A Technique to Support Incremental Construction and Verification in Component-Based Software Development

Faranak Nejati[1], Ng Keng Yap[2], Abdul Azim Abd Ghani[3]
Lee Kong Chian Faculty of Engineering Science (LKC FES)-Department of Computing (DC),
Universiti Tunku Abdul Rahman (UTAR), Selangor, Malaysia[1]
Faculty of Computer Science and Information Technology and Institute for Mathematical Research,
Universiti Putra Malaysia (UPM), Seri Kembangan 43400, Malaysia[2]
Faculty of Computer Science and Information Technology, Universiti Putra Malaysia (UPM),
Seri Kembangan 43400, Malaysia[3]

***Abstract*—Technological advancements in recent decades have significantly increased the scale and complexity of software systems, which poses challenges to their development and reliability. Component-based software development (CBSD) offers a promising solution by enabling modular and efficient software construction. However, CBSD alone cannot fully address challenges such as ensuring reliability and avoiding errors like deadlocks. Verification techniques, such as model-checking, are necessary to ensure the correctness of CBSD systems. Despite its effectiveness in verifying system properties, model-checking faces a critical issue known as state-space explosion (SSE), which hinders scalability. This study introduces an incremental verification technique for CBSD to address SSE and ensure deadlock freedom. The proposed technique incrementally constructs and verifies component-based systems, eliminating verified portions of components to minimize state-space size during subsequent verification steps. It utilizes a component model that supports encapsulation of computation and control, making incremental verification feasible. Evaluation of the technique using coloured petri nets with non-trivial case studies demonstrates its ability to detect deadlocks early and manage SSE effectively, thereby improving the efficiency of the verification process.**

***Keywords*—Component-based software development; incremental software construction; software verification**

I. Introduction

The rapid growth of technology has led to increasingly complex software systems, which necessitate advanced development methods to manage this complexity while ensuring system reliability. One such approach is Component-based Software Development (CBSD), which optimizes software development by breaking down systems into reusable, self-contained components [1]. CBSD allows for modular design, which can reduce development time and cost. However, as software systems grow in scale, even CBSD faces challenges in ensuring system correctness and avoiding errors, such as deadlocks, during development.

To enhance the reliability of CBSD, formal verification methods are needed to identify and address potential errors. Model-checking is a well-established technique for verifying the correctness of software systems by exhaustively exploring the state space (SS) of the system [2]. It can automatically detect issues like deadlocks by analyzing system behaviors. However, model-checking suffers from the state-space explosion (SSE) problem, where the number of states to be analyzed grows exponentially with system complexity, making verification infeasible for large systems [3].

Addressing SSE in model-checking is crucial for the practical application of verification in CBSD. Despite various approaches in the literature to mitigate SSE, a critical gap remains: the need for techniques that integrate verification throughout the development process rather than waiting until system construction is completed. Early detection of design errors is highly desirable, as it can significantly reduce the cost of fixing these errors compared to discovering them during later stages of development or maintenance [4].

Incremental verification is a promising approach to overcoming challenges in SSE by verifying components as they are developed and integrated, rather than waiting for the entire system to be completed. This approach enables early detection of defects and manages SSE more effectively by verifying smaller portions of the system incrementally [5]. However, despite its potential, incremental verification remains underexplored in the context of CBSD. This gap is partly due to the difficulties in maintaining consistent behavior and synchronization across different increments, making it challenging to ensure that new components integrate seamlessly without disrupting previously verified functionality [6].

To address this gap, this paper introduces a technique for the incremental construction and verification of component-based systems using a specialized component model called PUTRACOM [7]. PUTRACOM is designed to support encapsulation and concurrency, making it particularly well-suited for incremental verification. Its ability to decouple computation and control addresses the challenges of managing interdependencies between components, which are commonly encountered in CBSD. The proposed approach emphasizes two critical aspects: maintaining system behavior consistency during the incremental integration of components and integrating verification at each stage to avoid redundant checks on already verified components.

This research offers two key contributions. First, it presents a method for preserving system behavior and synchronization

during the incremental integration process, ensuring that the addition of new components does not compromise the verified properties of existing ones. Specifically, this research focuses on verifying the property of deadlock freedom. Second, it introduces a verification strategy that isolates previously verified components, preventing their re-verification and thereby reducing the overall state-space size during subsequent verification stages. This technique is evaluated using coloured Petri nets, a well-established tool for modeling and verifying concurrent systems [8], and applied to complex case studies, including the Common Component Modeling Example (CoCoME) [9].

The results show that the proposed incremental verification approach effectively mitigates the SSE problem and enhances the efficiency of the verification process in CBSD. By identifying errors early in the design stage and reducing verification effort, this approach improves the practical application of model-checking in component-based systems.

II. Background

This section provides preliminary information about the PUTRACOM component model [7], which is designed for building concurrent systems using exogenous connectors, inspired by the X-MAN model. PUTRACOM comprises three fundamental units: computation units (CU), observer/observable units (OOU), and exogenous connectors.

The CU encapsulates all computations and is modeled using reactive transition systems (RTSs) [4]. This encapsulation allows for decoupling between components while ensuring fixed behavior.

The OOU captures events visible to other components or the computing environment, modeled by interface automata [10]. This approach simplifies communication, avoiding complexities associated with traditional models like message passing and port-to-port connections. The OOU observes all events from other components or the environment. An atomic component in PUTRACOM is defined through RTSs, which represent potential behaviors of discrete systems. RTSs distinguish three types of events: input, internal, and output. Below is the formal definition of a computation unit based on RTS.

Definition 1 (computation unit)**.** A CU is a reactive transition system defined by the tuple (s_0, S, E, G, Δ), where:

- S is a set of states, with s_0 as the initial state;
- E is a set of events, including input events (E^I), output events (E^O), and hidden events (E^H);
- G is a guard function assigning boolean constraints to events;
- $\Delta \subseteq S \times E \times G \times S$ is a set of transitions.

Input and output events are considered observable ($E^{Obs} = E^I \cup E^O$) and accessible from the OOU. The OOU is modeled as a multiset, meaning the same event can occur multiple times. These events influence the component's state depending on whether the input events are admissible. Output events produced by the CU are also observable in the OOU. The formal definition of an OOU is as follows.

Definition 2 (Observer/Observable Unit)**.** The OOU is a finite multi-set defined as (E^{Obs}, f), where $f : E^{Obs} \to \mathbb{N}$ maps observable events to their multiplicities.

III. Proposed Solution and Methodology

This section outlines the proposed methodology focuses on the incremental construction and verification within the extended PUTRACOM framework. An overview of the solution is illustrated in Fig. 1. In the first step, the approach utilizes exogenous connectors, ensuring encapsulated component interactions. The methodology involves defining key elements such as noticing and publishing functions to manage event interactions between components and connectors. The composition of components is then formalized, allowing for structured integration within the PUTRACOM system. The detailed explanations are in Section IV.

