Computing (2013) 95:785–816
DOI 10.1007/s00607-013-0295-3

Development of global specification for dynamically adaptive software

Yongwang Zhao · Zhuqing Li · Hualei Shen · Dianfu Ma

Received: 13 March 2012 / Accepted: 9 January 2013 / Published online: 1 February 2013

Abstract As software systems are becoming increasingly complex, they need to dynamically and continually adapt their behavior to changing conditions in the long-term running. There will be large numbers of adaptations in these systems when evolving and the adaptations may be unknowable until system operation. To specify these adaptations, this paper proposes the mode-supported Linear Temporal Logic (mLTL) that is an effective way to describe global specifications of adaptive software. The global specifications are defined for adaptive software as requirements from the perspective of global adapting process. The model checking problem of mLTL is also resolved using Linear Temporal Logic (LTL) and Labelled Transition System Analyser (LTSA). Finally, we provide a prototype implementation for modelling and analyzing adaptive programs, and experimental evaluation shows feasibility and scalability of our approach.

Keywords Autonomic computing · Dynamic adaptation · Formal specification · Verification

Y. Zhao (✉) · Z. Li · H. Shen · D. Ma
National Laboratory of Software Development Environment (NLSDE),
School of Computer Science and Engineering, Beihang University,
Haidian District, Beijing, China
e-mail: zhaoyw@act.buaa.edu.cn

Z. Li
e-mail: lizq@act.buaa.edu.cn

H. Shen
e-mail: shenhl@act.buaa.edu.cn

D. Ma
e-mail: dfma@buaa.edu.cn

Mathematics Subject Classification (2000) 68N30 Mathematical aspects of software engineering (specification, verification, metrics, requirements, etc).

1 Introduction

To decrease cost and time of human supervision for continue operation, computer software must modify its structure and behavior dynamically in response to the changes in its execution environment [41]. Such a modification is commonly referred to as dynamic adaptation. Being highly complex, adaptive programs are generally more difficult to be specified, verified, and validated. Assurance of high dependability of these programs would thus be a great challenge [33]. Correctness of adaptive programs becomes very crucial, especially when they are applied in safety-critical domains. The adaptive program must be ensured that it functions correctly during and after adaptations. Effective and precise specification development and assurance for adaptation are key issues for dependable adaptive software. On the other hand, software systems are becoming increasingly complex systems characterized by thousands of platforms, sensors, decision nodes, etc., connected through heterogeneous wired and wireless networks [38]. They have diverse and changeable requirements, and software failures are unavoidable. Therefore, there will be large numbers of adaptations in these systems when evolving and adaptations may be unknowable until system operation. The transitional specification that focus on the adaptation from one behavior to another of the adaptive software at one moment could not easily deal with the continuous running and evolution of these complex software systems. New specifications that are able to effectively specify adaptive software focusing on the global process of all adaptations are needed. They should be independent on the explicit adapting process and be able to specify continuous adaptations unknowable at design time. In this paper, they are called *global specification* for complex software systems, including the specifications for maintainability, survivability and recoverability, etc. The global specification concerns the temporal relationship of multiple adaptations on the adapting process. Development and assurance of them is a challenge for adaptive software. New effective specifying and verifying approaches are needed for high dependability of these systems.

The adaptive system and its behavior have been formalized to clarify key notions that can help to specify the adaptive system, the context or environment, and the subject [13,14]. Bradbury et al. [12] surveyed numerous research efforts to formally specify the dynamically adaptive software. Most of research works [23,29,33,37] have focused on the structural changes of adaptive software. Few efforts have formally specified the behavioral changes of adaptive software. Zhang et al. [47–49] proposed a model-based development approach for dynamically adaptive software. Biyani and Kulkarni [10] discussed an approach to model and verify dynamic adaptation in distributed systems. These approaches focus on the transitional specification including properties of the source program, the target program and the adaptation between them. Their approaches are explicit to the adapting process of the adaptive program. This leads to the difficulty to specify the adaptations unknowable at design time and the inefficiency for large amount of adaptations. Goal-based modeling [20,24,36,46] has also been used for specifying dynamically adaptive software. A common feature of these works is that they assume that all adaptation choices are known and enumerated at design time. When adaptations in the complex software systems are unknowable at design time, developers could not use these models to specify and analyze adaptive programs.

This paper focuses on formally developing the global specifications for the behavioral adaptation of adaptive programs. Our approach assumes an universal model of adaptive programs. We consider that adaptive programs have different *behavioral modes* for different operation contexts and a non-adaptive program (briefly program) labelled with a name is used to present the software behavior in each mode. The adaptation is the behavioral transition from one mode to another. We use finite state machine (FSM) to formally describe our model by the reason of FSM's universality. Different from the transitional specification, the global specification are temporal properties on sequences of behavioral modes and adaptations. For instance, the property holds on all behavioral modes or will hold on one behavioral mode eventually. To specify the global requirements of adaptive programs, we introduce the mode-supported Linear Temporal Logic (mLTL), a variant of LTL, by which adaptive programs could be specified on the behavioral modes on the global adapting process. We adopt LTL style operators and add mode-related elements for mLTL. mLTL could express the temporal relationship of multiple adaptations on the global adapting process. LTL formulae could be embedded in mLTL to enable specifying properties of the non-adaptive program, and the global specification is constructed by composition of these LTL properties using mLTL operators. To reuse LTL properties predefined on non-adaptive programs and mature model checking tools, we normalize the adaptive programs and propose the global semantic of them according to the mLTL semantics. This enables mLTL model checking based on available LTL model checkers. Then, the mLTL model checking could be done by three steps: (1) extract LTL local properties of the non-adaptive program and translate the mLTL property to an LTL formula, (2) checking local properties on each non-adaptive program, and (3) checking the LTL formula from (1) on a transition system constructed by the global semantic and the result of step (2). We use Finite State Process (FSP) [28] and Labelled Transition System Analyser (LTSA)[1] to implement our approach in a prototype, Modelling and Analysis Tool for Adaptive Programs (MATAP).

A key contribution of our approach is the global specifications for adaptive software. They provide a novel way to specify adaptive programs from a new perspective. Transitional specification is between two explicit non-adaptive programs. To specify the requirements of adaptive programs, it assumes that all adaptation choices are known in advance. To a certain extent, mLTL does not require all possible alternative adaptations to be designed during requirements engineering. mLTL is independent on the global adapting process, and requirements of adaptive programs could be specified by mLTL before the adpating process is known. We believe that it is sufficiently

[1] http://www.doc.ic.ac.uk/ltsa/.

effective for continuous evolution and unknowable behavior when designing complex software systems in the future. Therefore, mLTL is able to specify the program behavior unknowable before system operation and could be applied for runtime verification [7,11,32]. mLTL and other approaches can be used in a complementary fashion to specify adaptive software.

