS framework has been extended through a distributed prototype built on ROS2, opening the way to practical deployment in real-world modular systems. Its modular structure makes it especially suitable for verifying concurrent and distributed systems, ranging from multi-robot applications to cloud-based infrastructures.

While challenges such as state explosion and synchronization overhead remain, the trajectory of research indicates a strong movement toward hybrid approaches, combining symbolic encodings, distributed infrastructures, and reduction-based methods. The synergy between theoretical foundations and real-world applications promises a future where scalable verification becomes a standard component of the engineering of next-generation systems. RDSS contributes to this vision by providing a modular, scalable, and extensible verification framework that addresses both academic challenges and industrial needs.

VII. THREATS TO VALIDITY

As with any empirical and methodological contribution, our evaluation and claims are subject to certain threats to validity. We outline the most relevant ones below.

A. Model Representativeness

The benchmarks used (e.g. ERK1, Philo3, Robot1, IoT case studies) are drawn from widely used academic examples of modular and concurrent systems. However, they may not capture the full diversity of industrial-scale systems, especially those with heterogeneous architectures or non-Petri net formalisms. Future work should incorporate larger and more varied case studies to ensure broader coverage.

B. Property Selection Bias

Our focus on stuttering-invariant $LTL \setminus X$ properties, such as deadlock-freedom, local liveness, and reachability, reflects the classes of properties naturally supported by RDSS. Nevertheless, this may bias the evaluation, as other properties (e.g. CTL, fairness, or real-time constraints) are not currently addressed. Extending RDSS to these classes remains an important direction.

C. Tool Configuration Bias

Comparisons with state-of-the-art tools such as LTSmin or SPIN depend on configuration choices (e.g. symbolic encodings, partial-order options, memory allocation). While we sought fair defaults, alternative configurations might yield different relative performance. A more exhaustive benchmarking campaign would provide stronger evidence.

D. Generality Across Formalisms

RDSS is defined over Modular Petri Nets (MPNs), which provide a natural and expressive formalism for concurrent systems. This focus raises a validity threat regarding generality: it remains to be shown to what extent RDSS principles carry over to other formalisms, such as process algebras, timed automata, or hardware verification languages. Initial indications are promising, but further formalization is required.

Overall, these threats highlight the need for additional validation, both empirically (through more benchmarks and diverse case studies) and theoretically (by extending RDSS to broader classes of systems and properties). We report them here to provide a transparent account of the current scope and limitations of our contribution.

VIII. CONCLUSION

The verification of modular concurrent systems is no longer an abstract academic pursuit but a practical necessity for the safe deployment of autonomous platforms, IoT infrastructures, and large-scale distributed software. Ensuring correctness at

scale requires techniques that tame the state explosion problem without sacrificing soundness.

In this survey, we have examined the landscape of techniques for scalable verification, including state space reduction, abstraction, compositional reasoning, symbolic methods, and distributed verification. Within this context, we have introduced and positioned the Reduced Distributed State Space (RDSS) framework as a novel and effective contribution. RDSS reduces the global state space into modular meta-graphs, enforces stuttering equivalence, and supports verification at the module level. This enables properties to be checked locally, a capability not offered by existing approaches. Our experimental comparisons against powerful tools such as LTSmin demonstrate the scalability gains of RDSS, particularly when modules are not all synchronized on the same transitions. Furthermore, we have shown that RDSS integrates naturally with techniques such as partial-order reduction, symmetry reduction, and abstraction, achieving strong reductions while maintaining correctness.

Beyond these contributions, the RDSS framework has been extended through a distributed prototype built on ROS2, opening the way to practical deployment in real-world modular systems. Its modular structure makes it especially suitable for verifying concurrent and distributed systems, ranging from multi-robot applications to cloud-based infrastructures.

Nevertheless, several challenges remain open. Scalability limits may reappear when dealing with extremely large or highly interconnected modular systems, where synchronization overhead can dampen the benefits of reduction. Moreover, while RDSS preserves stuttering equivalence, there are trade-offs between accuracy and efficiency, particularly when aggressive fusion of meta-states or τ-hiding strategies are applied. Another promising direction lies in complementing RDSS with AI-driven heuristics that guide exploration or with hybrid reductions that combine symbolic encodings and distributed infrastructures. Such extensions would strengthen the adaptability of RDSS across diverse system classes and push its applicability to industrial-scale deployments.

