Modular Verification of Dynamically Adaptive Systems [*]

Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng[†]
Department of Computer Science and Engineering, Michigan State University
East Lansing, Michigan, USA
zhangji9@cse.msu.edu, hjg@cse.msu.edu, chengb@cse.msu.edu

ABSTRACT

Cyber-physical systems increasingly rely on dynamically adaptive programs to respond to changes in their physical environment; examples include ecosystem monitoring and disaster relief systems. These systems are considered high-assurance since errors during execution could result in injury, loss of life, environmental impact, and/or financial loss. In order to facilitate the development and verification of dynamically adaptive systems, we separate functional concerns from adaptive concerns. Specifically, we model a dynamically adaptive program as a collection of (non-adaptive) *steady-state programs* and a set of adaptations that realize transitions among steady state programs in response to environmental changes. We use Linear Temporal Logic (LTL) to specify properties of the non-adaptive portions of the system, and we use A-LTL (an adapt-operator extension to LTL) to concisely specify properties that hold during the adaptation process. Model checking offers an attractive approach to automatically analyzing models for adherence to formal properties and thus providing assurance. However, currently, model checkers are unable to verify properties specified using A-LTL. Moreover, as the number of steady-state programs and adaptations increase, the verification costs (in terms of space and time) potentially become unwieldy. To address these issues, we propose a *modular model checking* approach to verifying that a formal model of an adaptive program satisfies its requirements specified in LTL and A-LTL, respectively.

*This work has been supported in part by NSF grants CCF-0541131, CNS-0551622, CCF-0750787, IIP-0700329, and CCF-0820220, Army Research Office W911NF-08-1-0495, and the Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, and a grant from Michigan State University's Quality Fund.

†Please contact this author for all correspondences.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Software/Program Verification - Formal methods, Model checking, Reliability, Validation*

General Terms

Design, Verification, Reliability

Keywords

Dynamic Adaptation, Reliability, Autonomic Computing, Global Invariants, Formal Specification, Verification, Modular Model Checking.

1. INTRODUCTION

Cyber-physical systems increasingly rely on dynamically adaptive programs to respond to changes in their physical environment; examples include ecosystem monitoring and disaster relief systems. These systems are considered high-assurance since errors during execution could result in injury, loss of life, environmental impact, and/or financial loss. In this paper, we focus on the modeling and verification phases of an aspect-oriented software development approach to dynamically adaptive programs. In order to manage the complexity of analyzing dynamically adaptive systems, we leverage concepts used to promote and facilitate *modular reasoning* [13, 25, 28], including those specific for aspect-oriented software [7], we separate the business (or functional) logic from the adaptive logic. Specifically, we model a dynamically adaptive program as a collection of (non-adaptive) *steady-state programs* and a set of adaptations that realize transitions among steady state programs in response to environmental changes. This separation of concerns enables us to reduce the complexity for modeling, development, and verification purposes. We use Linear Temporal Logic (LTL) to specify *global invariants* that describe properties to be satisfied by the adaptive program throughout its execution, and *local properties* that represent properties to be satisfied by a specific steady-state program. Additionally, we use A-LTL, an adapt-operator extension to LTL that we previously developed [33], to concisely specify *transitional properties* that hold *during* the adaptation process. Model checking offers an attractive approach to automatically analyzing models for adherence to formal properties. As the number of steady-state programs and adaptations increase, however, the verification costs (in terms of space and time) potentially become unwieldy. To that end, in this paper, we propose a *modular model checking* approach to verifying that a formal model of an adaptive program satisfies its requirements specified in LTL and A-LTL, respectively. This approach reduces the verification cost by a factor of n, where n is the number of steady-state programs encompassed by the adaptive program.

Recently, model checking has been used to verify properties of adaptive software [1, 19, 31]. All of these approaches verify global invariant properties and are able to verify local properties. However, these approaches face several common challenges: First, when the number of steady-state programs encompassed by an adaptive program is large, the adaptive program is usually too complex to be verified directly. Second, incremental changes to the adaptive program require the entire verification process to be repeated. Third, transitional properties cannot be verified by these approaches. Verifying transitional properties poses a particularly challenging task because an adaptive program may adapt in an infinite number of different sequences of steady state programs. Thus, the number of different transitional properties is also infinite, which makes direct verifications computationally infeasible.

Modular model checking has been used to verify large non-adaptive programs by decomposing the program into smaller verification modules (e.g., [2, 8, 12, 15, 17, 18, 20, 22]). Several of these techniques [8, 17, 18, 20] use the assume-guarantee paradigm [17, 18] to verify the modules. Specifically, the *assume/guarantee paradigm* [17, 18] defines *assumptions* as the conditions of the running environment that are assumed to be true and *guarantees* as the assurances provided by the program module under the assumptions. If the guarantees imply the assumptions, then the program module adheres to the properties being verified.

In this paper, we extended the assume/guarantee paradigm to develop a modular model checking technique to verify that a given adaptive program adheres to its LTL properties (i.e., local properties and global invariants) and A-LTL properties (i.e., transitional properties). Our approach applies a separation of concerns strategy for performing the model checking within the assume/guarantee paradigm. Specifically, we defined *guarantees* as the conditions satisfied by each steady-state program *without adaptation* and *assumptions* as the sufficient conditions that each steady-state program must satisfy in order for the adaptive program to meet the guarantees *with adaptation*. A given state's *obligations* are the union of its assumptions and guarantees. We leveraged the extended version of the assume/guarantee paradigm to develop three modular model checking algorithms that are used to verify global invariants and transitional properties, where local properties are verified as part of the transitional property model checking algorithm.

Our approach, called AMOEBA (Adaptive program MOdular Analyzer) enables developers to verify that adaptive programs adhere to requirements specified in LTL and A-LTL, including transitional properties. Additionally, it reduces the complexity of verifying an adaptive program by a factor of n, where n is the number of steady-state programs. Moreover, additional or modified steady-state programs can be verified in isolation without repeating the verification of the entire adaptive program. As such, we apply the separation of concerns strategy both in the modeling and in the verification process, thereby simplifying both steps. We model and analyze properties of an adaptive TCP routing protocol to illustrate AMOEBA. The remainder of the paper is organized as follows. In Section 2, we overview A-LTL, describe the adaptive TCP routing protocol, and provide additional detail on the verification challenges of adaptive programs. Section 3 introduces a formal model for adaptive programs and describes the marking algorithm used by our model checking algorithms. Section 4 details our model checking algorithms, states important properties of the proposed algorithms, and applies the algorithms to the adaptive TCP routing protocol. Section 5 discusses the scalability of our approach. Related work is described in Section 6. Section 7 concludes this paper and briefly discusses our future directions.

2. SPECIFYING ADAPTIVE SYSTEMS

In this section, we overview A-LTL, describe a simplified version of an adaptive TCP routing protocol, and discuss the verification challenges posed by adaptive programs. The adaptive TCP routing protocol was developed for RAPIDware [27], an Office of Naval Research project that provides concrete applications for adaptive program verification challenges and is also used to illustrate our proposed solution.

2.1 A-LTL

Previously, we introduced the A-LTL (Adapt-operator extended LTL) [33] to specify the properties to be satisfied for adaptive systems. More specifically, we write $\phi \overset{\Omega}{\rightharpoonup} \psi$, where ϕ, ψ, and Ω are three LTL formulae, to mean that an execution initially satisfies ϕ; in a certain state A, it stops being constrained by ϕ, and in the next state B, it starts to satisfy ψ, and the two-state sequence (A, B) satisfies Ω.

In practice, we have found A-LTL to be more convenient than LTL in specifying various adaptation semantics [33]. For example, if we want to express adapting from a program satisfying $\Box(A \rightarrow \Diamond B)$ to a program satisfying $\Box(C \rightarrow \Diamond D)$, then in A-LTL we write

$$\Box(A \rightarrow \Diamond B) \rightharpoonup \Box(C \rightarrow \Diamond D),$$

which directly captures the intent. The equivalent LTL formula is

$$(A \rightarrow \Diamond(B \wedge \Diamond\Box(C \rightarrow \Diamond D)))\,\mathcal{U}\,(\Box(C \rightarrow \Diamond D),$$

which is much more cumbersome and potentially confusing.[1] We have proved that while A-LTL and LTL have the same expressive power, A-LTL is at least exponentially more succinct than LTL in specifying transitional properties (even when $\Omega \equiv \mathit{true}$) [32].