The second step detailed in Section V is the incremental construction approach to develop a modular system by gradually integrating components and composite components. Each increment, will be systematically added using connectors for synchronization and interaction. The methodology will involve defining trace functions to analyze component behavior and applying n-way synch and asynch compositions to ensure effective integration. The process includes maintain consistency between the original and composed systems, preserving interaction behavior throughout the construction stages by defining observable traces and refinement relations.

Third step is relayed to the incremental verification detailed in Section VI, verifying each system increment before adding the next. It generates and checks the state space for local and global properties (deadlock-freeness). To manage state space explosion, it reduces state space size by focusing on observable behaviors and removing hidden parts, ensuring efficient and consistent verification. To validate our approach, we modeled three case studies using Coloured Petri Nets (CPNs), detailed in Section VII. Evaluation metrics include the number of generated state spaces, memory consumption, and processing time.

IV. Initialization of the Extended PUTRACOM Framework

This section presents additional definitions for extended PUTRACOM related to incremental construction and verification. Composition operators, termed connectors, are exogenous, allowing components to remain encapsulated without directly sending or receiving messages. The OOU manages and controls observable events without involving the CUs in the control process.

Definition 3 (connector)**.** A *connector* is a tuple $\Gamma = (L, T, Sub, E_\Gamma, G_\gamma)$, where:

- L is a set of connection lines, representing the physical or logical links between components;
- $T = \{sync, async, con, seq, itr\}$ indicates the types of connectors, including synchronous, asynchronous, conditional, sequential, and iterative interactions;
- $Sub = \{b_0, b_1, b_2, ..., b_n\}$ is a set of components subscribed to the connector;

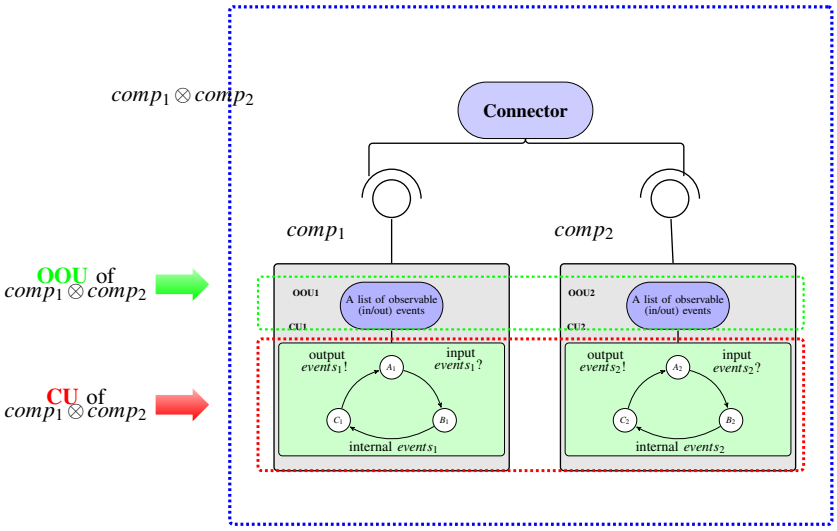

Fig. 1. Components with details in PUTRACOM.

- E_Γ is a set of events that consists of two mutually disjoint sets: output events $E_\Gamma^O \subseteq (E^O \cup env)$ and input events $E_\Gamma^I \subseteq (E^I \cup env)$, where, E^O is the set of output events; E^I is the set of input events defined for the system; *env* represents the set of external events or environmental conditions that can influence or interact with the connector.
- $G_(\gamma) = \{g_0, g_1, ..., g_n\}$ is a set of boolean constraints over the connector $\gamma \in L$.

Once output events are ready, the corresponding OOU notifies the connector using a *noticing* function.

Definition 4 (noticing)**.** Let Γ be a set of connectors and S be a set of components where $s \in S$. A *noticing* function $N : E^O \rightarrow \Gamma$ maps output events of components to their corresponding connectors, such that $N(e_o) = \gamma$, where $\gamma \in \Gamma$ and $e_o \in E^O$.

The activation and occurrence of a noticing depend on verifying predefined conditions associated with the event, as specified by the function *BoolConst*. The following definition serves as the primary definition of an atomic component in PUTRACOM, summarizing the core concepts presented in the preceding definitions.

Definition 5 (atomic component)**.** An atomic component in PUTRACOM is a tuple $com = \{CU, OOU, init, G, N\}$, where:

- $CU = (s_0, S, E, V, \Delta)$ is a reactive transition system;
- *OOU* is the observer/observable unit of the component, defined as a multiset (E^{Obs}, f);
- $init : E \rightarrow S$ is an initial function that assigns each event $e \in E$ to its corresponding initial state s_0 in S;
- $G_{com} : E \rightarrow$ BoolCons is a function mapping each event to its corresponding set of user-defined Boolean constraints;
- $N : E^O \rightarrow \Gamma$ is a noticing function that notifies the corresponding connector.

Once a connector is notified of an event occurrence by the OOU, it is responsible for disseminating this event to the relevant components. This process is managed by the publishing function, which ensures that all intended recipients receive the event.

Definition 6 (publishing)**.** Let $b \in Sub$, $e_o \in E_\Gamma^O$, $\gamma \in \Gamma$, and *env*. A *publishing* is defined as a function $P : E_\Gamma^O \rightarrow OOU$ that maps an output event e_o through the connector γ to the corresponding OOU, such that $OOU \subseteq \{OOU_{\{b_i\}_{i=0}^n} \cup env\}$.

Publishing an event requires satisfying conditions from G_γ. If met, the event is sent to the OOU of the relevant subscribers. In the noticing function, events are marked as "!", and in publishing, as "?".

Once a connector received an event by noticing and published the event, an interaction occurs.

Definition 7 (interaction)**.** Let the connector $\gamma \in \Gamma$ and $b \in Sub$. Consider the following events:

- There exists $e_o \in E^O$ such that $N(e_o \cup env) = \gamma$.
- There exists $e_i \in E_\gamma^I$ such that $P(e_i) = OOU$ and $OOU \subseteq \{OOU_{\{b_i\}_{i=0}^n} \cup env\}$.

Then, the set of interactions Π contains a finite number of interactions π, where $\pi = (e_o, \gamma, e_i)$ such that $\gamma \cap \Gamma = 1$. This research uses $(e_o, e_i)_\gamma$ or $\xrightarrow{e}_\Gamma$ instead of $\pi = (e_o, \gamma, e_i) \in \Pi$. Interactions can occur only when specific conditions are met which it is called admissible events.

Definition 8 (admissible interaction)**.** An interaction is admissible only if the following rules are satisfied:

1) $\forall e \in \Pi$, e is admissible,
2) $\xrightarrow{e}_\Gamma \in \Pi \models G_\Gamma$. It means $\xrightarrow{e}_\Gamma$ satisfies the set of conditions of connector Γ for publishing.