The rest of the paper is structured as follows. In Sect. 2, we state the background and motivate our paper. Section 3 introduces the formal model of our approach. mLTL and its model checking approach are discussed in Sects. 4 and 5 respectively. We introduce our MATAP prototype implementation and evaluate our approach in Sect. 6. Section 7 compares this paper with related works. The last section concludes this paper and discusses future research directions.

2 Background and motivation

To justify our approach, we first introduce what the adaptive program is, and introduce the existing categories of its properties. We also motivate our paper by illustrating a few of basically global properties and analyzing why existing approaches are not effective.

2.1 Adaptive program and specification

A dynamically adaptive program operates in different behavioral modes and transits from one mode to another at runtime in response to changes of context. Therefore, an adaptive program usually contains multiple non-adaptive programs and multiple adaptations connecting these programs [47]. Each of these programs exhibits a different steady-state behavior [1], and operates in a different mode. An adaptation is a projection of the adaptive program behavior from one steady-state behavior to another by transitions. Figure 1 illustrates an adaptive program which has three programs and two adaptations among them.

The correctness of adaptive programs cannot be properly addressed without precisely specifying the requirements for adaptation. Existing researches mostly consider

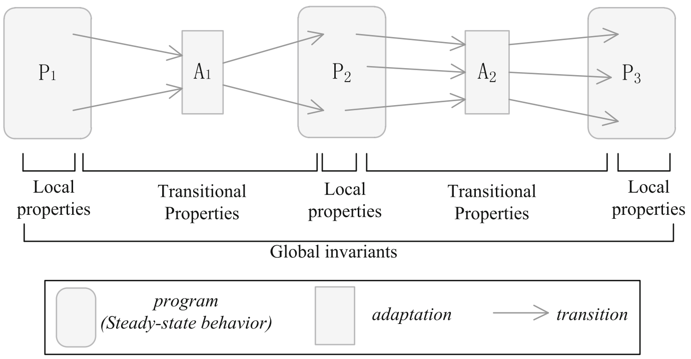


Fig. 1 An adaptive program and its specifications

the transitional specification instead of the global specification. Typically, Zhang et al. [48,49] classified the properties of the adaptive program into three categories as follows. These properties are generally used to define the transitional specification of the adaptive program, not the global specification.

- **Local properties**: properties define the specification of a specific non-adaptive program. They specify the steady-state behavior of an adaptive program operating in a behavioral mode. They may be such properties as safety, liveness, and invariants, etc.
- **Transitional properties**: properties that hold during the adaptation process. They specify the dynamic adaptation between two non-adaptive programs, and should be satisfied during interval state when the adaptive program is being adapted from one behavioral mode to another.
- **Global invariants**: properties to be satisfied by the adaptive program throughout its execution. They consider steady-state behaviors and adaptations as a whole regardless of the adaptations.

The scopes of these categories of properties are presented in the lower part of Fig. 1.

Linear Temporal Logic (LTL) is a widely used temporal logic to specify systems. LTL expresses properties of state paths of programs viewed as sets of executions. LTL formulae over the set AP of atomic proposition are formed according to the following grammar:

$$\varphi ::= \mathbf{true} \mid \alpha \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2 \tag{1}$$

where $\alpha \in AP$. Other operators are defined based on these operators, such as $\vee(or)$, $\rightarrow (imply)$, $\leftrightarrow (coimply)$, $\Diamond(eventually)$ and $\Box(always)$, etc. The local properties can be specified by LTL directly. Verification of these properties is carried out on the state sequence of a non-adaptive program. The global invariants can also be specified by LTL. To specify the transitional properties, Zhang and Cheng [48] proposed the A-LTL by extending LTL with the *adapt* operator ($\overset{\Omega}{\rightharpoonup}$). They formally define A-LTL as follows:

- If ϕ is an LTL formula, then ϕ is also an A-LTL formula.
- If ϕ and φ are both A-LTL formulae, then $\phi \overset{\Omega}{\rightharpoonup} \varphi$ is an A-LTL formula.
- If ϕ and φ are both A-LTL formulae, then $\neg\phi$, $\phi \wedge \varphi$, $\phi \vee \varphi$, $\phi\mathcal{U}\varphi$ are all A-LTL formulae.

The A-LTL semantics are defined as follows:

- If σ is an infinite state sequence and ϕ is an LTL formula, then $\sigma \models \phi$ in A-LTL if and only if $\sigma \models \phi$ in LTL.
- If σ is a finite state sequence and ϕ is an A-LTL formula, then $\sigma \models_f \phi$ iff $\sigma' \models \phi$, where σ' is an infinite state sequence constructed by repeating the last state of σ.
- $\sigma \models \phi \overset{\Omega}{\rightharpoonup} \varphi$ iff there exists a finite state sequence $\sigma' = (s_0, s_1, ..., s_k)$ and an infinite state sequence $\sigma'' = (s_{k+1}, s_{k+2}, ...)$, such that $\sigma = \sigma' \frown \sigma''$, $\sigma' \models_{fin} \phi$, $\sigma'' \models \varphi$, and $(s_k, s_{k+1}) \models_{fin} \Omega$, where ϕ, φ and Ω are A-LTL formulae.
- Other operators are defined similarly as those used in LTL.

Simply saying that an adaptive program satisfies $\phi \overset{\Omega}{\rightharpoonup} \varphi$, if this program initially satisfies ϕ, and in a certain state A, it stops being constrained by ϕ, then in the next state B, it starts to satisfy φ. And the two-state sequence (A, B) satisfies Ω. For instance, in the *adaptive TCP routing protocol* in [49], only trusted nodes are selected for packet delivery in "safe" configuration and any packet must be encrypted before being transferred to an untrusted node in "normal" configuration. The transitional property that must be satisfied by executions adapting from "safe" configuration to "normal" configuration could be expressed by the A-LTL formula $\Box(!unsafe) \overset{true}{\rightharpoonup} \Box(unsafe \rightarrow (!sent\ \mathcal{U}\ encrypted))$.

After introducing the adaptive program and existing approaches to specify them, we will motivate our approach in next subsection by stating the global specification and why the existing approaches are not suitable for this specification.