Semantics.

We define A-LTL semantics over both finite state sequences (denoted by "$\models_{fin}$") and infinite sequences (denoted by "$\models_{inf}$").

- Operators ($\rightarrow$, $\wedge$, $\vee$, $\Box$, $\Diamond$, $\mathcal{U}$, !, etc) are defined similarly as those defined in LTL.
- If σ is an infinite state sequence and ϕ is an LTL formula, then σ satisfies ϕ in A-LTL if and only if σ satisfies ϕ in LTL.
- If σ is a finite state sequence and ϕ is an A-LTL formula, then $\sigma \models_{fin} \phi$ if and only if $\sigma' \models_{inf} \phi$, where σ'

[1]It is a common misunderstanding that the *adapt* operator in A-LTL can be simply replaced by the *next* or *until* operator in LTL. We have previously proved [32] that it is not the case.

is the infinite state sequence constructed by repeating the last state of σ.

- $\sigma \models_{inf} \phi \overset{\Omega}{\rightharpoonup} \psi$ if and only if there exists a finite state sequence
  $\sigma' = (s_0, s_1, \cdots s_k)$, and an infinite state sequence $\sigma'' = (s_{k+1}, s_{k+2}, \cdots)$, such that $\sigma = \sigma' \frown \sigma''$, $\sigma' \models_{fin} \phi$, $\sigma'' \models_{inf} \psi$, and $(s_k, s_{k+1}) \models_{fin} \Omega$, where ϕ, ψ, and Ω are A-LTL formulae, and the $\frown$ is the sequence concatenation operator.
  We can use $\Omega = InP_1$ to specify that the adaptation transition must emit from a state within the program P_1. For simplicity, in this paper we assume $\Omega \equiv true$ and write $\phi \rightharpoonup \psi$ instead of $\phi \overset{\Omega}{\rightharpoonup} \psi$. However, our approach also applies to cases where Ω can be an arbitrary LTL formula [32].

In general, in an adaptive program with n steady-state programs $P_1, P_2, ...P_n$, any execution σ_j with a sequence of $(k-1)$ steps of adaptations, starting from P_{j_1}, going through $P_{j_2}, \cdots P_{j_k}$ must satisfy the following transitional property

$$LP_{j_1} \rightharpoonup LP_{j_2} \rightharpoonup LP_{j_3} \cdots \rightharpoonup LP_{j_k}, \text{ where } j_i \neq j_{i+1}.$$

Each LP_i is a local property for program P_i. Note that each different adaptation sequence corresponds to a different transitional property.

2.2 Adaptive TCP Routing

The *adaptive TCP routing protocol* is a network protocol involving adaptive middleware in which a node balances the network traffic by dynamically choosing the next hop for delivering packets. We assume two types of next hop nodes: *trusted* and *untrusted*. The protocol can be implemented for two different configurations: safe and normal. In "safe" configuration, only trusted nodes are selected for packet delivery, and in "normal" configuration, both types are used in order to maximize throughput. Any packet must be encrypted before being transferred to an untrusted node. We consider the program running in the safe and normal configurations to be two steady-state programs P_1 and P_2, respectively.

Figure 1 depicts an elided portion of the finite state machines for the adaptive protocol; parameters have been omitted for brevity. The upper rectangle of Figure 1 depicts P_1. Initially, P_1 is in the ready1 state, in which, P_1 may receive a packet and move to state received1. At this point, P_1 searches for a trusted next hop and moves to state routed1 when one is found. Then P_1 sends the packet to the trusted next hop and returns to the ready1 state. The lower rectangle in Figure 1 illustrates P_2. The ready2 and received2 states are similar to those in P_1. In state received2, the next hop may be either an untrusted or a trusted node. If the next hop is a trusted node, then P_2 moves to the routed2-safe state. If the next hop is not trusted and the packet is not encrypted, then P_2 moves to the routed2-unsafe state. From state routed2-unsafe, P_2 encrypts the packet and goes to the routed2-safe state. Four adaptive transitions are defined between P_1 and P_2: a1, a2, a3, and a4. The adaptive routing protocol uses the adaptive transitions to switch between steady state programs in response to changing resource levels because P_1 has less throughput than P_2.

The global invariants and local properties of the adaptive program are specified in linear temporal logic (LTL) [29]. For the adaptive routing protocol, a global invariant is that

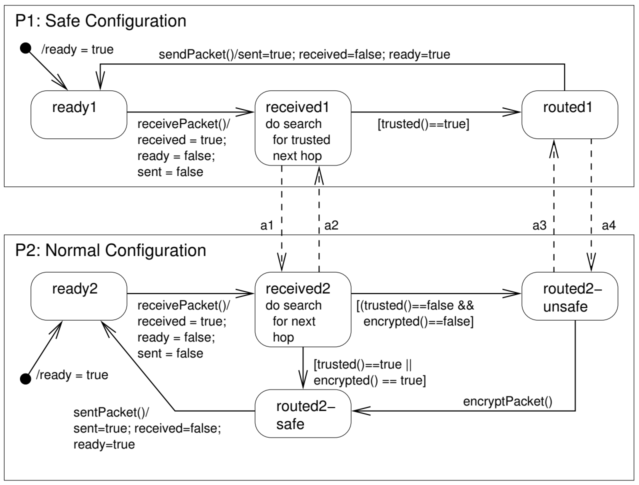

Figure 1: Case study: adaptive routing protocol

the program should not drop any packet throughout its execution, i.e., after it receives a packet, it should not receive the next packet before it sends the current packet. Formally in LTL:

$$inv = \Box(received \Rightarrow (!ready\, \mathcal{U}\, sent)).$$

Additionally, we define local properties for each steady state program. For P_1, we require the system to never use an untrusted (unsafe) next hop. Formally in LTL, we write

$$LP_1 = \Box(!unsafe).$$

where $unsafe \equiv trusted() == false$. For P_2, we require the system to encrypt a packet before sending the packet if the next hop is not trusted. Formally in LTL, we write

$$LP_2 = \Box(unsafe \Rightarrow (!sent\, \mathcal{U}\, encrypted)).$$

For an execution of an adaptive program, if it adapts (i.e., the control flow changes) among the steady-state programs of the adaptive program, then the execution must sequentially satisfy the corresponding local properties of the steady-state programs in the same order.

In the adaptive routing protocol, we express the transitional property that must be satisfied by executions adapting from P_1 to P_2 with the A-LTL formula $LP_1 \rightharpoonup LP_2$, and the transitional property that must be satisfied by executions adapting from P_1 to P_2 and back to P_1 with the A-LTL formula $LP_1 \rightharpoonup LP_2 \rightharpoonup LP_1$, etc.

2.3 Verification Challenges

Several model checking techniques have been proposed for adaptive systems. We overview work in this area with a focus on the types of properties analyzed and the time and space complexity of the approaches.

Global invariants. Allen *et al* [1] used model checking to verify that an adaptive program adapting between two steady-state programs satisfy certain global properties. While they do not explicitly address adaptations of n-plex adaptive programs (for $n > 2$), a straightforward extension could be to apply *pairwise model checking* between each pair of steady-state programs, separately. A drawback of the pair-wise extension is that it requires n^2 iterations of model checking for a program with n steady-state programs.

A solution proposed by Magee [26], called the *monolithic* approach [8], treats an adaptive program as a general program and directly verifies the adaptive program against its global invariants. As with many model checking techniques, the monolithic approach encounters the state explosion problem: it is mostly limited by the large amount of memory required by the computation [9]. Also, the monolithic approach is not designed for incremental adaptive software development or verification, and therefore cannot reuse verification results.

Transitional Properties. Verifying transitional properties is an even more challenging task. Since executions of an adaptive program may adapt within its set of steady-state programs in an infinite number of different sequences, the number of different transitional properties is also infinite. Therefore, it is impossible to verify each transitional property separately. To the best of our knowledge, no existing approaches address the transitional property verification problem defined in this paper.