Based on the definitions provided, we can establish that composite components form the foundation for creating efficient and scalable systems within the PUTRACOM framework.

Definition 9 (composite components)**.** A composite component in PUTRACOM is a tuple $compos = \{CU, OOU, G, \Pi\}$.

V. Incremental Construction

Incremental construction in PUTRACOM consists of a finite set of increments denoted by *INC*. Each $inc \in INC$ represents a component or composite component, with the first increment serving as a foundational touchstone for the incremental composition, termed "basic," which connects other increments via a connector.

Definition 10 (Basic and Next Increments)**.** Let $C = \{c_0, c_1, \ldots, c_n\}$ be a set of all required components or composite components needed to construct the target system *Target*. The term *trail* refers to the composite component currently being built at the current construction level of the system *Target*. The definitions of *basic* and *next increment* are as follows:

$$basic = \{c \in C \mid \text{trail} = \emptyset \text{ and } c.\text{trail} \subseteq Target\}$$

$$inc_{next} = \{c \in C \mid \text{trail} \neq \emptyset \text{ and } c.\text{trail} \subseteq Target\}$$

Each increment consists of both CUs and OOUs, making it crucial to understand their behavior for effective component composition and system construction. To facilitate this understanding, a trace function is employed to extract the behavior of the CU, formally defined as a recursive function that returns a sequence of events originating from the initial state s_0.

Definition 11 (trace)**.** A transition *trace* of a RTS, denoted by ζ, is a sequence of events $e_0, e_1, \ldots, e_i$ such that for all $i \geq 0$, there exist states $s_0, s_1, \ldots, s_{i+1} \in S$ and guards $g_i \in G$ satisfying the conditions: $\zeta = (s_i, e_i, g_i, s_{i+1}), (s_{i+1}, e_{i+1}, g_{i+1}, s_{i+2}), \ldots, (s_n, e_n, g_n, s_{n+1}) \in \Delta$ or equivalently, $\zeta = s_i \xrightarrow{e_i, g_i} s_{i+1}, s_{i+1} \xrightarrow{e_{i+1}, g_{i+1}} s_{i+2}, \ldots, s_n \xrightarrow{e_n, g_n} s_{n+1}$. A trace function ε generates a sequence of transitions from an initial state s_0 based on admissible transitions.

A connector is defined by a set of types T that represent synchronization among subscribed increments and their interactions, highlighting the necessity of composing multiple increments. The following definitions describe the n-way composition capability of increments in PUTRACOM connectors.

Definition 12 (n-way sync composition)**.** Let C_i and E_i indicate the set of increments and their corresponding events respectively. The n-way sync composition of n increments is defined as follow:

$$\|_{i=0}^{n} \ (C_i, E_i) = C_1 \ {}_{E_1}\|_{\{E_2 \cup E_3 \cup \ldots \cup E_n\}} \ (\ldots (C_{n-1} \ {}_{E_{n-1}}\|_{E_n} \ C_n)) \quad (1)$$

Definition 13 (n-way asynchronous composition)**.** Let C_i indicates the sets of components. The $n-way\ asynchronous\ composition$ of n components are defined as follow:

$$|||_{i=0}^{n} \ (C_i) = C_1 \ ||| \ C_2 \ \ldots \ ||| \ C_n \quad (2)$$

In PUTRACOM, components encapsulate computation and are not called by other components. The CU exhibits fixed behavior without the environment's involvement. The only shared aspect is the OOU, which is involved in composition. Consequently, the hidden part can be omitted in traces, resulting in what is known as an Observable trace.

Definition 14 (observable trace)**.** *An observable trace is a sequence of events over a component or a composite component defined by:* $\zeta^{Obs} = \{(e_1), (e_2), \ldots, (e_i) \mid e \in E \backslash E^H\}$.

To obtain an observable RTS (ORTS) from an original RTS, a zoom-out function is presented. Let $\exists e \in E^H$ and $s, s' \in S$, then, the zoom-out function omits the arc $s \xrightarrow{e} s'$. If $s \xrightarrow{e_i, e_j} s'$ such that $i \neq j$ and $e_j \in E^{Obs}$ and $e_i \in E^H$, the zoom-out function will only omit $s \xrightarrow{e_i} s'$. The ORTS of the component is shown in Fig. 2.

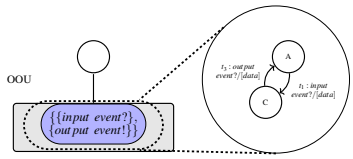

Fig. 2. An ORTS example of a component.

Observable RTS focuses on noticing, publishing, and interactions through connectors and will later help calculate possible component interactions. It also introduces an important aspect: the conformance between ORTS and RTS, defined using the refinement relation (RR) [11]. Refinement relations indicate that every behavior of system A is allowed by system (or specification) B.

Definition 15 (refinement relation)**.** Consider a RTS R and its corresponding ORTS S such that $E_S^I = E_R^I$ and $E_S^O = E_R^O$. S conforms to R, written $S \preceq R$, there is an RR between R and S, if the following conditions hold:

- $s_0 = r_0$, thus $s_0 \preceq r_0$ where s_0 and r_0 are the initial states of S and R respectively.
- For every transitions of R such that $\forall e \in E^H$ there exist a successor of r indicates by r' (where $r\prime \neq r$) and a RR between $r\prime$ and s, written by $s \preceq r\prime$.
- For every transitions of R such that $\forall e_i \in E^I$, $\exists r \xrightarrow{e_i} r\prime$ there exist a successor of s, written by $s \xrightarrow{e_i} s\prime$, such that $s\prime \preceq r\prime$.
- For every transitions of R such that $\forall e_o \in E^O$, $\exists r \xrightarrow{e_o} r\prime$ there exist a successor of s, written by $s \xrightarrow{e_o} s\prime$, such that $s\prime \preceq r\prime$.

Fig. 3 shows the composition of ORTSs from two increments, with the first component as *basic* and their combination as trail. Adding any increment signifies the next construction level, inc_{next}. Connectors are exogenous, defining connection and communication between increments. The initial increment serves as the touchstone for the incremental composition, referred to as inc_{base}, connected by a connector Γ.

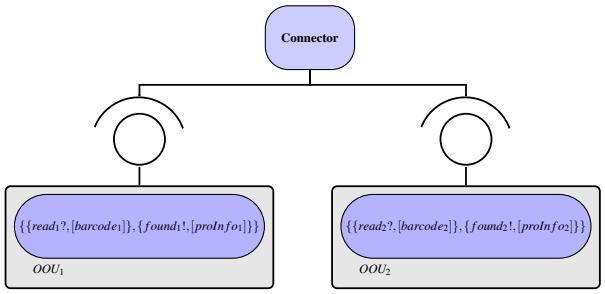

Fig. 3. Composition of two increments.