2.2 Motivation

Software systems are becoming increasingly complex systems characterized by thousands of platforms, sensors, decision nodes, etc., connected through heterogeneous wired and wireless networks [38]. These ultra-large-scale(ULS) systems will be developed and used by a wide variety of stakeholders and requirements may be unknowable until system operation. Software and hardware failures will be the norm rather than the exception, and people will not just be the users but also the elements of the system. These facts cause continuous evolution for systems, and adaptability becomes increasingly important. Assurance of system requirements at the macro level such as maintainability, survivability and recoverability, etc., is a challenge for dynamic adaptation.

The complex software systems will be in service for a long time, and there will be an increasing need to integrate new capabilities while they are operating. The system will be evolving not in phases, but continuously to meet new and modified requirements and to incorporate new technologies. Moreover, the adaptation and its target behavior of the adaptive program may be unknowable until system operation. Therefore, there may be large amount of (unknowable) adaptations in these systems when evolving. We define global specifications as the requirement for adaptive programs from the perspective of adapting process at the macro level. For instance, the degraded system functions dealing with encountering errors will eventually be upgraded after some adaptations. There exist a few of basically global properties for adaptive programs based on the taxonomy of dependable computing [3]:

- **Reachability**: it means that the adaptive program will eventually running as a behavioral mode. Formally, an LTL property will hold on a non-adaptive program eventually when an adaptive program is evolving.
- **Safety**: it means absence of catastrophic behavior for the adaptive program. Formally, an LTL negative property will never hold on any program when the adaptive program is evolving.
- **Maintainability**: it means the ability to undergo modifications and repair. In the adapting process of an adaptive program, if a negative property holds on a

non-adaptive program, which means that the system has some error, it will eventually be repaired on one of its subsequent programs.

- **Integrity**: it refers to the absence of improper system adaptation. There does not exist such a program that a negative property holds on it and another negative property holds on one of its immediately succeeding programs.

After analyzing the global specification, we could find that the approach to specify adaptive programs considering the global process of all adaptations should not require all possible alternative adaptations to be specified during requirements engineering. It needs to support specifying requirements of adaptive programs before the adapting process is known. Second, since there may be a large amount of adaptations in the adaptive program, this approach should be effective and avoid its dependence on the explicit adapting process. It should have operators to traverse the possible adapting processes.

We could find that the three categories of properties in Sect. 2.1 for adaptive programs will be exhausted in this case. A-LTL extends LTL with the *adaptor* to support transitional specification of adaptive programs. But it does not have direct notation support for global specification. It is both too complex and inconvenient for our purposes. First, A-LTL is explicit to the adapting process of the adaptive program [49]. The temporal relationship of multiple adaptations considering the global process of all adaptations is difficult to be specified. Second, A-LTL is not effective for large amount of adaptations. For example, if we want to express a very simple property, the reachability of the behavioral mode satisfied by $SPEC$ in an adaptive program that has three non-adaptive programs and two adaptations, then in mLTL we write $\Diamond_m[SPEC]$ as discussed later, which directly captures the intent. The equivalent A-LTL formula is $(SPEC \overset{true}{\rightharpoonup} true \overset{true}{\rightharpoonup} true) \vee (true \overset{true}{\rightharpoonup} SPEC \overset{true}{\rightharpoonup} true) \vee (true \overset{true}{\rightharpoonup} true \overset{true}{\rightharpoonup} SPEC)$, which is much more cumbersome. If the adaptive program has large amount of non-adaptive programs, A-LTL is exhausted for the reachability. LTL is more cumbersome than A-LTL on this problem, because A-LTL is at least exponentially more succinct than LTL in specifying transitional properties [49]. If we want to express the maintainability or integrity of adaptive programs, the LTL or A-LTL formula will be too complex to be understood.

These reasons lead us to propose the mLTL which is sufficiently effective for continuous evolution and unknowable behavior when designing complex software systems in the future. mLTL is able to specify the program behavior unknowable before system operation. Although LTL is not effective for the global specification, the properties specified by LTL are independent with the state trace of a program and LTL's operators can traverse each state trace. Therefore, mLTL adopts similar operators of LTL. mLTL could express the temporal relationship of multiple adaptations considering the global process of all adaptations. Since mLTL and A-LTL focus on global and transitional specifications respectively, they can be used in a complementary fashion to specify adaptive software.

For the global specification, formal model of adaptive programs should have the ability to distinguish different non-adaptive programs. We use the concept "*mode*", adopted from mode change [9] in real-time system, to present each program which can be considered as a behavioral mode of an adaptive program. Secondly, specification language of dynamic adaptation needs to support *mode*, and its operators are able to traverse mode space of adaptive programs. The next section presents the formal model of dynamic adaptation, and Sect. 4 discusses the mode-supported linear temporal logic.

3 Formal model

In this section, we first introduce a formal model for adaptive programs. We use the FSM to represent an adaptive system. Then we use *Adaptive Communication Protocol* (*ACP*) as an illustrative example to demonstrate the formal model and our specification approach in the following sections.

3.1 Adaptive programs

We consider that an adaptive program consists of multiple non-adaptive programs and adaptations among them. Each of these programs exhibits a different steady-state behavior that can be presented by an FSM [26]. In FSM, actions trigger the transition between program states. The trace sets of actions and paths of states are considered as behavior at runtime. These FSMs present non-adaptive programs at different behavioral mode. When the context changes, adaptive programs will transit their behavior from one FSM to another and these transitions are adaptations.

We use FSM to define the non-adaptive program and adaptive program.

Definition 1 (*Non-adaptive program*) A non-adaptive program $\mathcal{P}$ is a finite state machine. $\mathcal{P}$ is a tuple $\langle v, S, Act, \sigma, s_0, AP, L\rangle$, where

- v: name of this program
- S: a set of states
- Act: a set of actions
- $\sigma : S \times Act \to S$ is a state transition relation
- $s_0 \in S$ is the initial state
- $L : S \to 2^{AP}$ is a labelling function.

The labelling function L maps each state to a set of atomic propositions. $s \in S$ is a state name, and L represents the logic state. $(s, a, s') \in \sigma$ is a transition, where $s, s' \in S$ and $a \in Act$. If $\mathcal{P}$ has a transition (s, a, s'), it means that a is enabled in state s, and execution of action a in state s will lead to state s'. We only consider the deterministic behavior, and if $(s, a, s'), (s, a, s'') \in \sigma$ then $s' = s''$.

To enable mLTL model checking later, we add the name of the non-adaptive program to its states as the *flag proposition* to identify the state's current mode. Thus, labelling function L will maps each state to a proposition to indicate the mode that current state belongs to.