To address the above problems, we propose a modular model checking approach for adaptive programs against their global invariants and transitional properties that not only reduces verification complexity by a factor of n, where n is the number of steady-state programs, but also further reduces verification cost by supporting verification of incrementally developed adaptive software.

3. FORMAL MODELS

In this section, we first introduce a formal model for adaptive programs. Specifically, we use a finite state machine to represent an adaptive system. Then we introduce several notations, algorithms, and a data structure needed to support the modular model checking. Specifically, a key part of our modular model checking approach is the marking algorithm used to annotate each state with guarantees and assumptions. The marking algorithms typically used for analyzing non-adaptive programs are not sufficient because: (1) they do not support A-LTL properties and (2) it is assumed that the initial states of the program all satisfy the same property, whereas the initial state of each of our steady-state programs must satisfy a potentially different local property. Therefore, in order to build a modular model checker for adaptive programs: (1) we translate LTL and A-LTL properties to partitioned normal form, the notation used to express obligations; (2) we extend the property automata construction algorithm to support A-LTL properties; and (3) we create a new marking algorithm specifically for adaptive programs.

Simple and Adaptive Programs.

Given a set of atomic propositions AP, a *finite-state machine* (FSM) is a tuple $M = (S, S_0, T, L)$, where S is a set of states, transitions $T : S \times S$ is a set of state pairs, where $(s, t) \in T$ represents that there is an arc from s (the predecessor) to t (the successor). The initial state set $S_0 \subseteq S$ is a subset of the states. The function $L : S \rightarrow 2^{AP}$ labels each state s with a set of atomic propositions that are evaluated *true* in s. We represent the states, the transitions, the labels, and the initial states of a given labeled transition system M with $S(M)$, $T(M)$, $L(M)$, and $S_0(M)$, respectively. An FSM is an *extended FSM* (EFSM) if it does not contain a deadlock state.

We define the *composition* of two programs $comp(P_i, P_j)$ to be a program with all the states and transitions in P_i and P_j, and with initial states coming from only P_i:

$$\begin{aligned} comp(P_i, P_j) &= (S, S_0, T, L), \text{ where} \\ S &= S(P_i) \cup S(P_j),\ T = T(P_i) \cup T(P_j), \\ L &= L(P_i) \cup L(P_j), \text{ and } S_0 = S_0(P_i). \end{aligned}$$

The *comp* operation can be recursively extended to accept a list of programs:

$$comp(P_{i_1}, P_{i_2}, \cdots P_{i_n}) = comp(comp(P_{i_1}, P_{i_2}), P_{i_3}, \cdots, P_{i_n}).$$

Similarly, we define the *union* of two programs $union(P_i, P_j)$ to be a program with all the states, transitions, and initial states from P_i and P_j.

$$\begin{aligned} union(P_i, P_j) &= (S, S_0, T, L), \text{where} \\ S &= S(P_i) \cup S(P_j),\ T = T(P_i) \cup T(P_j), \\ L &= L(P_i) \cup L(P_j),\ S_0 = S_0(P_i) \cup S_0(P_j). \end{aligned}$$

The *union* operation can be extended to accept a list of programs:

$$union(P_{i_1}, P_{i_2}, \cdots P_{i_n}) = union(union(P_{i_1}, P_{i_2}), P_{i_3}, \cdots, P_{i_n}).$$

A *simple adaptive program* $SA_{i,j}$ from program P_i to P_j includes the source program P_i, the target program P_j, and the adaptation set $A_{i,j}$ that comprises the intermediate states and transitions connecting P_i to P_j. Formally, we define the simple adaptive program from program P_i to P_j to be the composition

$$SA_{i,j} = comp(P_i, P_j, A_{i,j}).$$

We formally model an *n-plex adaptive program* as an EFSM that contains n steady-state programs $P_1, P_2, \cdots, P_n$, each of which is an EFSM, and there exist adaptations among the P_is. Thus, an *n-plex adaptive program M* contains the union of all the states, transitions, and initial states of n steady-state programs and the corresponding adaptation sets. Formally:

$$M = comp(union(P_1, \cdots p_n), union(A_{1,2}, A_{1,3}, \cdots A_{n,n-1})).$$

An execution of an n-plex adaptive program M is an infinite state sequence $s_0, s_1, s_2 \cdots$ such that $s_i \in S(M)$, $(s_i, s_{i+1}) \in T(M)$, and $s_0 \in S_0(M)$ (for all $i \geq 0$). A non-adaptive execution is an execution $s_0, s_1, s_2 \cdots$, such that all its states are within one program $s_i \in P_j$, for all s_i and some P_j. An adaptive execution goes through one or more adaptation transitions, and two or more steady-state programs.

For an A-LTL/LTL formula ϕ, an execution sequence σ of an adaptive program satisfies ϕ if and only if $\sigma \models \phi$. Conventionally, we say a state s of an adaptive program satisfies a formula ϕ (i.e., $s \models \phi$) if and only if all execution paths initiated from s satisfy ϕ. And an adaptive program M satisfies ϕ (i.e., $M \models \phi$), if and only if all its initial states satisfy ϕ. In addition, for convenience, we define a formula mapping function $\Psi : S_0 \rightarrow$ *A-LTL/LTL* that assigns each initial state a formula. We say M satisfies Ψ (i.e., $M \models \Psi$), if for any initial state $s_0 \in S_0(M)$, we have $s_0 \models \Psi(s_0)$.

Algorithms and Data Structure.

For the interested reader, the appendix contains the formal notations, algorithms, and basic data structures that are required by our model checking algorithm. In general, we define an obligation of a state s of a program P to be a necessary condition that the state must satisfy in order for the program to satisfy a given temporal logic formula ρ. The appendix describes the Partitioned Normal Form (PNF) that is used to propagate the obligations for analysis. Intuitively, the algorithm first marks the initial states of P with obligation ρ, then the obligations of each state are propagated to its successor state(s) in such a way that preserves the necessary conditions along the propagation paths. If a state is reachable from the initial states from multiple paths, then the obligations of the state is the conjunction of the necessary conditions propagated to the state along all these paths.

3.1 Property Automaton

Bowman and Thompson's [4] tableau construction algorithm first creates a *property automaton* based on an initial formula ϕ, and then constructs the product automaton of the property automaton and the program. Their approach is suited for verifying that all initial states satisfy the same initial formula. However, our model checking algorithm requires us to mark program states with necessary/sufficient conditions for different initial states to satisfy different initial formulae. Therefore, we extend their property automaton construction algorithm for our purpose as follows.

New property automaton construction algorithm: A property automaton is a tuple (S, S_0, T, P, N), where S is a set of states. S_0 is a set of initial states where $S_0 \subseteq S$. $T : S \rightarrow 2^S$ maps each state to a set of next states. $P : S \rightarrow \mathit{proposition}$ represents the propositional conditions that must be satisfied by each state. $N : S \rightarrow \mathit{formula}$ represents the conditions that must be satisfied by all the next states of a given state.

Given a set of A-LTL/LTL formula Φ, we generate a property automaton $PROP(\Phi)$ with the following features:

- For each member $\phi \in \Phi$, create an initial state $s \in S_0$ such that $P(s) = \mathit{true}$, $N(s) = \phi$.
- For each state $s \in S$, let the PNF of $N(s)$ be $(pe \wedge \mathit{empty}) \vee \bigvee_i (p_i \wedge \bigcirc qi)$, then it has a successor $s'_i \in S$ for each p_i field with $P(s'_i) = p_i$ and $N(s'_i) = q_i$.

A path of a property automaton is an infinite sequence of states $s_0, s_1, \cdots$ such that $s_0 \in S_0$, $s_n \in S$, and $s_i, s_{i+1} \in T$, for all i $(0 \leq i < n)$. We say a path of a property automaton $s_0, s_1, \cdots$, *simulates* an execution path of a program $s'_1, s'_2, \cdots$, if $P(s_i)$ *agrees with* s'_i for all i $(0 < i)$. We say a property automaton *accepts* an execution path from initial state $s \in S_0$, if there is a path in the property automaton starting from s that simulates the execution path. It can be proved [32] that the property automaton constructed above, from initial state $s \in S_0$, accepts exactly the set of executions that satisfy $N(s)$.[2]

3.2 Product Automaton Construction and Marking

Our algorithm handles the case when each initial state of a program P is required to satisfy a different A-LTL/LTL formula. Given a program $P = (S^P, S^P_0, T^P, L^P)$ and a formula mapping function $\Psi : S^P_0 \rightarrow$A-LTL/LTL, we use the following algorithm to mark the states of P with sufficient/necessary conditions in order for P to satisfy Ψ.