Incremental construction in PUTRACOM can be summarized in one definition:

Definition 16 (incremental construction)**.** An *incremental construction in PUTRACOM* could be a tuple $PCM_{inc} = (INC,\ basic,\ \Gamma,\ trail,\ inc_{next})$, where:

- INC. It is a finite set of components or composite components.
- $basic$. The first $inc \in INC$ of construction is $basic$.
- Γ is a set of connectors.
- $trail$. is a *non-empty* set of synthesized increments at the current level of construction.
- $inc_{next} \in INC$. Is a component or a composite component to be added to *trail* for building the next level of construction.

In composing increments, preserving interaction behavior is essential for adding new increments without defects. To ensure this, the sets of E^{Obs} must be disjoint, and both the trail and new increments must be free of event collisions before integration. Once confirmed, we can check the conditions for preserving behavior and synchronization.

Definition 17 (behavior preservation)**.** Consider that $basic$ is the first step and $trail$ is the next level of construction of system S. Let the relation between these two be $basic \subseteq trail$. Then, inc_{next} is allowed to be added into the system S as an increment if and only if $basic \subseteq trail \subseteq inc_{next}$ where $inc_{next} \in INC$.

The definition states that at each construction level, behavior and enforced synchronization must remain unchanged. To ensure this, we use the concept of isolated interactions, which clarify which interactions will be checked for preserving system behavior. An isolated interaction is defined based on the concept of a neighborhood.

Definition 18 (neighborhood)**.** Let Π represent a set of possible interactions, and let π be an interaction (or a set of interactions) within Π. The neighborhood of π, denoted by D_π, is defined as a subset of Π that includes all possible interactions containing π. Formally, this can be expressed as:

$$D_\pi = \{U \subseteq \Pi \mid \pi \in U\}$$

where U represents any subset of Π that includes the interaction π. A neighborhood includes a group of interactions that are similar to a specific interaction but still belong to the larger set of all interactions.

Definition 19 (isolated interaction)**.** Let $\pi \in \Pi$ represent an interaction (or $\pi \subseteq \Pi$ denote a subset of interactions), and let D_π be the neighborhood of the interaction π. An isolated interaction, denoted by $islt_\pi$, is defined as a subset of D_π that contains no other interactions from Π. Formally, this can be expressed as:

$$islt_\pi = \{d \in D_\pi \mid (\Pi \cap d) \setminus \{\pi\} = \emptyset\}$$

This means that the set $islt_\pi$ includes only those elements d in D_π for which there are no other interactions from Π present.

A trace function is employed to generate potential interactions and simulate transitions between $trail_i$ and inc_{next}.

Definition 20 (local interactions)**.** Consider a target system $Target$ such that $\{trail, inc_{next}\} \in Target$. If inc_{next} can be incremented to $trail$, denoted by $trail \otimes inc_{next}$, to facilitate the next level of construction, then the set of local interactions for $trail \otimes inc_{next}$, denoted by Π_{loc}, is defined as:

$$\Pi_{loc} = \{\pi_{islt}\langle trail\rangle \cup \pi_{inc_{next}} \cup \{\pi_{islt}\langle trail\rangle \otimes \pi_{inc_{next}}\}\}$$

To check whether the next level of increment contains the enforced synchronization over the transition of the trail, trace containment checking (TCC) [11] has been adopted. Especially, adding a new increment leads to more interaction than what the trail has already enforced. By adopting TCC, the set of interactions that have been added will be checked. The following TCC is extended to accommodate the requirements of this research.

Definition 21 (trace containment)**.** Consider a target system $Target$ such that $\{trail, inc_{next}\} \in Target$. The increment inc_{next} can be integrated into $trail$, denoted by $trail \otimes inc_{next}$, to facilitate the next level of construction if there exists a TCC satisfying the following conditions:

- For all $\pi \in \Pi_{\{trail \otimes inc_{next}\}}$, it holds that $\bowtie \notin \pi$.
- For all $\pi \in \Pi_{\{trail \otimes inc_{next}\}}$, there exists a $\pi_{islt} \in \Pi$ such that $\pi_{islt} \precsim \pi$.

The first condition states that all interactions lack collision events, while the second ensures that isolated interactions remain unchanged from the trail.

In the following, Theorem 1 presents a preservation checking method for $trail_i$ in incremental PUTRACOM. Both sufficient and necessary conditions are presented.

Theorem 1. Consider the target system $Target$ such that $\{trail, inc_{next}\} \in Target$. Let inc_{next} incremented to $trail$ as defined in definition 21, $L = trail \otimes\ inc_{next}$. Suppose Π_{loc} be the local interaction of L as defined in the definition 20. Then, adding inc_{next} to $trail$ preserves the enforced synchronization on the interactions if and only if the following proposition is hold $\forall \pi \in \Pi_{loc}$, $\bowtie \notin \pi$ and $\pi_{islt} \sqsubseteq \pi$.

proof of sufficiency. Let an interaction $\ell = \{\pi \mid \pi \in \Pi_{loc}\}$. Then, ℓ is proven to be a trace containment of local interactions over L by induction. First, we know $\pi = (n,\ p)$, thus:

- For $n \in N$, we have $N \subseteq (E^{obs} \setminus E^I) = E^O$. Then, from condition 2 of the definition 21, we have $\bowtie \notin E^O_L$. Hence $\bowtie \notin \ell$.
- For $p \in P$, we have $P \subseteq (E^{obs} \setminus E^O) = E^I$. Then, from condition 2 of the definition 21, we have $\bowtie \notin E^I_L$. Hence $\bowtie \notin \ell$.

Second, based on definition 20, we know $\Pi_{loc} = \{\pi_{islt}\langle trail\rangle \cup \pi_{inc_{next}} \cup \{\pi_{islt}\langle trail\rangle \otimes \pi_{inc_{next}}\}\}$. Then, for $\pi \in \Pi_{loc}, \exists \pi_{islt} \in \Pi_{loc}$ such that $\pi_{islt} \sqsubseteq \pi$. Hence $\pi_{islt} \sqsubseteq \ell$.

Therefore, ℓ is a trace containment, and adding inc_{next} to *trail* preserves the enforced synchronization on the interactions.

proof of necessity. Let Υ is an alternating trace containment over the local interactions of Π_{loc}. Consider ξ be an observable trace over L from $s_{initial}$ and s_{reach} is a reachable state via interaction (n,p) on ξ. The set of interactions that leads to s_{reach} called $(n,p)_{reach}$. Then, we prove $\bowtie \notin (n,p)_{reach}$ and $\pi_{islt} \sqsubseteq (n,p)_{reach}$ by induction on the trace ξ. Firstly, when $\xi = \lambda$, the two conditions of the theorem hold. Next, suppose the conditions $\forall (n,p)_{reach} \in \Pi_{loc}$, $\bowtie \notin (n,p)_{reach}$ and $\pi_{islt} \sqsubseteq (n,p)_{reach}$ hold for any ξ.