We represent the name, the states, the actions, the transitions, the initial state, and the labelling function of a given program $\mathcal{P}$ with $v(\mathcal{P})$, $S(\mathcal{P})$, $Act(\mathcal{P})$, $\sigma(\mathcal{P})$, $s_0(\mathcal{P})$, $L(\mathcal{P})$, respectively.

The whole behavior of an adaptive program consists of a set of non-adaptive programs and it can also be described by a finite state machine.

Definition 2 (*Adaptive program*) An adaptive program $\mathcal{AP}$ is a finite state machine. $\mathcal{AP}$ is a tuple $\langle A, E, \varphi, \mathcal{P}_0 \rangle$, where

- A: a set of non-adaptive programs, with each of them being a mode, and for each program $\mathcal{P}_i \in A, \mathcal{P}_i = \langle v_i, S_i, Act_i, \sigma_i, s_{0_i}, AP_i, L_i \rangle$
- E: a set of adaptations
- $\mathcal{P}_0 \in A$ is the initial program of $\mathcal{AP}$
- $\varphi : A \times \bigcup_{\mathcal{P} \in A} S(\mathcal{P}) \times E \rightarrow A \times \bigcup_{\mathcal{P} \in A} S(\mathcal{P})$ is the adaptation relation, where $\bigcup_{\mathcal{P} \in A} S(\mathcal{P})$ is the set of all states of programs in A. For each $(\mathcal{P}, s, e, \mathcal{Q}, t) \in \varphi$, $s \in S(\mathcal{P})$ and $t \in S(\mathcal{Q})$.

To conveniently present the adaptation as transitions, we replace the adaptation that is a transition set as $Adaptation$ in Fig. 2 by adaptations, each of which is a transition, such as e_{11}, e_{12}.

The names of all programs in $\mathcal{AP}$ must be different from each other. Adaptation is a switch between two programs. We do not consider the causes of adaptation and only use *adaptation* to model this behavior. An adaptation is an action which switches the behavior from one state in a program to one state in another program. We only consider the deterministic adaptation and if $(\mathcal{P}_i, s, e, \mathcal{P}_j, s'), (\mathcal{P}_i, s, e, \mathcal{P}_k, s'') \in \varphi$ then $\mathcal{P}_j = \mathcal{P}_k$ and $s' = s''$. For determinism, the initial programs set of $\mathcal{AP}$ is a single

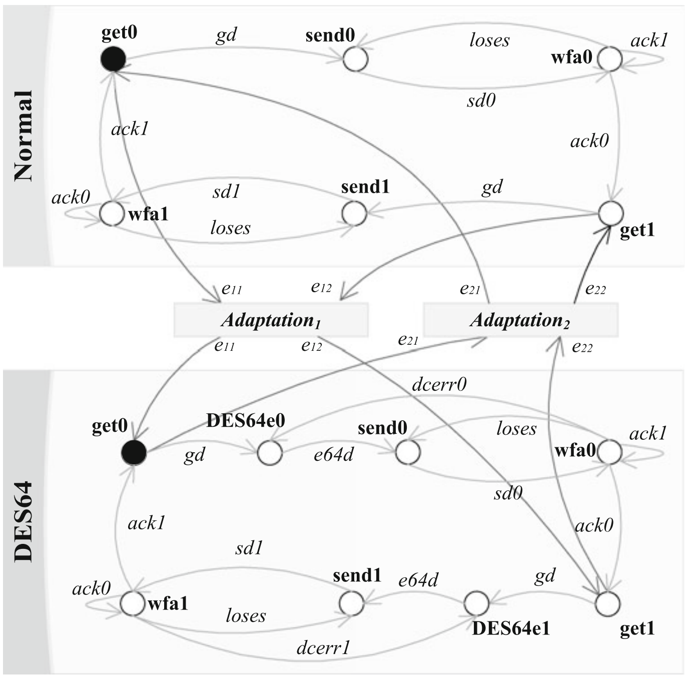


Fig. 2 Adaptation of SENDER by adding/removing DES64e filter

program. Moreover, if $(\mathcal{P}_i, s, e, \mathcal{P}_j, s') \in \varphi$ then $\mathcal{P}_i \neq \mathcal{P}_j$. It means that adaptation from one state to another state on the same program is forbidden.

We represent the non-adaptive programs, the adaptations, the initial program, and the adaptation relations of a given adaptive program $\mathcal{AP}$ with $A(\mathcal{AP})$, $E(\mathcal{AP})$, $\mathcal{P}_0(\mathcal{AP})$, $\varphi(\mathcal{AP})$, respectively.

3.2 The ACP study case

To illustrate our approach clearly in the following sections, a study case is introduced here. With large-scale deployment of wireless communication services and advances in handheld computing devices, distributed applications are sensitive to the heterogeneity of the devices and networks where they are deployed on. Adapting the communication substrate at runtime is one of the key challenges in designing these systems. We illustrate a study case, Adaptive Communication Protocol (ACP) that is a revisited version of the one used for example in [48,49], to demonstrate our approach for dynamic adaptation. We use an alternating bit for reliability. Filters that manipulate the data stream can be inserted or removed dynamically at runtime in response to the external conditions as implemented by MetaSocket [40].

ACP consists of four adaptive programs, SENDER, RECEIVER, S2R, and R2S. We have five filters, a data compression filter (DCM), a data decompression filter (DDM), a DES 64-bit encryption filter (DES64e), a DES 128-bit encryption filter (DES128e), and a DES decryption filter (DESd) capable of DES 64-bit and DES 128-bit decryption. The DCM, DES64e and DES128e filters can be configured on SENDER and others on RECEIVER. S2R and R2S are unreliable communication channels on which message losses may occur. To ensure the performance of ACP, data compression/decompression filters can be inserted and removed with the change in the size of communicating data. The encryption/decryption filters are configured by different security requirement.

The program Normal without filters, the program DES64 with DES64e filer of SENDER, and the adaptation between them are depicted in Fig. 2. Finite state machine is used to present the program behavior in this paper. In Normal program, the initial state of SENDER is waiting for data input (get0). After getting data (gd), it transits to be ready for sending data (send0). Then it sends data to RECEIVER (sd0). 0 is the alternating bit. After sending data, it transits to be waiting for acknowledgement (wfa0) from RECEIVER. If RECEIVER acknowledges correctly (ack0), it will prepare to get another data (get1). If data loss happens on the channel, SENDER will resend the old data. If the acknowledge is not bit 0, SENDER will keep on waiting for correct acknowledgement. After inserting the DES64e filer, SENDER is ready to encrypt data (DES64e0) after getting data. The encrypting action e64d encrypts the data and transmits SENDER to be ready to send it. In RECEIVER side, the received data should be decrypted before being delivered to the user. If error occurs in data decrypting (dcerr0), RECEIVER needs to receive the data again and SENDER should encrypt the data again.