The synergy between theoretical foundations and real-world applications promises a future where scalable verification becomes a standard component of the engineering of next-generation systems. RDSS contributes to this vision by providing a modular, scalable, and extensible verification framework that addresses both academic challenges and industrial needs, while opening avenues for innovation in hybrid and AI-augmented verification methods.

REFERENCES

[1] A. Valmari, "The state explosion problem," in W. Reisig and G. Rozenberg, Eds., *Lectures on Petri Nets I: Basic Models (ACPN 1996), Lecture Notes in Computer Science*, vol. 1491. Berlin, Heidelberg: Springer, 1998, pp. 429–528, doi: 10.1007/3-540-65306-6_21.

[2] S. Alagar and S. Venkatesan, "Techniques to tackle state explosion in global predicate detection," *IEEE Transactions on Software Engineering*, vol. 27, no. 8, pp. 704–714, Aug. 2001, doi: 10.1109/32.940566.

[3] S. C. Sølvsten and J. van de Pol, "Symbolic Model Checking in External Memory," preprint, May 2025.

[4] I. Marmanis, M. Kokologiannakis, and V. Vafeiadis, "Model Checking C/C++ with Mixed-Size Accesses," *Proc. ACM Program. Lang.*, vol. 9, no. POPL, Art. 75, Jan. 2025, 21 pp., doi: 10.1145/3704911.

[5] M. Dam and H. Nemati, "Compositional Verification of Concurrency Using Past-Time Temporal Epistemic Logic," preprint, Feb. 2025.

[6] J. Srba, "On-The-Fly Verification: Advancements in Dependency Graphs (Invited Talk)," in *Proc. 36th International Conference on Concurrency Theory (CONCUR 2025)*, Leibniz International Proceedings in Informatics (LIPIcs), vol. 348, pp. 3:1–3:5, Dagstuhl, Germany, 2025, doi: 10.4230/LIPIcs.CONCUR.2025.3.

[7] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk, "LTSmin: High-Performance Language-Independent Model Checking," in *Proc. 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, vol. 9035, pp. 692–707, Springer, 2015, doi: 10.1007/978-3-662-46681-0_61.
in *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, vol. 9035, 2015. Extended overview of recent (2024) multi-core symbolic and partial-order reduction back-ends.

[8] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Boston, MA: Addison–Wesley, 2004.

[9] C. A. Abid, K. Klai, H. Ouni, and M. K. Kholladi, "Distributed Modular Verification Based on Symbolic Observation Graphs," *International Journal of Critical Computer-Based Systems*, vol. 10, no. 1, pp. 36–60, 2020, doi: 10.1504/IJCCBS.2020.110498.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *Proc. 14th Int. Conf. on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, vol. 2404, pp. 359–364, Springer, 2002, doi: 10.1007/3-540-45657-0_29.

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

[12] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[13] S. Christensen and L. Petrucci, "Modular state space analysis of coloured Petri nets," in *Lectures on Petri Nets I: Basic Models (ACPN 1996), Lecture Notes in Computer Science*, vol. 1491. Berlin, Heidelberg: Springer, 1998, pp. 429–528, doi: 10.1007/3-540-65306-6_21.

[14] C.Edmund and Emerson, E. (2008). "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic", in Lecture Notes Comput Sci. 131. 196-215.

[15] Andrés, Miguel. Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems. Diss. Radboud University, Nijmegen, 2011.

[16] E. M. Clarke, P. Huang, M. Nováček, M. Paulk, B. Schuppan, and P. Zuliani, "Model Checking and the State Explosion Problem," Tech. Rep., Carnegie Mellon University, 2011.

[17] Li, Shuo, et al. "On-the-fly unfolding with optimal exploration for linear temporal logic model checking of concurrent software and systems." Automated Software Engineering 32.2 (2025): 60.