For checking the first condition of the theorem:

For $n \in \{env \cup E^O_L\}$ and $p \in \{env \cup E^I_L\}$, if $\exists A_L \xrightarrow{(n,p)} B_L$ is a $(n,p)_{reach}$ by ξ, then $\bowtie \notin (n,\ p)_{reach}$, (Definition **??**, Condition 1) Because $(n,\ p) = \pi$, then we have $\bowtie \notin \pi_{reach}$.

For checking the second condition:

- For $\forall n \in \{env \cup E^O_{trail}\}$ and $\forall p \in \{env \cup E^I_{trail}\}$, if $\exists A_L \rightarrow B_L$ which is reachable by $(n,p)_{reach}$ in ξ, then $(n,p)_{reach} \in \Pi_{islt}\langle trail\rangle$. Hence, we have $\pi_{islt} \sqsubseteq (n,p)_{reach}$.
- For $\forall n \in \{env \cup E^O_{inc_{next}}\}$ and $\forall p \in \{env \cup E^I_{inc_{next}}\}$, if $\exists A_L \rightarrow B_L$ which is reachable by $(n,p)_{reach}$ in ξ, then $(n,p)_{reach} \in \Pi_{inc_{next}}$. Hence, still we have $\pi_{islt} \sqsubseteq (n,p)_{reach}$.
- For $\forall n \in \{env \cup E^O_{\{\pi_{islt}\langle trail\rangle \otimes \pi_{inc_{next}}\}}$ and $\forall p \in \{env \cup E^I_{\{\pi_{islt}\langle trail\rangle \otimes \pi_{inc_{next}}\}}\}$, if $\exists A_L \rightarrow B_L$ which is reachable by $(n,p)_{reach}$ in ξ, then $(n,p)_{reach} = \{\pi_{islt}\langle trail\rangle \otimes \pi_{inc_{next}}\}$. Hence, we have $\pi_{islt} \sqsubseteq (n,p)_{reach}$.

Therefore, applying Υ over L implies $\forall \pi \in \Pi_{loc}$, $\bowtie \notin \pi$ and $\pi_{islt} \sqsubseteq \pi$ for any reachable trace over L.

VI. Incremental Verification

Verification is essential in system construction for identifying design obstacles and providing counterexamples to locate errors. It begins with generating the system's state space (SS), followed by checks for local properties, global properties, and deadlock-freeness.

1) Verification Process: The verification process checks the system's required properties and generates counterexamples for violations. In PUTRACOM, verification is integrated into the construction process, allowing the next increment only after verifying the previous one. However, model-checking faces limitations in industrial contexts due to the exponential growth of SS with the number of processes and states (SSE). This research proposes a technique to mitigate SSE's impact:

- Incremental verification after each addition, avoiding rechecking already verified parts.
- Eliminating hidden parts of increments and focusing on external behavior to reduce SS size.

2) State space generation: The system's SS must be generated before verification. PUTRACOM supports global and local SS generation, which resembles that in CPN [12] as a directed graph of reachable states and events. Detailed SS generation is omitted due to space constraints.

3) Verifying local properties in PUTRACOM: Each component in PUTRACOM must verify its local properties without assumptions due to their fixed behavior. Properties must be maintained during composition and when adding increments. The following theorem specifies how to prove the properties φ of inc_i in the trail.

Theorem 2. Given that inc_i is an increment in *trail* with respect to behavior preservation, local properties φ of inc_i hold on *trail* if firstly they hold on the local SS of inc_i.

Proof. Let L_{local} is the local SS of *trail*. Consider that $s \in S_{trail}$ is a reachable state in *trail* via trace sequence ζ such that s does not satisfy the set of local properties φ. It can be shown by $\zeta_s \not\models \varphi$. We know the observable trace is consistent with trace sequence $\zeta^{Obs} \preceq \zeta$. Therefore, if any violation against the properties is detected in ζ then its ζ^{Obs} has violation as well. Let $\zeta_{trail} = \varepsilon(trail)$ be a trace sequence over $L_{local}(trail)$ produced by the trace function $\varepsilon(trail)$. Since the trace sequence on *trail* includes the observer trace of all the composed increments $\{inc_1, ..., inc_n\}$ in *trail*, then ζ_{trail} may consist of $\zeta^{Obs}_{inc_i}$ and the violation appears in the $L_{local}(trail)$ as well. As a conclusion, a satisfying φ in *trail* can be detected in local SS inc_i, thus the theorem holds.

The theorem indicates that a set of local properties φ of inc_i is preserved in the trail that includes inc_i. Thus, in order to check and prove such propert in the trail, it must first be proven in the local SS of inc_i.

4) Verifying global properties in PUTRACOM: System-global properties are the set of properties that are related to the global SS of the system. Generally, it involves observable events with multiple increments in *trail*. Essentially, the global properties of the trail should be checked over a clausal normal form, in which clauses are disjunctions of observable events of increment in the trail through the set of connectors Γ_i. Its formal definition is given below.

Definition 22. Consider *trail* as the current level of a given system with a set of increments inc_i and Γ_j, $1 \leq i, j \leq k$, a global property of trail check over conjunction of following disjunction:

$$e^{obs}_1 \vee e^{obs}_2 \vee ... \vee e^{obs}_k$$

such that e^{obs}_k is an observable event of an increment $inc_i \in trail$ and $\forall n, m : 1 \leq n, m \leq k, n \neq m$ *implies* $inc_n \neq inc_m$.

This definition restricts each increment to at most one observable event, simplifying the integration of events from

other increments. Since the equation uses a conjunction of E^{obs} clauses, each can be proven independently and reused for future verification if no new events are introduced.

To verify global properties of a composite component, it is common to check each state and construct the SS as the Cartesian product of component states, which risks SS explosion. Instead, hidden events are eliminated, focusing on the SS without states hidden from other components or the environment. Local property violations impact global properties through observable events, which are critical for interactions in *trail*. Thus, the relevant part for global properties is the set of interactions Π, as proven in the theorem below.

Theorem 3. *Given inc_i is an increment added to trail with respect to behavior preservation, and a set of local properties called φ. Let w is a derived global SS of trail with respect to the elimination of the hidden events. Then φ holds on the w if for all $\pi \in Pi_t rail$, $\exists i : 1 \leq i \leq k, \pi_i \models \varphi_i$.*

Proof. Suppose an interaction $\pi \in trail$ is reachable via trace sequence ξ such that $\pi \not\models \varphi$. We know that an interaction in PUTRACOM can be simply denoted by $\pi = e^o \vee E^I$ and can be extended by $e^o_i \vee e^i_1 \vee e^i_2 \vee ... \vee e^i_k$ for $j : 1 \leq i \leq k$. If at least one of these events violates the properties φ, the *trail* does not holds the global properties φ, $trail \not\models \varphi$. Therefore, to check the global properties of *trail* its interactions must be checked.