A product automaton is defined to be a tuple: $Q = (S^Q, S^Q_0, T^Q)$, where S^Q is a set of states with two fields: $(\mathit{pstate}, \mathit{nextform})$. The *pstate* field represents one state of program P. The *nextform* field contains an A-LTL/LTL formula declaring what should be true in the next state. S^Q_0 is a set of initial states, $S^Q_0 \subseteq S^Q$. T^Q is a set of transitions, $T^Q : S^Q \times S^Q$. Given a program automaton P, and a mapping function Ψ from its initial states to a set of A-LTL/LTL formulae, we generate a product automaton $PROD(P, \Psi)$ as follows:

1. Calculate the relational image of the initial states of P under the mapping function Ψ.
$$\Phi = \{\phi \mid \exists\, s \in S_0,\, \phi = \Psi(s)\}.$$
2. Construct a property automaton $PROP(\Phi)$ with the set of initial formulae Φ using the property automaton construction algorithm introduced in Section 3.1.
3. For each initial state of the program $s_i \in S^P_0$, add a pseudo program state ps_i (a program state that was not originally in the state space S^P) and a transition (ps_i, s_i) to P, and label ps_i with $L^P(ps_i) = \mathit{true}$.
4. For each pseudo program state ps_i, create an initial product state s^Q_0, where $s^Q_0 \in S^Q_0$ and $s^Q_0 = (ps_i, \phi)$.
5. Create the cross product of P and $PROP(\Phi)$ from the above initial states [30, 4].

Marking states: Given a program P and an initial formula mapping function Ψ, we first construct the product automaton $PROD(P, \Psi)$, then we construct the *marking of each program state* $MARK(s)$ to be the set of *nextform* fields of states in $PROD(P, \Psi)$ that correspond to s. Our *marking algorithm* generates the function $MARK$ over selected states. We also applied optimizations to the algorithm so that we do not need to store the entire product automaton. Further details about the algorithm are available in [32].

The markings generated by the marking algorithm contribute to the assumptions and guarantees in our model checking approach. Specifically, the marking of a state contains the necessary conditions that the state must satisfy in order for the program to satisfy Ψ.

Theorem 1. *For a program P with initial states S_0 and an initial formula mapping function Ψ, let $MARK$ be the marking for the states generated using the marking algorithm above, and let θ be the conjunction of the marking of a state s, $\theta = \bigwedge MARK(s)$, then P satisfies Ψ implies that s satisfies θ. That is*

$$(P \models \Psi) \Rightarrow (s \models \bigwedge MARK(s)).$$

Proof. The proof is described in [32]. □

Additionally, for a state s, if s satisfies all the formulae in the marking of s, then all paths starting from $s_i \in S_0$ going through s satisfy $\Psi(s_i)$.

Theorem 2. *For a program P with initial states S_0 and an initial formula mapping function Ψ, using the marking procedure above, for any state s of P, let θ be the conjunctions of all the marking of s: $\theta = \bigwedge MARK(s)$, then s satisfies θ implies all paths of P starting from $s_i \in S_0$, going through s satisfy $\Psi(s_i)$.*

Proof. The proof is described in [32]. □

[2]We ignore the eventuality constraint [30] (a.k.a self-fulfillment [24]) at this point. However, later steps will ensure eventuality to hold in our approach.

3.3 Interface Definition

We use an *interface* structure to record assumptions and guarantees. An interface I of a program is a function from a program state to a set of A-LTL formulae.

$$I : S(P) \rightarrow 2^{\text{A-LTL/LTL}}.$$

We can compare two interfaces I_G and I_A with an $\Rightarrow$ operation, which returns *true* if and only if for all states s, the conjunction of the formulae corresponding to s in I_G implies the conjunction of the formulae corresponding to s in I_A.

$$I_G \Rightarrow I_A \equiv \forall s : S(P), I_G(s) \Rightarrow I_A(s).$$

We also define a special *top* interface $\top$, which will serve as the initial value in the model checking algorithms presented in the next section:

$$\top = \lambda(x : S(P)).\mathit{true}.$$

4. MODULAR VERIFICATION

Next, we present the new modular model checking algorithms that can be used to analyze adaptive programs. Recall that the general approach is to verify that: (1) each steady state program adheres to its local properties; (2) the global invariants hold for the adaptive program regardless of adaptations; and (3) when the program adapts within its steady-state programs, the corresponding transitional properties are satisfied.

Formal Verification Problem Statement.

Assume that we are given an n-plex adaptive program M with steady-state programs $P_1, P_2, \cdots, P_n$, and adaptation sets $A_{i,j}$, for some i, j ($i \neq j$). Also assume that we are given a local property ϕ_i for each steady-state program, and global invariant properties INV for the n-plex adaptive program M, all written in LTL. We want to verify the following properties by model checking:

- For an arbitrary execution σ_i initiated from program P_{i_1}, with $k-1$ ($k \geq 2$) times of adaptation through $P_{i_2}, P_{i_3}, \cdots P_{i_k}$ where $i_j \neq i_{j+1}$ satisfies
  $$\phi_{i_1} \rightharpoonup \phi_{i_2} \cdots \rightharpoonup \phi_{i_k},$$
  that is sequentially satisfying $\phi_{i_1} \cdots \phi_{i_k}$.
- Any execution of M with k ($k \geq 0$) times of adaptation satisfies INV.

4.1 Modular Model Checking Algorithms

In the following, we present three modular model checking algorithms for adaptive programs and then apply them to the verification of the adaptive routing protocol. The first algorithm checks whether a simple adaptive program satisfies its transitional property. The second algorithm extends the first algorithm in order to check the transitional properties of an n-plex adaptive program. The third algorithm checks the global invariants of an n-plex adaptive program. All three algorithms have four key steps:

1. **Verify base conditions**: Use a traditional model checker to verify that each steady-state program adheres to a set of base conditions.
2. **Compute guarantees**: Use the marking algorithm to annotate each state of each steady-state program with guarantees that are satisfied by those states when there is no adaptation. These guarantees can be inferred from the base conditions and the conditions identified as true for each state in the original model. AMOebA stores the guarantees in interface structure I_G.
3. **Compute assumptions**: Use the marking algorithm to annotate each state of the steady-state programs with assumption that must be satisfied by the state in order for the adaptive program to satisfy the guarantees. AMOebA stores the assumptions in interface structure I_A.
4. **Compare guarantees with assumptions**: Compare interface I_A to interface I_G. If the guarantees imply the assumptions, then the process returns success. Otherwise, it returns with a counterexample.

4.1.1 Simple Adaptive Programs

We first introduce the modular model checking procedure for a simple adaptive program. Given a source program P_i, a target program P_j, an adaptation set $A_{i,j}$, a source local property ϕ_i, a target local property ϕ_j, and an adaptive constraint $\Omega_{i,j}$, the algorithm determines whether the adaptation from P_i to P_j through $A_{i,j}$ satisfies $\phi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$, that is, changes from satisfying ϕ_i to satisfying ϕ_j.