[18] F. Herbreteau, S. Larroze-Jardiné, G. Point, and I. Walukiewicz, "Revisiting Stateful Partial-Order Reduction," 2024.

[19] R. Cleaveland and C. Trippel, "Memory Consistency Model-Aware Cache Coherence for Heterogeneous Hardware," in *Proc. 25th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2024.

[20] S. Merz, "Stuttering Equivalence and Stuttering Invariance," Inria Nancy & LORIA Technical Report, March 17, 2025.

[21] T. R. Nicely, "Pentium FDIV flaw," Tech. Rep., Lynchburg College, 1995.

[22] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *Computer*, vol. 26, no. 7, pp. 18–41, IEEE, 1993, doi: 10.1109/MC.1993.274940.

[23] House Committee on Transportation and Infrastructure, "The Boeing 737 MAX Aircraft: Costs, Consequences, and Lessons from Its Design, Development, and Certification," U.S. Government Report, 2020.

[24] S. Appicharla, "Analyses of the Boeing 737 MAX accidents: formal models and safety implications," *Sociotechnical Systems Journal*, vol. 12, pp. 1–15, 2023.

[25] National Transportation Safety Board (NTSB), "Collision Between Vehicle Controlled by Developmental Automated Driving System and Pedestrian," Highway Accident Report NTSB/HAR-19/03, 2019.

[26] P. Kocher et al., "Spectre Attacks: Exploiting Speculative Execution," in *Proc. 40th IEEE Symposium on Security and Privacy (S&P)*, pp. 1–19, 2019, doi: 10.1109/SP.2019.00002.

[27] Nasrabadi, Faezeh, Robert Kuennemann, and Hamed Nemati. "Symbolic Parallel Composition for Multi-language Protocol Verification." 2025 IEEE 38th Computer Security Foundations Symposium (CSF). IEEE Computer Society, 2025.

[28] Amazon Web Services, "Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region," Incident Report, Dec. 2021.

[29] C. Kalaskar and T. S., "Fault tolerance of cloud infrastructure with machine learning," *Cybernetics and Information Technologies*, vol. 23, no. 4, pp. 90–103, 2023, doi:10.2478/cait-2023-0034.

[30] Author Unknown, "Optimizing Resource Usage in Cloud Infrastructure," *Proc. ACM Symposium on Cloud Computing*, June 2025.

[31] C. Coti, L. Petrucci, C. Rodríguez, M. Sousa, and H. T. T. Nguyen, "Quasi-Optimal Partial Order Reduction," *Formal Methods in System Design*, vol. 57, pp. 3–33, 2021, doi: 10.1007/s10703-020-00350-4.

[32] E. Albert, L. Gómez, and A. Legay, "Optimal Context-Sensitive Dynamic Partial-Order Reduction with Observers," *Journal of Systems Architecture*, vol. 144, 2023, doi: 10.1016/j.sysarc.2023.102853.

[33] X. Yang and C. Ye, "Dynamic Partial Order Reduction Based on Interference Solution," *Research Square*, preprint, Version 1, 18 Sept. 2023. doi: 10.21203/rs.3.rs-3257081/v1.

[34] T. Gibson-Robinson and G. Lowe, "Symmetry Reduction in CSP Model Checking," *Int. Journal on Software Tools for Technology Transfer*, vol. 21, no. 4, pp. 481–496, 2019, doi:10.1007/s10009-019-00516-4.

[35] R. Barbosa, A. Fonseca, and F. Araujo, "Reductions and Abstractions for Formal Verification of Distributed Round-Based Algorithms," *Software Quality Journal*, vol. 29, no. 3, pp. 705–731, 2021, doi:10.1007/s11219-020-09539-6.

[36] P. Eichler, S. Jacobs, and C. Weil-Kennedy, "Parameterized Verification of Systems with Precise (0,1)-Counter Abstraction," in *Proc. 26th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS, vol. 15529, pp. 101–124, Springer, 2025.

[37] T. Neele, A. Valmari, and A. Wijs, "Stubborn Set Reduction for Two-Player Reachability Games," in *Proc. 2021 Int. Conf. on Application of Concurrency to System Design (ACSD)*, pp. 23–33, IEEE, 2021. doi: 10.1109/ACSD51164.2021.00011.