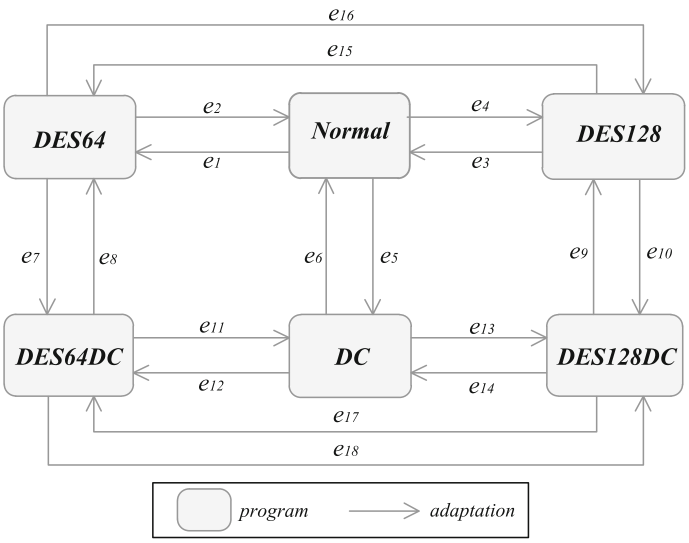


Fig. 3 Programs and adaptations of ACP

All programs and adaptations of ACP are illustrated in Fig. 3. We omit the detailed behavior and use a circle with a program name to represent each of them. Although there may exist more than one adaptation from Normal to DES64 program, such as e_{11} and e_{12} in Fig. 2, we depict only one adaptation e_1 for simplicity in Fig. 3.

4 Mode-supported Linear Temporal Logic

Following the formal model of adaptive programs, this section presents different types of traces in adaptive programs at first. Based on these traces, a new Linear Temporal Logic, mLTL, and its semantics are introduced. We use mLTL to specify the global requirements of ACP to illustrate our approach.

4.1 Traces of adaptive programs

The traces of a finite state machine are sequences of states. The states themselves are not observable, but just their atomic propositions are observable. Traces represent the possible executions of a state machine and can be also used to specify linear temporal properties [4]. An adaptive program in this paper has two types of traces, state traces and mode traces.

State traces represent the behavior of non-adaptive programs and adaptations in an adaptive program. Thus, they can be classified into two categories. The first category is the non-adaptive state trace, which is the execution of non-adaptive programs. All states in this trace belong to a non-adaptive program. The second is the adaptive state trace, which is the concatenation of non-adaptive state traces of different non-adaptive programs. This trace describes the adaptive behavior. States in the adaptive state trace may transit from one program to another.

Each non-adaptive state trace in an adaptive state trace is called a "*interval*". An adaptive state trace $t = s_0 s_1 s_2 \ldots$ (where $s_0, s_1, s_2, \ldots$ are state labels) may have $n (n > 0)$ intervals labelled with $i_1, \ldots, i_n$ respectively. If we use $\widetilde{i}$ to denote the corresponding non-adaptive state trace of the interval i, the adaptive state trace $t = \widetilde{i_1} \frown \widetilde{i_2} \frown \ldots \frown \widetilde{i_n}$ is the concatenation of the state sequence of its intervals. To specify the global properties for adaptive programs, we propose "mode trace" to represent the evolution process of an adaptive program when it is adapting continuously. For an adaptive state trace t which has n intervals labelled with $i_1, \ldots, i_n$ respectively, the corresponding mode trace is represented as $i_1 i_2 i_3 \ldots i_n$.

A non-adaptive state trace is depicted in Fig. 4a, in which each state belongs to the non-adaptive program Normal. Labels under the state circle are state names, and propositions hold on each state are shown above it. An adaptive state trace is depicted in Fig. 4b, in which there exists an adaptation between non-adaptive state traces of program Normal and DES64. For convenience, we also use the non-adaptive program's name to denote the interval name. The first two intervals of this adaptive state trace are labelled with "Normal" and "DES64" respectively, and the state sequence of interval Normal is $get_0 send_0 wfa_0 get_1$. The mode trace of this adaptive state trace is shown in Fig. 4c. The circle denotes a program, and the label under it is the program's name. Arrows between circles refer to adaptations. Different from the atomic propositions on each state of state traces, the propositions on each program in mode traces are whether an LTL formula satisfied on the state sequence of this interval. In Fig. 4c, LTL properties χ_1, χ_2 and χ_3 hold on the first program of the mode trace.

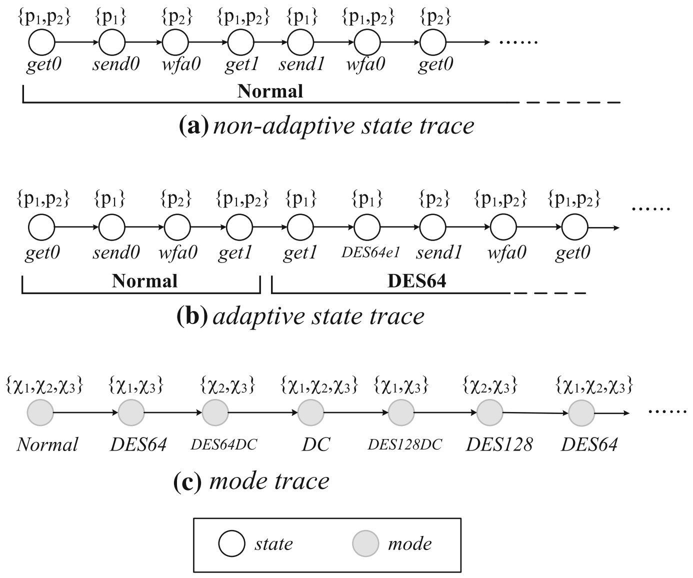

{p1,p2} {p1} {p2} {p1,p2} {p1} {p1,p2} {p2}
get0 send0 wfa0 get1 send1 wfa0 get0
Normal
(a) non-adaptive state trace
{p1,p2} {p1} {p2} {p1,p2} {p1} {p1} {p2} {p1,p2} {p1,p2}
get0 send0 wfa0 get1 get1 DES64e1 send1 wfa0 get0
Normal
DES64
(b) adaptive state trace
{χ1,χ2,χ3} {χ1,χ3} {χ2,χ3} {χ1,χ2,χ3} {χ1,χ3} {χ2,χ3} {χ1,χ2,χ3}
Normal DES64 DES64DC DC DES128DC DES128 DES64
(c) mode trace
state
mode

Fig. 4 State traces and mode traces

4.2 Syntax and semantics of mLTL

This subsection describes the syntax and semantic of mLTL. Mode traces are linear temporal and we adopt the LTL style formulae to define mLTL. mLTL formulae are formed according to the following grammar:

$$\kappa ::= \mathbf{true} \mid [\chi] \mid @v \mid \kappa \wedge \kappa \mid \neg\kappa \mid \bigcirc_m \kappa \mid \kappa \mathcal{U}_m \kappa \tag{2}$$

where χ is an LTL formula, and v is a mode name. Other operators such as $\vee (or)$, $\rightarrow (imply)$, $\leftrightarrow (coimply)$ can also be derived from above operators.