ALGORITHM 1: Transitional properties for simple adaptive programs

```
input $P_i$, $P_j$: EFSM
input $A_{i,j}$: FSM
input $\phi_i, \phi_j, \Omega_{i,j}$: LTL
output ret: boolean
local $I_A, I_G$: Interface
begin
0. Initialize : Initialize two interfaces.
    $I_A := \top$
    $I_G := \top$
1. Verify base conditions :
        • Verify programs $P_i$ and $P_j$ against properties $\phi_i$ and $\phi_j$ locally using traditional LTL model checking.
2. Compute guarantees :
        • Construct marking $MARK$ by running the marking algorithm on $P_j$ with initial formula $\phi_j$.
        • Calculate the state intersection $tos_i$ of $A_{i,j}$ and $P_j$, where $tos$ stands for target of outgoing adaptation states.
            $tos_i := S(A_{i,j}) \cap S(P_j)$
        • Construct interface $I_G$ such that the conditions associated with states in $tos_i$ are the same as their markings in $MARK$, and the conditions associated with states not in $tos_i$ are $true$:

            $I_G := \lambda(x : State).(\textbf{if } x \in tos_i$
                                  $\textbf{then } MARK(x)$
                                  $\textbf{else true endif })$

3. Compute assumptions :
        • Construct marking $MARK'$ by running the marking algorithm on $P_i$ with initial formula $\phi_i$.
        • Calculate the state intersection $sos_i$ of $P_i$ and $A_{i,j}$, where $sos$ stands for source of outgoing adaptation state.
        • Construct marking $MARK''$ by running the marking algorithm on $A_{i,j}$ with the initial formula mapping function $\Psi$ as follows:
            (a) For each $s \in sos_i$, a formula $(x \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j)$ is a conjunct of $\Psi(s)$ iff $x \in MARK(s)'$.
        • Construct interface $I_A$ such that

            $I_A := \lambda(x : State).(\textbf{if } x \in tos_i$        (1)
                                  $\textbf{then } MARK''(x)$
                                  $\textbf{else true endif })$

4. Compare guarantees with assumptions :
    Compare $I_G$ and $I_A$, $ret := I_G \Rightarrow I_A$.
end
```

4.1.2 *N-plex Adaptive Programs*

This algorithm extends the previous algorithm to a general n-plex adaptive program M. Given a set of steady-state programs P_i, a set of adaptation sets $A_{i,j}$, a set of local properties ϕ_i, and a set of adaptive constraints $\Omega_{i,j}$, the algorithm determines

1. For all $i \neq j$, whether the adaptation from P_i to P_j through $A_{i,j}$ satisfies $\phi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$, that is, changes from satisfying ϕ_i to satisfying ϕ_j.
2. Whether any execution from P_{j_1}, going through $P_{j_2}, P_{j_3}, \cdots P_{j_k}$ ($j_i \neq j_{i+1}$) satisfies $\phi_{j_1} \overset{\Omega_{j_1,j_2}}{\rightharpoonup} \phi_{j_2} \cdots \phi_{j_k}$, that is, sequentially satisfying $\phi_{j_1} \cdots \phi_{j_k}$.

This algorithm repeatedly applies **Algorithm 1** to each single adaptation from P_{j_1} to P_{j_k}. As some of the marking and comparison operations overlap, the algorithm is optimized by removing the redundancies.

```
ALGORITHM 2: Transitional property for n-plex
             adaptive programs
input $P_i$ ($i = 1 \cdots n$): EFSM
input $A_{i,j}$ ($i,j = 1 \cdots n$): FSM
input $\phi_i, \Omega_{i,j}$ ($i,j = 1 \cdots n$): LTL.
output ret: boolean
local $I_A, I_G$: Interface
begin
0. Initialize : Initialize two interfaces.
    $I_A := \top$
    $I_G := \top$
    For each program $P_i$
    1. Verify base conditions :
        • Verify programs $P_i$, against properties $\phi_i$ locally
          with traditional LTL model checking.
    2. Compute guarantees :
        • Generate markings $MARK$ by running the marking
          algorithm on $P_i$ with initial formula $\phi_i$.
        • Calculate the state intersection $tis_i$ of $P_i$ and $A_{j,i}$
          for all $j \neq i$, where $tis$ stands for target of incoming
          adaptation states.
        • Update interface $I_G$ with $I'_G$ such that the condi-
          tions associated with states in $tis_i$ are the conjunc-
          tion of their values in $I_G$ and their markings in
          $MARK$, and the conditions associated with states
          not in $tis_i$ are those in $I_G$;

              $I'_G := \lambda(x : State).(\textbf{if } x \in tis_i$
                  $\textbf{then } MARK(x) \wedge I_G(x)$
                      $\textbf{else } I_G \textbf{ endif })$

    3. Compute assumptions :
        • For each $j \neq i$, do
          (i) Calculate the state intersection $sos_{i,j}$ of $P_i$ and
              $A_{i,j}$.
          (ii) Construct marking $MARK'$ by running the mark-
              ing algorithm on $A_{i,j}$ with initial formula map-
              ping function $\Psi$ as follows:
              For each $s \in sos$, a formula ($x \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$) is a conjunct
              of $\Psi(s)$ iff $x \in MARK(s)$.
          (iii) Calculate the state intersection $tos_{i,j}$ of $P_j$ and
              $A_{i,j}$.
          (iv) Update interface $I_A$ with $I'_A$ such that

              $I'_A := \lambda(x : State).(\textbf{if } x \in tos_{i,j}$
                  $\textbf{then } MARK'(x) \wedge I_A(x)$
                      $\textbf{else } I_A(x) \textbf{ endif })$

    4. Compare guarantees with assumptions :
        Compare $I_G$ and $I_A$, $ret := I_G \Rightarrow I_A$.
end
```

4.1.3 *Global Invariants*

Given a set of steady-state programs P_i, a set of adaptation sets $A_{i,j}$, and a global invariant INV, the global invariant model checking algorithm determines whether all executions of an n-plex adaptive program satisfy the global invariant INV.

```
ALGORITHM 3: Global invariants
input $P_i$ ($i = 1 \cdots n$): EFSM
input $A_{i,j}$ ($i,j = 1 \cdots n$): FSM
input $INV$: LTL
output ret: boolean
local $I_A, I_G$: Interface
begin
0. Initialize : Initialize two interfaces.
    $I_A := \top$
    $I_G := \top$
 For each program $P_i$
    1. Verify base conditions :
        • Verify programs $P_i$ against global invariants $INV$
          with traditional LTL model checking.
    2. Compute guarantees :
        • Construct the program composition
          $C_i = comp(P_i, union(A_{i,1}, A_{i,2}, \cdot A_{i,n}))$.
        • Construct marking $MARK$ by running the marking
          algorithm on $C_i$ with initial formula $INV$.
        • Calculate the target states of all incoming transi-
          tions $tis_i = P_i \cap A_{j,i}$ for all $j \neq i$.
        • Update interface $I_G$ with $I'_G$ such that the condi-
          tions associated with states in $tis_i$ are the same
          as the conjunction of their values in $I_G$ and their
          markings in $MARK$, and the conditions associated
          with states not in $tis_i$ are the values in $I_G$;

              $I'_G = \lambda(x : State).(\textbf{if } x \in tis_i$
                  $\textbf{then } MARK(x) \wedge I_G(x)$
                      $\textbf{else } I_G \textbf{ endif })$

    3. Compute assumptions :
        • Calculate the state intersection $tos_i$ of $A_{i,j}$ and $P_j$
          for all $j \neq i$
        • Update interface $I_A$ with $I'_A$ such that

              $I'_A = \lambda(x : State).(\textbf{if } x \in tos_i$
                  $\textbf{then } MARK(x) \wedge I_A(x)$
                      $\textbf{else } I_A(x) \textbf{ endif })$

    4. Compare guarantees with assumptions :
        Compare $I_G$ and $I_A$, $ret := I_G \Rightarrow I_A$
end
```

4.2 Adaptation Model Checking Process

Given an n-plex adaptive program, the overall process to verify its local properties, transitional properties, and global invariants are as follows.

1. Use traditional model checking approach to verify each steady-state program against its local properties and the global invariants locally.
2. Apply **Algorithm 2** to verify transitional properties for the adaptive program.
3. Apply **Algorithm 3** to verify global invariants for the adaptive program.

We illustrate this process by verifying the adaptive routing protocol adheres to its local properties, transitional properties, and global invariants (Section 2).

4.2.1 Applying Algorithm 2

First, we apply **Algorithm 2** to verify the transitional properties for the adaptive routing protocol. Figure 2 depicts the markings that result from verifying the transitional properties for both P_1 and P_2. Specifically, the guarantee markings created in step 2 are prefixed with "$g.*$". The assumption markings created in step 3 are prefixed with "$a.*$". Both sets of markings are annotated in the state in braces.

1. Verify base conditions. We use Spin to model check the local properties, LP_1 and LP_2.

2. Compute guarantees. Next, we use the marking algorithm to annotate each state of the steady-state programs with obligations. We compute guarantees by marking the initial states with local properties.