[38] S. Ben Rhouma, K. Klai, and H. Ben-Abdallah, "On Improving Model Checking of Time Petri Nets and Its Application to the Formal Verification," *Int. J. of Information System Modeling and Design (IJISMD)*, vol. 12, no. 3, pp. 1–22, 2021. doi: 10.4018/IJISMD.20210701.oa1.

[39] Y. Kim, W. Jamroga, W. Penczek, and L. Petrucci, "Practical Abstractions for Model Checking Continuous-Time Multi-Agent Systems," in *Proc. AAMAS 2025*, May 2025.

[40] G. Bruns, "User-Driven Abstraction for Model Checking," preprint, 2023. . Available: https://arxiv.org/abs/2307.15820

[41] A. Benveniste et al., "Contracts for System Design," *Found. Trends Electron. Des. Autom.*, vol. 12, no. 2-3, pp. 124–400, 2018. doi: 10.1561/1000000053.

[42] K. Pierce et al., "Automated Contract Composition and Refinement for System-on-Chip Integration," in *Proc. Design, Autom. Test Eur. Conf. (DATE)*, 2023, pp. 1–6. doi: 10.23919/DATE56975.2023.10137125.

[43] N. Tabassam et al., "Contract-Based Design Methodologies for Cyber-Physical Systems," in *Proc. Int. Conf. World Conf. Smart Trends Syst., Secur. Sustainability (WorldS4)*, Springer, Singapore, 2025.

[44] O. Goudsmid, O. Grumberg, and S. Sheinvald, "Compositional Model Checking for Multi-Properties," in *Proc. Int. Conf. Verif., Model Checking, Abstract Interpret. (VMCAI)*, 2021, pp. 1–23. doi: 10.1007/978-3-030-67067-2_1

[45] R. Krauss, M. Goli, and R. Drechsler, "EDDY: A Multi-Core BDD Package with Dynamic Memory Management and Reduced Fragmentation," in *Proc. 28th Asia South Pac. Des. Autom. Conf. (ASP-DAC)*, 2023, pp. 1–6.

[46] O. S. Bak et al., "GPU Accelerating Statistical Model Checking for Extended Timed Automata," in *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part II*, Cham: Springer Nature Switzerland, 2024, pp. 267–292.

[47] S. Evangelista, L. M. Kristensen, and L. Petrucci, "Evaluation of a Distributed Explicit State Space Exploration Algorithm with State Reconstruction for RDMA Networks," *Int. J. Softw. Tools Technol. Transfer*, vol. 27, no. 2, pp. 149–168, 2025.

[48] L. Davis and T. Ji, "Evaluating SAT and SMT Solvers on Large-Scale Sudoku Puzzles,"preprint 2025.

[49] Spork, Timm, et al. "A Spectrum of Approximate Probabilistic Bisimulations." 35th International Conference on Concurrency Theory. 2024.

[50] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on UPPAAL," *Formal Methods in System Design*, vol. 23, no. 1, pp. 47–104, 2003, doi: 10.1023/A:1023688603802.

[51] I. Grobelna, K. Gajewski, and A. Karatkevich, "A Systematic Review on the Applications of UPPAAL," *Sensors*, vol. 25, no. 11, p. 3484, 2025, doi: 10.3390/s25113484.

[52] J. Laarman, D. van de Pol, and J. Wijs, "LTSmin: High-Performance Language-Independent Model Checking," in *Proc. 23rd Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, vol. 10205, pp. 692–707, Springer, 2017, doi: 10.1007/978-3-662-54577-5_9.

[53] J. Barnat, L. Brim, and P. Ročkai, "DiVinE: Parallel Distributed Model Checker," in *Proc. Int. Conf. on High Performance Computing (HiPC)*, pp. 11–20, IEEE, 2010, doi: 10.1109/HIPC.2010.5696880.