This theorem, together with Theorem 2, facilitates the separation of concerns between proving local and global properties. By encapsulating and removing hidden events of increments, it reduces the potential SS of the system. Furthermore, due to the fixed behavior of components, the composition and verification of increments are assumption-free.

5) Deadlock-freeness preservation: Deadlock-freeness verification checks for the presence of activities in any reachable trace of the system. A deadlock indicates that parts or the entire system cannot progress. In PUTRACOM, a deadlock occurs when no admissible events exist for a state or a set of states, halting progress in an RTS or network of RTSs. The formal definition of deadlock-freeness in PUTRACOM is given below.

Definition 23. Let C is considered as a set of RTSs and $Comp = \{c_0, C, E, \Delta\}$ is a composition of C through a connector where E is a set of events and Δ is a set of transitions. Then, $Comp$ is deadlock-free for all $c \in C \vee Comp$, $E_{admis}(c) \neq \emptyset$.

Definition 24. Consider $Comp = \{c_0, C, E, \Delta\}$ is a composition of a set of components C through connector Γ. Let ξ be a given trace of $Comp$ and $s \in S$ is a reachable state via ξ. A state s is considered a deadlock state if $E_{admis}(s) = \emptyset$. $Comp$ can be said to be deadlock-free if, in any possible trace in the $Comp$, no deadlock has been reached.

Once all states in the component have an admissible event, it means that there is progress in the component. However, if there is any state in the component that does not have at least one admissible event to make progress on it, deadlock happens.

VII. Experimental Results

This section evaluates the proposed approach for feasibility in incremental construction, verification, efficiency in reducing SS, and addressing the SSE problem. It includes subsections on the case studies and tools, followed by implementation details and evaluation results.

A. Case Studies

a) Common Component Modeling Example (CoCoME): is a supermarket trading system model for sales, purchases, and inventory management, and widely used for evaluating component models [13], [14]. CoCoME has a cash desk line which processes payments with subcomponents like cash boxes, cash desk controller, cash desk app, Cah desk GUI, print controller, PoS terminal, and interface named *connectorIf*) facilitate communication among the cash desk, store, and enterprise. A store can manage multiple cash desks through a store server that oversees inventory.

b) Dining Philosophers Case Study: It shows how state space can expand exponentially with more philosophers. Seated around a table, each philosopher alternates between thinking and eating, using two shared forks. If all philosophers pick up the left fork at once, a deadlock occurs.

c) Automatic Machine Teller case study (ATM): enables banking transactions without needing human assistance. The system includes three main components: the user, the ATM, and the bank network, with subcomponents like authentication, balance check, cash withdrawal, deposit, help, and maintenance.

B. Implementation Tool

CPN Tools is a graphical platform for simulating CPNs, offering features such as the Create and Simulate Palette for implementing various systems and interactions, the Hierarchy Palette for constructing systems incrementally, and the State-Space Palette to calculate, check, and analyze the state space for properties like deadlock freedom while generating reports. CPNs are a formal modeling language for describing systems. They have been applied in various fields, including robotics [15], or communication networks [16]. A CPN includes places (representing states), transitions (actions or events), and arcs (connections between places and transitions). Each place is assigned a color, indicating the type of resources or objects it represents. Tokens represent the system's state and are placed in the CPN's places. Transitions fire when input places have enough tokens, moving them to output places and changing the system's state. CPNs can include guards and invariants to constrain transition firing, using an inscription language.

CPN is defined as a tuple $cpn = (P, T, R, C, \Sigma, EXPR, W, G, Init)$ where:

- P is a finite set of places;
- T is a finite set of transitions, where $T \cap P = \emptyset$;
- R is a finite set of directed arcs, where $R \subseteq (P \times T) \cup (T \times P)$;
- C represents a set of colors (types), and Σ is a function that maps places to color sets, i.e., $\Sigma : P \to C$. Colors in CPNs represent the types of tokens that places can hold;

- *EXPR* defines net inscriptions, such as guards, arc expressions, and initial markings, typically specified using an inscription language like CPN ML;
- W is the arc expression function $W : R \rightarrow EXPR$, assigning an expression to each arc $r \in R$;
- G is the guard function $G : T \rightarrow EXPR$, assigning a guard to each transition $t \in T$;
- *Init* is the initial marking function $Init : P \rightarrow EXPR$, specifying the initial distribution of tokens in each place.

Both non-incremental and incremental constructions of PUTRACOM can be modeled in this tool.

C. Simulation PUTRACOM by the CPN model

To simulate PUTRACOM using CPN, we start by listing all states, events, guards, and transitions in the model, facilitated by the trace function ε. Once the model elements are identified, we translate them into corresponding elements in the CPN tool, as outlined in Algorithm 1 (Fig. 4). The variable *steps* in Algorithm 1 represents traces of states, events, guards, and transitions from the trace function. Each state $s \in S$ corresponds to a place $p \in P$, and each transition Δ maps to a transition $t \in T$ in the CPN model with two directed arcs. Events $e \in E$ act as tokens, while guards $g \in G$ are conditions for each place. The variable *EXPR* holds expressions for guards, color sets, and markings, differentiating input, output, and internal events, which are then added to transitions. The OOU handles external interactions. Each CU must be encapsulated. Transitions that produce output events generate tokens for the OOU. Trace containment checking (TCC) is simulated in CPN Tools using the ASK-CTL library and SML, enabling CTL-like logic.

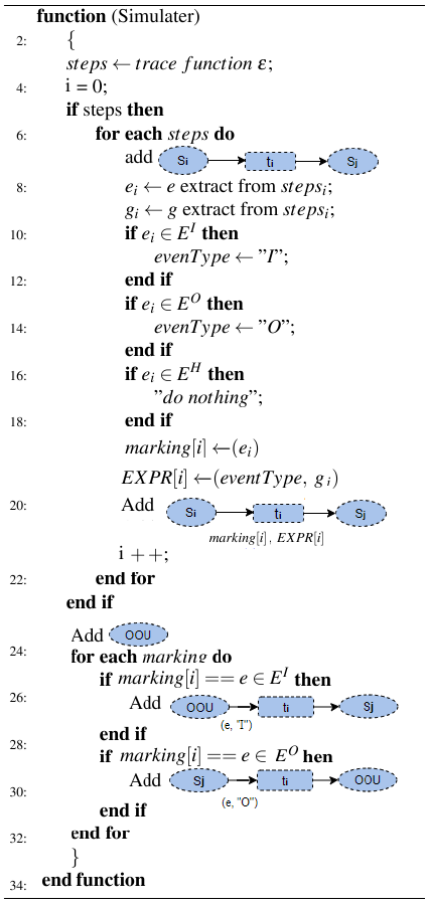

Fig. 4. Algorithm 1: Simulation of PUTRACOM with CPN

D. Evaluation

This section presents experimental results for the aforementioned case studies implemented using the proposed construction technique in CPN Tools. Evaluations were conducted on a Core(TM) i5-3337U CPU @ 1.80 GHz with 4GB of RAM.