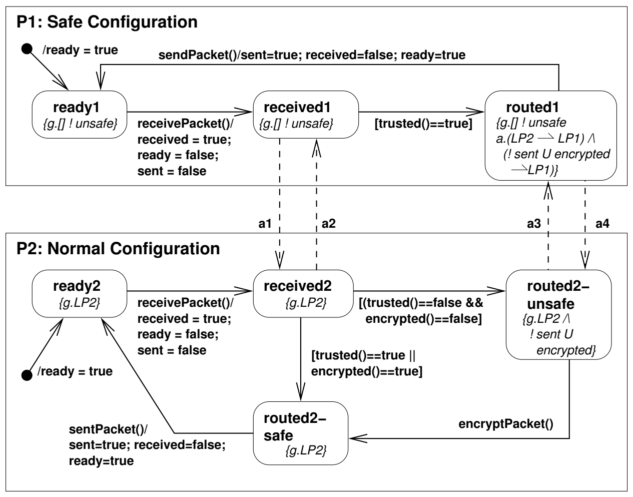
P1: Safe Configuration
/ready = true
sendPacket()/sent=true; received=false; ready=true
ready1 {g.[] ! unsafe}
receivePacket()/ received = true; ready = false; sent = false
received1 {g.[] ! unsafe}
[trusted()==true]
routed1 {g.[] ! unsafe a.(LP2 ⇀ LP1) /\ (! sent U encrypted ⇀LP1)}
a1
a2
a3
a4
P2: Normal Configuration
ready2 {g.LP2}
receivePacket()/ received = true; ready = false; sent = false
received2 {g.LP2}
[(trusted()==false && encrypted()==false]
routed2–unsafe {g.LP2 /\ ! sent U encrypted}
/ready = true
[trusted()==true || encrypted()==true]
routed2–safe {g.LP2}
sentPacket()/ sent=true; received=false; ready=true
encryptPacket()

Figure 2: Markings for transitional properties

3. Compute assumptions. In this step, we start from the guarantee markings generated in the previous step. For each state in a steady-state program P_i with outgoing adaptive transitions going towards program P_j, we generate an obligation $\phi \rightharpoonup LP_j$ from each condition ϕ in its guarantee marking, where LP_j is the local property for P_j. Then we propagate the generated obligations to the states in P_j along the adaptive transitions to form their assumption markings. For example, the guarantee marking for routed2-unsafe is LP_2 and $(!sent\,\mathcal{U}\,encrypted)$. From this marking, we generate obligations $LP_2 \rightharpoonup LP_1$ and $(!sent\,\mathcal{U}\,encrypted) \rightharpoonup LP_1$, respectively. These obligations are propagated to the state routed1, then we create the assumption marking $LP_2 \rightharpoonup LP_1$ and $(!sent\,\mathcal{U}\,encrypted) \rightharpoonup LP_1$ for routed1. We repeat this process for all states in P_1 and P_2 with outgoing adaptive transitions, and the resulting assumption markings are shown in Figure 3, prefixed with $a.*$.

4. Compare guarantees with assumptions. In the adaptive routing protocol, we find that the guarantee for state routed1 (LP_1) does not imply the condition $(!sent\,\mathcal{U}\,encrypted) \rightharpoonup LP_1$ in its assumption. This assumption condition requires the obligation *encrypted* to be satisfied before the adaptation, while the guarantee does not ensure this obligation. Therefore, the model checking for the transitional property fails. As such, we generate a counterexample showing a path that violates the transitional properties. In this example, we return the trace (ready2, received2, routed2-unsafe, routed1). Clearly, the failure is caused by the adaptive transition a3 (from routed2-unsafe to routed1). We remove a3 from the adaptive program and repeat steps (3) and (4). The algorithm returns success.

4.2.2 Applying Algorithm 3

Next, we apply **Algorithm 3** to verify the global invariants. Figure 3 depicts the markings that result from verifying the global invariants for both P_1 and P_2. Specifically, the guarantee markings created in step 2 are prefixed with "$g.*$". The assumption markings created in step 3 are prefixed with "$a.*$".

1. Verify base conditions. We use Spin to determine that both P_1 and P_2 satisfy the invariant *inv* (page 3).

2. Compute guarantees. In the adaptive routing protocol, since P_1 satisfies *inv*, we conclude that state ready1 satisfies *inv*, therefore, we mark ready1 with the obligation *inv*. Then we propagate this obligation to its successor state(s) received1 as follows: First, it satisfies *inv*. Second, since *received* is *true* in the state, it must also satisfy $!ready\,\mathcal{U}\,sent$. Therefore, we mark received1 with obligations *inv* and $!ready\,\mathcal{U}\,sent$. Similarly, we compute the guarantees for the rest of the states.

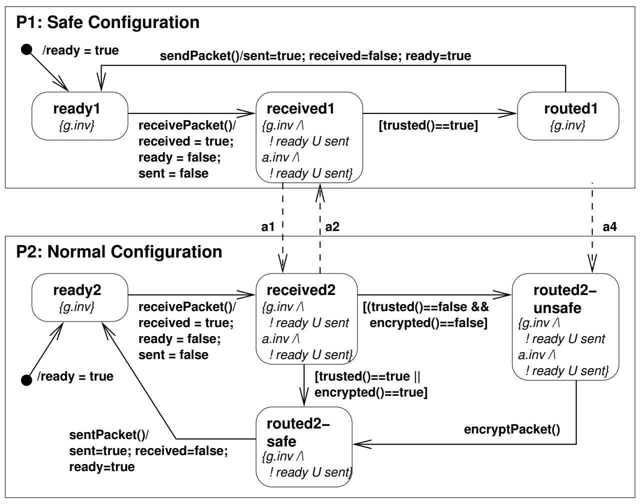
P1: Safe Configuration
/ready = true
sendPacket()/sent=true; received=false; ready=true
ready1 {g.inv}
receivePacket()/ received = true; ready = false; sent = false
received1 {g.inv /\ ! ready U sent a.inv /\ ! ready U sent}
[trusted()==true]
routed1 {g.inv}
a1
a2
a4
P2: Normal Configuration
ready2 {g.inv}
receivePacket()/ received = true; ready = false; sent = false
received2 {g.inv /\ ! ready U sent a.inv /\ ! ready U sent}
[(trusted()==false && encrypted()==false]
routed2–unsafe {g.inv /\ ! ready U sent a.inv /\ ! ready U sent}
/ready = true
[trusted()==true || encrypted()==true]
routed2–safe {g.inv /\ ! ready U sent}
sentPacket()/ sent=true; received=false; ready=true
encryptPacket()

Figure 3: Markings for global invariant *inv*

3. Compute assumptions. In the adaptive routing protocol, we propagate the markings of received1 to state received2 along the adaptive transition a1 and mark received2 with *inv* and $!ready\,\mathcal{U}\,sent$. Our process ensures that the assumption marking includes **exactly** the set of conditions that received2 must satisfy in order for all executions starting from ready1, taking adaptive transition a1, and taking no more adaptations, to satisfy the global invariant *inv*. Similarly, we propagate the marking of routed1 to routed2-unsafe, and from received2 to received1.

4. Compare guarantees with assumptions. Next we compare the guarantees with the assumptions. For each state, if the conjunction of its guarantee marking logically implies all the conditions in its assumption marking (checked automatically), then the process returns success, otherwise, it returns with a counterexample. For example, the guarantee marking for received2 indeed implies the assumption marking for received2. This result implies that all executions starting from ready1, taking adaptive transition a1, with no adaptation afterwards, satisfy *inv*. We perform the comparison on every state of the steady-state programs with incoming adaptive transitions. Successful comparisons guarantee that any execution starting from ready1 or ready2, undergoing one step of adaptation, satisfies *inv*.