[54] C. Hensel, J.-P. Katoen, J. Krčál, F. Olmedo, J. Wojtczak, and L. Zhang, "The STORM Model Checker: A Modern, Symbolic, and Flexible Framework for Probabilistic Verification," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 24, no. 2, pp. 205–227, 2022, doi: 10.1007/s10009-021-00633-z.

[55] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-Time Systems," in *Proc. 23rd Int. Conf. on Computer Aided Verification (CAV)*, LNCS, vol. 6806, pp. 585–591, Springer, 2011, doi: 10.1007/978-3-642-22110-1_47.

[56] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "PAT: Towards Flexible Verification under Fairness," in *Proc. 21st Int. Conf. on Computer Aided Verification (CAV)*, LNCS, vol. 5643, pp. 709–714, Springer, 2009, doi: 10.1007/978-3-642-02658-4_50.

[57] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic Predicate Abstraction of C Programs," *ACM SIGPLAN Notices*, vol. 47, no. 4a, pp. 37–47, 2012, doi: 10.1145/2103621.2103627.

[58] M. Kokologiannakis, I. Marmanis, and V. Vafeiadis, "SPORE: combining symmetry and partial order reduction," *Proc. ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1781–1803, 2024,

[59] M. Rashid, A. Ali, S. A. Shah, and F. Ahmed, "Formal modeling and verification of IoT-based smart transport system using SPIN model checker," in *Proc. 2024 IEEE 1st Karachi Section Humanitarian Technology Conference (KHI-HTC)*, Karachi, Pakistan, pp. 1–6, IEEE, 2024, doi: 10.1109/KHI-HTC61644.2024.10722564.

[60] N. Suresh Kumar and G. Santhosh Kumar, "Abstracting IoT protocols using timed process algebra and SPIN model checker," *Cluster Computing*, vol. 26, no. 2, pp. 1611–1629, 2023, doi: 10.1007/s10586-022-03769-5.

[61] Y. Yang and T. Holvoet, "Efficient Model Checking with Semantically-Equivalent Models for vGOAL," in *Proc. 24th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1–9, IFAAMAS/ACM, 2025.

[62] S. Kanwal and W. Fokkink, "A Framework for Model-Based Specification and Verification in Feature-Oriented Software Product Lines," in *Proc. Int. Conf. on Fundamentals of Software Engineering (FSEN)*, Cham, Switzerland: Springer Nature, 2025, pp. 1–18.

[63] M. Krichen, "Timed automata-based strategy for controlling drone access to critical zones: A UPPAAL modeling approach," *Electronics*, vol. 13, no. 13, p. 2609, 2024, doi: 10.3390/electronics13132609.

[64] D. Basile, "Modelling, verifying and testing the contract automata runtime environment with UPPAAL," in *Proc. Int. Conf. on Coordination Models and Languages (COORDINATION)*, Cham, Switzerland: Springer Nature, 2024, pp. 123–139.

[65] M. Osama and A. Wijs, "Hitching a ride to a lasso: massively parallel on-the-fly LTL model checking," in *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Cham, Switzerland: Springer Nature, 2024, pp. 145–165.

[66] A. Duret-Lutz, D. Poitrenaud, and Y. Thierry-Mieg, "Simplifying LTL Model Checking Given Prior Knowledge," in *Proc. Int. Conf. on Applications and Theory of Petri Nets and Concurrency (PETRI NETS)*, Cham, Switzerland: Springer Nature, 2025, pp. 45–65.

[67] F. R. Monteiro, M. R. Gadelha, and L. C. Cordeiro, "Model checking C++ programs," *Software Testing, Verification and Reliability*, vol. 32, no. 1, e1793, 2022.

[68] R. van Spreuwel and A. Wijs, "No need to be stubborn: partial-order reduction for GPU model checking revisited," in *Proc. Int. Symp. on Leveraging Applications of Formal Methods (ISoLA)*, Cham, Switzerland: Springer Nature, 2024, pp. 233–251.

[69] Y. Gao, D. Xu, and L. Chen, "Prediction and optimization of memory fragmentation based on eBPF and LightGBM in high-load environments," in *Proc. 2024 7th Int. Conf. on Artificial Intelligence and Pattern Recognition (AIPR)*, pp. 342–350, ACM, 2024. doi: 10.1145/3670810.3670924.