The precedence order on the operators is as follows. The [] and @ operators take precedence over any others. The unary operators take precedence over the binary ones. $\neg$ and $\bigcirc_m$ have same precedence. The temporal operator $\mathcal{U}_m$ takes precedence over $\wedge$, $\vee$, and $\rightarrow$.

To define the mLTL semantics, we first give the notation of LTL satisfaction on the infinite and finite state trace as follows:

- If the adaptive state trace t is infinite and ϕ is an LTL formula, then t satisfying ϕ is formally denoted as $t \vDash \phi$.
- If t is finite and ϕ is an LTL formula, then $t \vDash_{fin} \phi$ if and only if $t' \vDash \phi$, where t' is the infinite state trace constructed by repeating the last state of t.

Let $t = s_0 s_1 s_2 \ldots$ be an adaptive state trace with n intervals $i_1, i_2, \ldots i_n$. The mode trace mt of t is $i_1 i_2 \ldots$. We define mLTL semantics as follows:

- $t \vDash \mathbf{true}$.
- $t \vDash [\chi]$ if and only if $mt \vDash_m [\chi]$.
- $t \vDash @v$ if and only if $mt \vDash_m @v$.
- $t \vDash \bigcirc_m \kappa$ if and only if $mt \vDash_m \bigcirc_m \kappa$.
- $t \vDash \kappa_1 \mathcal{U}_m \kappa_2$ if and only if $mt \vDash_m \kappa_1 \mathcal{U}_m \kappa_2$.
- $t \vDash \kappa_1 \wedge \kappa_2$ if and only if $t \vDash \kappa_1$ and $t \vDash \kappa_2$.
- $t \vDash \neg\kappa$ if and only if $\neg t \vDash \kappa$.
- $mt \vDash_m \mathbf{true}$.
- $mt \vDash_m [\chi]$ if and only if $i_1 \vDash_m [\chi]$, which means $\widetilde{i_1} \vDash_{fin} \chi$.
- $mt \vDash_m @v$ if and only if $i_1 \vDash_m @v$, which means $\widetilde{i_1} \vDash_{fin} \Box v$.
- $mt \vDash_m \kappa_1 \wedge \kappa_2$ if and only if $mt \vDash_m \kappa_1$ and $mt \vDash_m \kappa_2$.
- $mt \vDash_m \neg\kappa$ if and only if $\neg mt \vDash_m \kappa$.
- $mt \vDash_m \bigcirc_m \lambda$ if and only if $mt' \vDash_m \lambda$, where $mt' = i_2 i_3 \ldots$.
- $mt \vDash_m \kappa_1 \mathcal{U}_m \kappa_2$ if and only if for some $j = 1, 2, \ldots, i_j \vDash_m \kappa_2$ and $i_1 \vDash_m \kappa_1, \ldots, i_{j-1} \vDash_m \kappa_1$.

As in LTL, the temporal operators of *eventually* and *always* on mode traces can be derived as follows:

$$\Diamond_m \kappa \overset{def}{=} [\mathbf{true}]\mathcal{U}_m \kappa \ (eventually) \tag{3}$$

$$\Box_m \kappa \overset{def}{=} \neg\Diamond_m \neg\kappa \ (always) \tag{4}$$

As a result, the following intuitive meaning of $\Diamond_m$ and $\Box_m$ is obtained. $\Diamond_m\kappa$ ensures that κ will be **true** eventually in the future. $\Box_m\kappa$ is satisfied if and only if it is not the case that eventually $\neg\kappa$ holds on the mode trace. This is equivalent to the fact that κ holds from now on forever. For instance, $\Diamond_m[\chi]$ ensures that the LTL formula χ will hold eventually on one program in the mode trace, while $\Box_m\chi$ ensures that χ is held on all programs in the mode trace. Other

Specially, we introduce the formula $@v$ for the properties on program v or related programs. By this formula, global properties can be specified on special modes.

$$\kappa_1 = \Box_m(@v \rightarrow [\chi]) \tag{5}$$
$$\kappa_2 = \Box_m(\bigcirc_m(@v) \rightarrow \kappa) \tag{6}$$
$$\kappa_3 = \Box_m(@v \rightarrow \bigcirc_m\kappa) \tag{7}$$
$$\kappa_4 = \Box_m(([\mathbf{true}]\mathcal{U}_m @v) \rightarrow \kappa) \tag{8}$$

For instance, the mLTL formula κ_1 (Eq. 5) holds on a mode trace, if χ holds on programs named v. Moreover, properties on predecessor and successors of one program on mode traces may be presented. The mLTL formula κ_2 (Eq. 6) assures that mLTL formula κ holds on directly preceding programs of v. While the mLTL formula κ_3 (Eq. 7) assures that κ holds on directly successive programs of v. Beside the direct predecessor and successors of a program, all preceding programs can be expressed by the mLTL formula κ_4 (Eq. 8) which assures that κ holds on all preceding programs of v. However, the all successive programs could not be presented without *Past* operators [8].

The transitional properties could also partially be expressed by mLTL. The A-LTL [48] use $\phi \overset{true}{\rightharpoonup} \varphi$ to specify a program that initially satisfies ϕ, and in a certain state A, it stops being constrained by ϕ, then in the next state B, it starts to satisfy φ. We use the mLTL formula $[\phi] \wedge \bigcirc_m[\varphi]$ to specify this program. The state trace has two intervals. The state sequence of the first interval satisfies the LTL formula ϕ and the second satisfies the LTL formula φ.

4.3 Global specification of ACP

We specify some global properties of ACP to illustrate mLTL.