a) CoCoME: In the evaluation of CoCoME, firstly the local SS of each component is generated, and then the global SS of the composite components will be calculated incrementally. We firstly verified each component mentioned in VII-A separately. the cash boxes and cash desk controller into a composite component named Composite Component 1, which we verified. Subsequently, we incrementally added the remaining components, resulting in Composite Components 2 through 4. The tables in this section report the number of states, indicating the amount of saved SS in memory. The status reflects whether the verification was fully completed or if a partial SS explosion occurred for both non-incremental and incremental approaches.

The system was verifiable with up to 20 cash desk lines and stores, but non-incremental verification failed early, as shown in Table I, with most results being partial and indicating an SSE. This was due to the need to generate all state spaces (SS) of atomic components and their compositions in memory, causing early SSE as the number of components increased.

In contrast, incremental construction successfully generated the SS by omitting the component's CU, verifying only the newly added increments. The PUTRACOM approach simplifies this by allowing fixed-behavior CUs, which are decoupled from other components, to be excluded. With reduced SS, the verification process is faster and less labor-intensive, making incremental construction more efficient. CPN Tools can produce counterexamples at each verification level, enabling early error detection and enhancing system safety.

b) Dining philosophers: Table II compares the verification results for the dining philosophers problem using the proposed method, CPN Tools, and NuSMV. In this experiment, we assess the amount of memory utilized. Incremental verification completes with lower time and memory consumption, while CPN and NuSMV face the SSE problem. Verification time is longer for dining philosophers due to substantial SS in interaction components (philosophers and forks). Omitting hidden parts did not significantly reduce SS. In CoCoME, diverse components interact through OOU and connectors, allowing hidden parts to be omitted, resulting in faster verification times.

c) Automatic machine teller: Table III shows the ATM case study results using three verification methods. In this experiment, we assess the amount of memory utilized. The enumerative CPN and NuSMV methods could not verify even 50 ATMs, while the proposed incremental method achieved nearly linear performance. The ATM case study outperformed

TABLE I. RESULTS OF COCOME LINE SIMULATION

Component	No. of States (CDL=1)	Status	No. of States (CDL=5)	Status	No. of States (CDL=10)	Status	No. of States (CDL=20)	Status
Cash box	$\approx 10^4$	full/full	$\approx 10^6$	partial/full	$\approx 10^7$	partial/full	$\approx 10^{17}$	partial/full
Cash box controller	$\approx 10^2$	full/full	$\approx 10^3$	partial/full	$\approx 10^2$	partial/full	$\approx 10^4$	partial/full
Cash Desk App	$\approx 10^2$	full/full	$\approx 10^3$	partial/full	$\approx 10^2$	partial/full	$\approx 10^4$	partial/full
Print Controller	$\approx 10^2$	full/full	$\approx 10^3$	partial/full	$\approx 10^2$	partial/full	$\approx 10^4$	partial/full
Cash Desk GUI	$\approx 10^2$	full/full	$\approx 10^3$	partial/full	$\approx 10^3$	partial/full	$\approx 10^5$	partial/full
POS	$\approx 10^2$	full/full	$\approx 10^3$	partial/full	$\approx 10^2$	partial/full	$\approx 10^5$	partial/full
connectorIF	$\approx 10^2$	full/full	$\approx 10^3$	partial/full	$\approx 10^2$	partial/full	$\approx 10^3$	partial/full
Composite Component 1	$\approx 10^4$	full/full	$\approx 10^7$	partial/full	$\approx 10^3$	partial/full	$\approx 10^{18}$	partial/full
Composite Component 2	$\approx 10^5$	full/full	$\approx 10^9$	partial/full	$\approx 10^{10}$	partial/full	$\approx 10^{20}$	partial/full
Composite Component 3	$\approx 10^5$	full/full	$\approx 10^9$	partial/full	$\approx 10^{10}$	partial/full	$\approx 10^{20}$	partial/full
Composite Component 4	$\approx 10^7$	full/full	$\approx 10^{10}$	partial/full	$\approx 10^{10}$	partial/full	$\approx 10^{20}$	partial/full

TABLE II. RESULTS OF DINING PHILOSOPHERS

	Time (min)			Memory (MB)		
Scale	Incremental	Num CPN	NuSMV	Incremental	Num CPN	NuSMV
40	6	47.15	28	13	500	60
100	33	NA	644	29	NA	498
500	209	NA	NA	64	NA	NA
1000	349	NA	NA	112	NA	NA
1500	514	NA	NA	194	NA	NA

the dining philosophers due to having three main components (user, ATM, and bank) with sub-components, whereas the dining philosophers have only two. The hidden parts of each ATM could be omitted after verification, unlike the philosophers' problem, where most state space is tied to interactions. The ATM study results were similar to CoCoME, due to the number of evaluated ATMs and interactions.

VIII. RELATED WORKS

Addressing SSE is a critical challenge in model-checking for CBSD. While model-checking is a valuable tool for verifying system properties like deadlock-freedom, it faces significant scalability issues due to the exponential growth in the number of states to be analyzed [17]. This challenge has led to the development of various techniques aimed at mitigating SSE, which can be categorized into five main approaches: heuristics and probabilistic methods [17], [18], state space reduction techniques [19], compositional verification [20], memory optimization [21], and bottom-up verification [22]. A particular subset of the aforementioned verification methods is more suitable for addressing the SSE problem in CBSD, such as compositional verification and bottom-up approaches.

The compositional verification, particularly the assume-guarantee reasoning approach, decomposes a system into smaller subcomponents and verifies them independently [3]. However, compositional verification introduces new challenges, such as finding optimal ways to decompose systems and deriving accurate assumptions [23]. Additionally, circular dependencies between components can complicate verification, as addressing mutual interdependencies requires defining sound and complete rules, which is often difficult [24]. These limitations reduce the applicability of compositional approaches, especially in complex CBSD systems where component interactions are tightly coupled. Our work tackles these challenges with a bottom-up, incremental approach using a fully decoupled component model. This eliminates dependencies between components, removing the need for system decomposition, assumptions, or managing circular dependencies, making our method more practical for CBSD.