[70] Z. Huang, Y. Zhou, L. Wang, and J. Zhang, "Reducing GPU Memory Fragmentation via Spatio-Temporal Planning for Efficient Large-Scale Model Training," 2025 . Available: https://arxiv.org/abs/2507.16274

[71] M. Lampacrescia, M. Klauck, and M. Palmas, "Towards verifying robotic systems using statistical model checking in STORM," in *Proc. Int. Conf. on Bridging the Gap between AI and Reality (AIReality)*, Cham, Switzerland: Springer Nature, 2024, pp. 89–106.

[72] T. Quatmann, "What is the best algorithm for MDP model checking?," *International Journal on Software Tools for Technology Transfer (STTT)*, pp. 1–7, 2025, doi: 10.1007/s10009-025-00791-3.

[73] I. Valkov, A. F. Donaldson, and A. Miller, "Synchronisation in Language-Level Symmetry Reduction for Probabilistic Model Checking," in *Proc. Int. Symp. on Model Checking Software (SPIN)*, Cham, Switzerland: Springer Nature, 2024, pp. 210–228.

[74] H. Wang, Y. Zhao, X. Liu, and L. Zhang, "Towards a Model-Based Framework for Automated Traceable Systems and Probabilistic Model Checking," in *Proc. 2025 6th Int. Conf. on Computer Engineering and Application (ICCEA)*, IEEE, pp. 56–63, 2025. doi: 10.1109/ICCEA61623.2025.10789765.

[75] M. Djukic, D. Stojanovic, J. Protic, and Z. Rakic, "Correct orchestration of federated learning generic algorithms: Python translation to CSP and verification by PAT," *International Journal on Software Tools for Technology Transfer*, vol. 27, no. 1, pp. 21–34, 2025, doi: 10.1007/s10009-024-00789-5.

[76] Y. Yu, X. Zhou, Z. Liu, and Z. Yang, "Model checking concurrency in smart contracts with a case study of safe remote purchase," in *Proc. Int. Conf. on Formal Engineering Methods (ICFEM)*, Singapore: Springer Nature, 2024, pp. 145–161.

[77] P. Chatterjee, B. Jobstmann, D. Kroening, and F. Somenzi, "Distributed bounded model checking," *Formal Methods in System Design*, vol. 64, no. 1, pp. 50–72, 2024, doi: 10.1007/s10703-023-00421-9.

[78] R. Queiroz, A. Author2, A. Author3, and A. Author4, "A driver-vehicle model for ADS scenario-based testing," *IEEE Transactions on Intelligent Transportation Systems*, vol. 25, no. 8, pp. 8641–8654, 2024, doi: 10.1109/TITS.2023.3341234.

[79] S. Khlifa, C. A. Abid, and B. Zouari, "Local Model Checking on a Modular System," in *Proc. 10th Int. Conf. on Control, Decision and Information Technologies (CoDIT)*, IEEE, 2024, pp. 2899–2904. doi: 10.1109/CoDIT60129.2024.10563056.

[80] H. Ouni, C. A. Abid, and B. Zouari, "A Distributed State Space for Modular Petri Nets," in *Proc. 7th Int. Conf. on Modelling, Identification and Control (ICMIC)*, IEEE, 2015, pp. 1–6. doi: 10.1109/ICMIC.2015.7409426.

[81] S. Khlifa, C. A. Abid, and B. Zouari, "A Reduced Distributed State Space for Modular Petri Nets," in *Proc. Int. Conf. on Advanced Information Networking and Applications (AINA)*, Cham, Switzerland: Springer, 2023, pp. 319–331. doi: 10.1007/978-3-031-27077-2_28.

[82] S. Khlifa, C. A. Abid, A. B. Letaifa, and B. Zouari, "Local Model Checking on an IoT-Based System: Use Case of Cellular M2M in Agriculture," in *Proc. Int. Conf. on Advanced Information Networking and Applications (AINA)*, Cham, Switzerland: Springer, 2025, pp. 356–367. doi: 10.1007/978-3-031-59656-8_28.