For program Normal in Fig. 3, one global property is that SENDER should eventually send the data to RECEIVER once SENDER gets it from data source before any other adaptation occurs. This property can be specified by mLTL as below:

$$\kappa = \Box_m(@Normal \rightarrow [\Box(INPUT \rightarrow \Diamond SENT)]) \tag{9}$$

Another property of program Normal is that SENDER can infinitely often input data before any other adaptation occurs. This property can be specified by mLTL as below:

$$\kappa = \Box_m(@Normal \rightarrow [\Box\Diamond(INPUT)]) \tag{10}$$

A global property of ACP to ensure that the data will be sent eventually before any other adaptation occurs is

$$\kappa = \Box_m[\Box(INPUT \rightarrow \Diamond SENT)] \tag{11}$$

which means that SENDER should eventually send the data to RECEIVER once it gets it from data source on all programs.

A global property to ensure that if the program could not infinitely often input data before any other adaptation occurs, its next program could do that. This property is formulized as below:

$$\kappa = \Box_m[[!\Box\Diamond INPUT] \rightarrow \bigcirc_m[\Box\Diamond INPUT]] \tag{12}$$

A global property of reachability is that SENDER will eventually reach a program in which data are compressed before any other adaptation occurs. This property is formulized as below:

$$\kappa = \Diamond_m[\Box(INPUT \rightarrow \Diamond DATACOMPRS)] \tag{13}$$

In such a program, once input, the data will be compressed eventually before any other adaptation occurs.

A global property of safety is that SENDER will never reach a program in which data are encrypted by DES64e and DES128e filters simultaneously before any other adaptation occurs. This property is formulized as below:

$$\begin{aligned}\kappa \ &= \Box_m[\Box((DES64ENCD \rightarrow \neg\Diamond DES128ENCD) \\ &\wedge(DES128ENCD \rightarrow \neg\Diamond DES64ENCD))]\end{aligned} \tag{14}$$

In each program, there does not exit such a state on which the data, after having been processed by a DES 64-bit encryption filter, will be processed by another DES 128-bit encryption filter, or versa.

A global property of maintainability is that a program with data encryption error will be eventually repaired. This property is formulized as below:

$$\begin{aligned}\kappa \ &= \Box_m([\Box(INPUT \rightarrow \Diamond ENCRYPERR)] \rightarrow \\ &\Diamond_m[\Box(INPUT \rightarrow \Diamond SENT)])\end{aligned} \tag{15}$$

We consider that data encryption error always occur in a program, when a decryption error will eventually occur ($ENCRYPERR$) after data inputting ($INPUT$) before any other adaptation occurs. If there is a program in which data encryption error always occur, it will eventually be repaired, which means that the input data will be sent successfully ($\Diamond SENT$) before any other adaptation occurs.

5 mLTL model checking

In this section, we address the model checking problem for mLTL. The starting point is an adaptive program $\mathcal{AP}$ and an mLTL formula κ. The problem is to check whether $\mathcal{AP} \models \kappa$. If κ is refuted, an error trace will be provided for debugging purpose.

As discussed in Sect. 4, the adaptive state trace of the adaptive program has many intervals according to different non-adaptive program. The mLTL semantics on the state trace could be interpreted by mLTL semantics on the mode trace. The mLTL model checking could be decomposed into LTL checking on the state sequence of intervals and mLTL checking on the mode traces of the adaptive program. This characteristic enables mLTL model checking based on LTL and mature model checking tools. Firstly, the non-adaptive program may exhibit different behaviors because of multiple adaptations from or to this program. We should first normalize them to ensure that different behaviors of a non-adaptive program are decomposed into different non-adaptive programs and LTL checking on the state sequence of intervals could be done on the generated non-adaptive programs. We prove the trace equivalence between the adaptive program and its normalization. Secondly, a different semantic for the adaptive program, the global semantic, is proposed to support mLTL checking on the mode traces. Then, the mLTL model checking could be done by three steps: (1) extract LTL properties of the non-adaptive programs from the mLTL formula and translate the mLTL formula to an LTL formula, (2) checking LTL properties from (1) on each non-adaptive program, and (3) checking the LTL formula from (1) on a transition system constructed by the global semantic and the result of step (2). We also prove the correctness of our approach.

This section discusses how to normalize the adaptive program, and then the global semantic of the adaptive program. Finally, the LTL-based model checking approach is presented.

5.1 Normalization of adaptive programs

In the formal model of the adaptive program, the non-adaptive program may exhibit different behaviors because of the adaptations. For instance, there exists an adaptation, $(\mathcal{P}_i, s_i, e_i, \mathcal{P}_k, s) \in \varphi$. This adaptation implies that the adaptive program will run as the behavior of $\mathcal{P}_k$ after adaptation e_i, but the initial state of $\mathcal{P}_k$ is changed to s. On the other hand, this adaptation also changes the behavior of $\mathcal{P}_i$. The behavior of $\mathcal{P}_i$ terminates in the state s_i and switches to $\mathcal{P}_k$.

For instance, the adaptations e_{12} and e_{22} in Fig. 2 lead to the starting state get1 in DES64 program and get1 in Normal program respectively, and the ending state get1 in Normal program and get1 in DES64 respectively.

The Normal program will exhibit four different behaviors:

- Normal$_1$, in which get1 is the starting state and get0 is the ending state. All of non-adaptive state traces of Normal$_1$ are like get1send1...get0, start from get1 and end at get0.
- Normal$_2$, in which get0 is the starting state and get0 is also the ending state. All of non-adaptive state traces of Normal$_2$ are like get0...get0, start from get0 and end at get0.

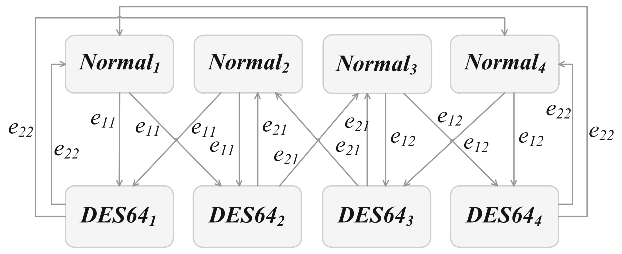


Fig. 5 Adaptation graph of adaptations between Normal and DES64 programs

- Normal$_3$, in which get0 is the starting state and get1 is the ending state. All of non-adaptive state traces of Normal$_3$ are like get0send0...get1, start from get0 and end at get1.
- Normal$_4$, in which get1 is the starting state and get1 is also the ending state. All of non-adaptive state traces of Normal$_4$ are like get1...get1, start from get1 and end at get1.