4.3 Incremental Model Checking

When a steady-state program P_i of the adaptive program is changed after a successful model checking has been performed, we only need to repeat **Algorithm 2** and **Algorithm 3** on P_i (and/or related adaptations) to determine whether the specification are still satisfied. Assume that if after reapplying marking algorithm on P_i, we compute the interfaces to be I'_A and I'_G, then

- if $I'_G \Rightarrow I_G$ and $I_A \Rightarrow I'_A$, then we have $I'_G \Rightarrow I'_A$, then no more model checking is required;
- if $I_A \not\Rightarrow I'_A$, then the model checking for outgoing adaptations from P_i needs to be repeated;
- if $I'_G \not\Rightarrow I_G$, then the model checking for incoming adaptations to P_i needs to be repeated.

The model checking results for all other parts still apply and therefore are reused.

When a new steady-state program P_{n+1} is incrementally introduced to an n-plex adaptive program after the adaptive program has been successfully verified with our algorithms, we will need to model check P_{n+1} and all the adaptations going into and out of P_{n+1}.

4.4 Claims

The following theorems capture the claims that we make as to how our approach addresses the verification problems presented in Section 2.3.

THEOREM 3. *For a simple adaptive program from P_i to P_j, if **Algorithm 1** returns true, then*

- *All non-adaptive executions within P_i (or P_j) satisfy the local property ϕ_i (or ϕ_j).*
- *All adaptive executions starting from P_i and adapting to P_j satisfy $\phi_i \overset{\Omega_{i,j}}{\rightharpoonup} \phi_j$.*

THEOREM 4. *For an n-plex adaptive program M, if **Algorithm 2**, returns true, then:*

- *All non-adaptive executions within a single steady-state program P_i satisfy the local property of P_i.*
- *Any execution σ_j starting from P_{j_1}, going through $P_{j_2}, \cdots P_{j_k}$ ($j_i \neq j_{i+1}$) satisfies*

$$\phi_{j_1} \overset{\Omega_{j_1,j_2}}{\rightharpoonup} \phi_{j_2} \overset{\Omega_{j_2,j_3}}{\rightharpoonup} \phi_{j_3} \cdots \overset{\Omega_{j_{k-1},j_k}}{\rightharpoonup} \phi_k.$$

THEOREM 5. *For an n-plex adaptive program M, if **Algorithm 3** returns true, then all execution paths of M satisfy the global invariant INV.*

Further details are provided in [32].

5. SCALABILITY AND COMPLEXITY

Briefly, we discuss AMOEBA performance, including performance and scalability. The algorithm introduced in Section 4 stores in memory the guarantee and assumption markings of all the states in the adaptive program, which may occupy a large amount of memory. However, in later steps, only markings of the interface states are used. For the AMOEBA implementation, the required memory space is significantly reduced by storing the markings for only the interface states instead of for all the states during the marking computations [32].

AMOEBA improves scalability of model checking adaptive programs by reducing the time/space complexity of the model checking algorithms. We illustrate this point by comparing our approach to the alternative approaches that we described in Section 2.3, namely, the pairwise approach [1] and the monolithic approach [8, 26]. Assume an n-plex adaptive program M contains steady-state programs $P_1, P_2, \cdots, P_n$. We denote the size of the steady-state program P_i as $\mid P_i \mid$, and we assume all steady-state programs are of similar size $\mid P \mid$: $\mid P \mid \approx \mid P_i \mid$, for all i. We assume that the size of adaptive states and transitions are significantly smaller than the steady-state programs (which we consider a key characteristic of adaptive software). Then we have $\mid M \mid \approx n * \mid P \mid$, where $\mid M \mid$ is the size of M.

The results of comparison are displayed in Table 1, where the key differences between the monolithic approach, pairwise approach, and our approach are bolded. Essentially, the monolithic approach requires the same order of time, but n times more memory space than AMOEBA. The pairwise approach requires the same order of space, but requires n times more execution time than AMOEBA. Additionally, AMOEBA supports the verification of A-LTL properties and can also be further optimized to verify incrementally developed software by reusing existing verification results.

Approach	**Time Complexity**	**Space Complexity**	**A-LTL**
AMOEBA	$O(2^{\mid INV \mid} \mid M \mid)$	$O(2^{\mid INV \mid} \mid P \mid)$	Yes
Monolithic	$O(2^{\mid INV \mid} \mid M \mid)$	$O(2^{\mid INV \mid} \mid \mathbf{M} \mid)$	**No**
Pairwise	$O(\mathbf{n}(2^{\mid INV \mid} \mid M \mid))$	$O(2^{\mid INV \mid} \mid P \mid)$	**No**

Table 1: Model Checking Comparison

6. RELATED WORK

In this section, we discuss existing modular verification techniques of non-adaptive programs and non-modular verifications of adaptive programs.

6.1 Modular Model Checking

Our work has been significantly influenced by several existing modular model checking approaches for non-adaptive programs. In this section, we focus on the analysis of the differences and relationships between our approach and other modular verification approaches.

Krishnamurthi, *et al* [20] introduced a modular verification technique for *aspect-oriented* programs. They model a program as a finite state machine (FSM), where an *aspect* is a mechanism used to change the structure of the program

FSM. An aspect includes a *point-cut designator*, which defines a set of states in the program, an *advice*, which is an FSM itself, and an *advice type*, which determines how the program FSM and the advice FSM should be composed. By model checking the program and the aspect individually, they verify CTL properties of the composition of the program and the aspect. In other work [11, 23], they introduced modular model checking techniques for cross-cutting features, which basically follows the same idea.

Our approach is largely inspired by their approach. The two approaches share the following similarities: (1) The objective of both approaches is to decompose the model checking of a large system into the model checking of smaller modules. (2) We both use the traditional assume/guarantee reasoning developed by Bowman and Thompson. (3) We both define interface states and use conditions to mark these states. Then we reason about the conditions among these states to draw conclusions about the entire system. (4) There are behavioral changes involved in both approaches, although in different ways. (5) Neither of our approaches are complete.

However, despite the similarities, the two approaches also have significant differences, and their approach is not applicable to our verification problems. (1) The most prominent difference is the fundamental difference between the underlying structures for the temporal logics that we handle. CTL is evaluated on states. The classic model checking method for CTL is accomplished by using state marking [10]. Their approach reused the classic CTL state marking algorithm on interface states, which can be considered a natural extension to the basic CTL model checking idea. However, LTL/A-LTL is evaluated on execution paths. Model checking of LTL *cannot* be solved by marking its states. Rather, tableau-based or automaton-based methods are conventionally used for path-based logic model checking. The challenge overcome by our approach is to design a novel marking algorithm to be applied to the interface states for a path-based logic, which has not been previously published in the literature [32]. Our algorithm also deals with eventuality, which is also a challenge for LTL/A-LTL, but not for CTL model checking. (2) Our approaches also differ in the way the system behavior may change. With their approach, one may consider the behavioral change from the base to the feature as an adaptation. They require the execution to change back to the base, which behaves analogously to a stack. However our approach enables arbitrary adaptation sequences, such as an adaptation through a sequence of programs A to B to C to A, etc. (3) Our approaches differ in the definition of modules. In their approach, a module refers to a separate (physical) piece of code, such as the base program or a feature. However, in our approach, each module refers to different behaviors. It may be exhibited by the same piece of code under different modes, or different pieces of code. (4) Finally, our approach supports the verification of A-LTL, which has not been previously addressed.

Henzinger *et al* [15] proposed the Berkeley Lazy Abstraction Software verification Tool (BLAST) to support extreme verification (XV). XV is modeled to be a sequence of program and specifications (P_i, Φ_i), where Φ_i is the specification for the i^{th} version of the program P_i, and Φ_i are non-decreasing, i.e., $\Phi_i \subseteq \Phi_{i+1}$. In order to reduce the cost of each incremental verification when verifying the i^{th} program, they generate an *abstract reachability tree* T_i. When model-checking P_{i+1}, they compare P_{i+1} to T_i to determine the part of P_{i+1}, if any, should be re-verified. Our approach differs in that they verify propositional, instead of temporal logic properties. Also, their approach is for general programs, while our incremental verification is optimized specifically for adaptive programs. We consider their approach to be complementary to ours because, in practice, each steady-state program is developed incrementally from some common base program. Their approach can verify the local properties and global invariants of each steady-state program locally for the base condition verification.