Bottom-up approach, which includes techniques like on-the-fly model-checking and incremental verification. On-the-fly model-checking verifies system properties by generating and checking individual paths one at a time, thereby reducing memory usage by not storing the entire state space [25]. However, this method can lead to inefficiencies, as previously verified paths might need to be regenerated if deleted, resulting in redundant checks and longer counterexamples. In contrast, our technique leverages the event-hiding feature of PUTRA-COM to retain essential verification information, avoiding the need to regenerate paths and thereby improving efficiency.

Incremental verification, which verifies systems as they are constructed, offers another solution for handling SSE. It has been explored in several domains, including rule-based systems, expert systems, and embedded systems [26]. However, most incremental techniques have been developed for non-component-based systems, and those tailored for CBSD remain rare. For example, Bensalem et al. [22] proposed an incremental construction and verification framework for component-based systems, using the BIP (Behavior, Interaction, Priority) model. While their work presents a valuable incremental construction approach, its reliance on symbolic methods (e.g., binary decision diagrams) and extremely relies on the variables ordering, which may limit the verification process. Our work is based on explicit verification methods,

TABLE III. RESULTS OF ATM

	Time (min)			Memory (MB)		
Scale	Incremental	Num CPN	NuSMV	Incremental	Num CPN	NuSMV
50	8	NA	NA	28.4	NA	NA
100	23	NA	NA	67.2	NA	NA
200	54	NA	NA	96.9	NA	NA
500	206	NA	NA	$\approx$ 1G	NA	NA

which are more scalable and easier to apply to large systems.

IX. CONCLUSION AND FUTURE WORK

This research introduces a novel technique for incremental construction and verification in CBSD, aimed at early detection of deadlocks during system development. The approach ensures behavior preservation and synchronization at each increment, verifying safety properties like deadlock freedom and promptly generating counterexamples. The method effectively handles complex CBSD systems, avoiding the SSE problem by excluding previously verified computation units with fixed behavior. This reduces the state space, memory use, and verification effort. Results confirm the method's ability to lower verification complexity and identify potential issues early. Future work includes developing a standalone PUTRACOM application and conducting experiments on real-world systems with larger state spaces for deeper analysis and comparison, enhancing understanding of the method's strengths and limitations.

ACKNOWLEDGMENT

We sincerely thank the Ministry of Higher Education Malaysia and the University Tunku Abdul Rahman for the generous grant #Project: UTAR/IPSR/RMC/UTARRF/2024-C1/F01 that supported this research. Our appreciation also goes to Dr. Kung Kiu Lau, Simone Di Cola, and Cuong M. Tran for their valuable assistance and for sharing their X-MAN tool.

REFERENCES

[1] K.-K. Lau and Z. Wang, "Software component models," *IEEE Transactions on software engineering*, vol. 33, no. 10, pp. 709–724, 2007.

[2] D. Beyer and A. Podelski, "Software model checking: 20 years and beyond," in *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*. Springer, 2022, pp. 554–582.

[3] F. Nejati, A. A. A. Ghani, N. K. Yap, and A. Jaafar, "Handling state space explosion in component-based software verification: A review," *IEEE Access*, 2021.

[4] Y. Jin, "Compositional verification of component-based heterogeneous systems," Ph.D. dissertation, University of Adelaide, SA, AU, 2004.

[5] R. E. Fairley and M. J. Willshire, "Iterative rework: the good, the bad, and the ugly," *Computer*, vol. 38, no. 9, pp. 34–41, 2005.

[6] K.-K. Lau, K.-Y. Ng, T. Rana, and C. M. Tran, "Incremental construction of component-based systems," in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*. ACM, 2012, pp. 41–50.

[7] F. Nejati, A. A. A. Ghani, N. K. Yap, and A. B. Jafaar, "Putracom: A concurrent component model with exogenous connectors," *IEEE Access*, vol. 6, pp. 15 446–15 456, 2018.

[8] K. Jensen and L. M. K. C. P. Nets, "Modelling and validation of concurrent systems," 2009.

[9] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel *et al.*, "Cocome-the common component modeling example," in *The Common Component Modeling Example*, 2008, pp. 16–53.

[10] L. De Alfaro and T. A. Henzinger, "Interface automata," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 109–120, 2001.

[11] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi, "Alternating refinement relations," in *International Conference on Concurrency Theory*. Springer, 1998, pp. 163–178.

[12] K. Jensen and L. M. Kristensen, *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media, 2009.

[13] V. Cortellessa, D. Di Pompeo, V. Stoico, and M. Tucci, "Many-objective optimization of non-functional attributes based on refactoring of software models," *Information and Software Technology*, vol. 157, p. 107159, 2023.

[14] V. Cortellessa, D. Di Pompeo, and M. Tucci, "Check for updates performance of genetic algorithms in the context of software model refactoring," in *Computer Performance Engineering and Stochastic Modelling: 19th European Workshop, EPEW 2023, and 27th International Conference, ASMTA 2023, Florence, Italy, June 20–23, 2023, Proceedings*, vol. 14231. Springer Nature, 2023, p. 234.

[15] M. Kloetzer and C. Mahulea, "Path planning for robotic teams based on ltl specifications and petri net models," *Discrete Event Dynamic Systems*, vol. 30, no. 1, pp. 55–79, 2020.

[16] M. Raeisi-Varzaneh and H. Sabaghian-Bidgoli, "A petri-net-based communication-aware modeling for performance evaluation of noc application mapping," *The Journal of Supercomputing*, pp. 1–24, 2020.

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

[18] Y. Liu and C. He, "A heuristics-based incremental probabilistic model checking at runtime," in *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 2020, pp. 355–358.

[19] A. Biere, "Bounded model checking," in *Handbook of satisfiability*. IOS press, 2021, pp. 739–764.

[20] K. Ghasemi, S. Sadraddini, and C. Belta, "Compositional synthesis via a convex parameterization of assume-guarantee contracts," in *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, 2020, pp. 1–10.

[21] L. Wu, H. Huang, K. Su, S. Cai, and X. Zhang, "An i/o efficient model checking algorithm for large-scale systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 5, pp. 905–915, 2015.

[22] S. Bensalem, M. Bozga, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, "Component-based verification using incremental design and invariants," *Software & Systems Modeling*, vol. 15, pp. 427–451, 2016.

[23] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke, "Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 2, p. 7, 2008.

[24] K. A. Elkader, O. Grumberg, C. S. Păsăreanu, and S. Shoham, "Automated circular assume-guarantee reasoning with n-way decomposition and alphabet refinement," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 329–351.

[25] S. Xiao, J. Li, S. Zhu, Y. Shi, G. Pu, and M. Vardi, "On-the-fly synthesis for ltl over finite traces," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 7, 2021, pp. 6530–6537.

[26] W. Chen, A. Chiesa, E. Dauterman, and N. P. Ward, "Reducing participation costs via incremental verification for ledger systems," *Cryptology ePrint Archive*, 2020.