Alur *et al* [2] introduced an approach to model-checking a *hierarchical state machine*, where higher-level state machines contain lower-level state machines. As the same state machine may occur at different locations in the hierarchy, its model checking may be repeated if we *flatten* the hierarchical state machine before applying traditional model checking. Their objective is to reduce the verification redundancy when a lower-level state machine is shared by a number of higher level state machines. Their approach can be applied to optimize our solution in that the steady-state programs may share parts of their behavior (sub-state machines). We can use their approach to reduce the redundancy when verifying the shared behavior.

Many others, including Kupferman and Vardi [22], Flanagan and Qadeer [12], have also proposed modular model checking approaches for different program models. They focus on verifying concurrent programs, where modules are defined to be concurrent threads (processes). We consider their approaches to be orthogonal and complementary to our approach.

6.2 Correctness in Adaptive Programs

Model checking has been applied to verify adaptive behavior by several researchers. Kramer and Magee [19] used *property automata* to specify the properties for the adaptive program and used LTSA to verify these properties. Allen *et al.* [1] integrated the specifications for both the architectural and the behavioral aspects of dynamic programs using the Wright ADL. In our previous work [31], we introduced a model-based adaptive software development process that uses Petri nets to model the behavior, and then uses existing Petri net-based model checking tools to verify these models against interesting properties. None of the above approaches address the verification problem modularly. Compared to these approaches, AMOEBA is less complex, more scalable, and supports not only LTL, but also A-LTL. More details regarding this comparison are given in Section 5.

Many others have also worked on providing assurance to adaptive systems, several of which describe algorithms to ensure that the system is in a *quiescent state* when a component is removed. Chen *et al* [5] proposed a *graceful adaptation protocol* that enables adaptations to be coordinated across hosts transparently to the application. Appavoo *et al* [3] proposed a *hot-swapping* technique, i.e., run-time object replacement. We have previously introduced a *safe adaptation protocol* [34]. These approaches provide safe adaptation protocols based on run-time dependency analysis, instead of model checking approaches as proposed in this paper. Kulkarni *et al* [21] introduced an approach using *proof-lattices* to verify that all possible adaptation paths do not violate global propositional constraints.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a modular approach to model-checking adaptive programs against their global invariants and transitional properties expressed in LTL/A-LTL. A key contribution of this approach is the ability to verify transitional properties which, previously, had not been analyzable. For validation purposes, we have implemented our approach in a prototype model checker AMOebA using C++, and have used the tool to verify a number of adaptive programs.

We note the potential for improving model checking performance by combining our approach with existing techniques [1, 2, 11, 12, 15, 16, 19, 20, 22, 23]. We are investigating strategies to combine our approach with others to further reduce the complexity of adaptive program model checking. Run-time verification of adaptive programs against their global invariants and transitional properties [14] is also part of our ongoing work [6].

8. REFERENCES

[1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[2] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, 2001.

[3] J. Appavoo, K. Hui, C. A. N. Soules, et al. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60, 2003.

[4] H. Bowman and S. J. Thompson. A tableaux method for Interval Temporal Logic with projection. In *TABLEAUX'98, International Conference on Analytic Tableaux and Related Methods*, number 1397 in Lecture Notes in AI, pages 108–123. Springer-Verlag, May 1998.

[5] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing adaptive software in distributed systems. In *Proc. of the 21st International Conference on Distributed Computing Systems*, Mesa, AZ, April 16 - 19 2001.

[6] B. H. C. Cheng, H. J. Goldsby, and J. Zhang. Amoeba-RT: Run-time verification of adaptive software. In *In Holger Giese, editor, Models in Software Engineering Workshops and Symposia at MoDELS 2007*, Nashville, TN, USA, October 2007. Springer Verlag.

[7] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *In Foundations of Aspect Languages*, pages 33–44, 2002.

[8] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *ISSTA'06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 97–108, 2006.

[9] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2-3):275–288, 1992.

[10] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 169–180. ACM Press, 1982.

[11] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 152–163, New York, NY, USA, 2001. ACM Press.

[12] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN 03: SPIN Workshop, LNCS 2648*, pages 213–225. Springer-Verlag, 2003.

[13] B. Hailpern and S. Owicki. Modular verification of concurrent programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–336, New York, NY, USA, 1982. ACM.

[14] K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification*, Paris, France, July 2001.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. Sanvido. Extreme model checking. *Verification: Theory and Practice, Lecture Notes in Computer Science 2772, Springer-Verlag*, pages 332–358, 2004.

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, NJ, USA, 1985.

[17] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[18] B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1-2):47–72, 1996.

[19] J. Kramer and J. Magee. Analysing dynamic change in software architectures: a case study. In *Proc. of 4th IEEE International Conference on Configurable Distributed Systems*, Annapolis, May 1998.

[20] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 137–146, New York, NY, USA, 2004. ACM Press.

[21] S. Kulkarni and K. Biyani. Correctness of component-based adaptation. In *Proceedings of International Symposium on Component-based Software Engineering*, May 2004.

[22] O. Kupferman and M. Y. Vardi. Modular model checking. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 381–401, London, UK, 1998. Springer-Verlag.

[23] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. *ACM SIGSOFT Software Engineering Notes*, 27(6):89–98, 2002.

[24] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107. ACM Press, 1985.

[25] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press and McGraw-Hill, Cambridge, 1986.

[26] J. Magee. Behavioral analysis of software architectures using LTSA. In *Proceedings of the 21st International Conference on Software Engineering*, pages 634–637. IEEE Computer Society Press, 1999.

[27] P. K. McKinley. RAPIDware. http://www.cse.msu.edu/rapidware/. Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan.

[28] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[29] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[30] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st Symposium on Logic in Computer Science*, pages 322–331, Cambridge, England, 1986.

[31] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of International Conference on Software Engineering (ICSE'06)*, Shanghai,China, May 2006.

[32] J. Zhang and B. H. C. Cheng. Modular model checking of dynamically adaptive programs. Technical Report MSU-CSE-06-18, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, March 2006. http://www.cse.msu.edu/ hjg/Zhang06Modular.pdf.

[33] J. Zhang and B. H. C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software (JSS), Architecting Dependable Systems*, 79(10):1361–1369, 2006.

[34] J. Zhang, B. H. C. Cheng, Z. Yang, and P. K. McKinley. Enabling safe dynamic component-based software adaptation. *Architecting Dependable Systems, Lecture Notes in Computer Science*, pages 194–211, 2005.

APPENDIX

A. PARTITIONED NORMAL FORM

In our work, we use the *Partitioned Normal Form* (PNF) [4] notation introduced by Bowman and Thompson to handle obligation propagation. We rewrite each A-LTL/LTL formula into its PNF as follows:

$$(pe \wedge empty) \vee \bigvee_i (p_i \wedge \bigcirc q_i),$$

where pe stands for the proposition where $empty$ is true (i.e., deadlock state), and p_i depicts the proposition where the next state satisfies q_i (i.e., non-deadlock state). Formally, pe, p_i and q_i satisfy the following constraints:

- pe and p_i are all propositional formulae
- p_i partitions *true*, i.e., $\bigvee_i p_i \equiv true$ and $p_i \wedge p_j \equiv false$ for all $i \neq j$.

All A-LTL/LTL formulae can be rewritten in PNF by applying PNF-preserving rewrite-rules [4]. As the adapt operator is not included in the original paper by Bowman and Thompson, we introduce the following A-PNF rewrite-rule:

$$\begin{aligned} \phi \overset{\Omega}{\rightharpoonup} \psi \quad = \quad & (empty \wedge false) \vee \\ & \bigvee_i \bigvee_j (p_i^\phi \wedge p_j^\Omega \wedge pe^\phi \wedge \bigcirc((q_j^\Omega \wedge \psi) \vee (q_i^\phi \rightharpoonup \psi))) \vee \\ & \bigvee_i (p_i^\phi \wedge \neg pe^\phi \wedge \bigcirc(q_i^\phi \rightharpoonup \psi)), \end{aligned}$$

where ϕ and ψ are A-LTL/LTL formulae, and we use superscripts on p_i, q_i and pe to represent the formula from which they are constructed. Since pe and p_i are all propositions, their truth values can be directly evaluated over the label of each single state. Therefore, the obligations of a given state can be expressed solely by a next state formula: the q_i part of a disjunct when the state has successor states, and/or *empty* in case the state is a deadlock state.