Different behaviors and adaptations between Normal and DES64 programs are shown in Fig. 5.

The objective of "Normalization" is to ensure that each non-adaptive program exhibits only one behavior. That means (1) all state traces of a non-adaptive program start from the only one state and end at another only one state. (2) the starting state is the initial state of the non-adaptive program. (3) the source state of an adaptation is the ending state of the source non-adaptive program and the target state of the adaptation is the starting state of the target non-adaptive program. Formally, for each non-adaptive program $\mathcal{P}$ in the normalized adaptive program, if $(\mathcal{P}_i, s_i, e_i, \mathcal{P}, s) \in \varphi$, then s is the starting state of $\mathcal{P}$, and if $(\mathcal{P}, s', e_j, \mathcal{P}_j, s_j) \in \varphi$, then s' is the ending state of $\mathcal{P}$.

To normalize adaptive programs, we first define a function to denote the behavior of a non-adaptive program in which the starting and ending states are designated.

Definition 3 *(normalize function)* The normalize function for the adaptive program is $\Delta : \Theta \times \Lambda \times \Lambda \rightarrow \Theta$, where Θ is the set of all non-adaptive programs and Λ is the set of all states.

Given $\mathcal{P} = \langle v, S, Act, \sigma, s_0, AP, L\rangle$, the normalize function $\Delta(\mathcal{P}, s, t) = \langle v, S, Act, \sigma, s, AP, L\rangle$ where $s, t \in S(\mathcal{P})$ and $s \neq t$. We use the label $\overrightarrow{\mathcal{P}^{st}}$ to denote the normalized non-adaptive program $\mathcal{P}$ with starting state s and ending state t for convenience.

The behavioral set function $BS(\mathcal{P})$ is used to present possibly different behaviors of a non-adaptive program, where $\mathcal{P}$ is the program and $BS(\mathcal{P})$ maps $\mathcal{P}$ to a program set. The calculation of $BS(\mathcal{P})$ is presented as follows:

$$BS(\mathcal{P}) = \begin{cases} \{\Delta(\mathcal{P}, s, t) \mid \exists(\mathcal{P}_1, s_1, e_1, \mathcal{P}, s), (\mathcal{P}, t, e_2, \mathcal{P}_2, s_2) \in \varphi\} \cup \{\mathcal{P}_0\} \\ \text{——(if } \mathcal{P} = \mathcal{P}_0) \\ \{\Delta(\mathcal{P}, s, t) \mid \exists(\mathcal{P}_1, s_1, e_1, \mathcal{P}, s), (\mathcal{P}, t, e_2, \mathcal{P}_2, s_2) \in \varphi\} \\ \text{——(if } \mathcal{P} \neq \mathcal{P}_0) \end{cases} \quad (16)$$

The initial state s_0 of the initial program $\mathcal{P}_0$ is a special state for $BS(\mathcal{P})$, if there does not exist an adaptation from other programs to s_0. As an adaptive program starts running from this state, $\mathcal{P}_0$ is in the behavioral set of $BS(\mathcal{P}_0)$ absolutely.

Based on the definition of behavioral set function, the Normalized Adaptive Program is defined as follows.

Definition 4 *(normalized adaptive program)* The normalized adaptive program of an adaptive program $\mathcal{AP} = \langle A, E, \varphi, \mathcal{P}_0\rangle$, is also an adaptive program, denoted by $\overrightarrow{\mathcal{AP}}$. $\overrightarrow{\mathcal{AP}} = \langle A^N, E^N, \varphi^N, \mathcal{P}_0^N\rangle$ where

- $A^N = \bigcup_{\mathcal{P}\in A} BS(\mathcal{P})$.
- $E^N = E$.
- $\mathcal{P}_0^N = \mathcal{P}_0$.
- $\varphi^N : A^N \times \bigcup_{\mathcal{P}\in A^N} S(\mathcal{P}) \times E^N \to A^N \times \bigcup_{\mathcal{P}\in A^N} S(\mathcal{P})$ where $\bigcup_{\mathcal{P}\in A^N} S(\mathcal{P})$ is the set of all states of programs in A^N. For each adaptation relation $(\mathcal{P}_1, t, e, \mathcal{P}_2, s) \in \varphi$, $\{(\overrightarrow{\mathcal{P}_1^{t't}}, t, e, \overrightarrow{\mathcal{P}_2^{ss'}}, s) \mid \overrightarrow{\mathcal{P}_1^{t't}} \in BS(\mathcal{P}_1), t' \in S(\mathcal{P}_1), \overrightarrow{\mathcal{P}_2^{ss'}} \in BS(\mathcal{P}_2, s' \in S(\mathcal{P}_2))\} \subseteq \varphi^N$.

If there is an adaptation e from the non-adaptive program $\mathcal{P}_1$ (the ending state t) to $\mathcal{P}_2$ (the starting state s) in the original adaptive program, this adaptation will also connect each non-adaptive program (with ending state t) in $BS(\mathcal{P}_1)$ to each non-adaptive program (with starting state s) in $BS(\mathcal{P}_2)$ in the normalized adaptive program.

According to the construction process of the normalized adaptive program, we could find a same state trace in $\overrightarrow{\mathcal{AP}}$ for each state trace in $\mathcal{AP}$, and vice versa. Thus, the adaptive program and its normalization are trace equivalent, as stated in the following lemma.

Lemma 1 *An adaptive program $\mathcal{AP}$ is trace equivalent to its normalization $\overrightarrow{\mathcal{AP}}$.*

Since trace equivalence of transition systems implies they satisfy the same line-time properties [4], this lemma implies that the adaptive program and its normalization satisfy the same line-time properties.

5.2 Global semantic of adaptive programs

The mLTL semantic on the state trace of adaptive program could be interpreted as mLTL semantic on the mode trace of its normalization. Inspired by this semantic of mLTL, we propose the global semantic of the adaptive program. To model checking an mLTL formula κ on the normalized adaptive program $\mathcal{AP}$, we first construct the global labelled transition system for $\mathcal{AP}$ over κ.

Definition 5 (*global labelled transition system*) Given a normalized adaptive program $\mathcal{AP} = \langle A, E, \varphi, \mathcal{P}_0\rangle$, the global semantic of $\mathcal{AP}$ over an mLTL formula κ is represented by a global labelled transition system(GLTS), $\Upsilon_{\mathcal{AP}}^{\Psi} = \langle A, E, \psi, \mathcal{P}_0, \mathcal{L}\rangle$, where

- Ψ: the LTL formula set extracted from κ
- ψ: $A \times E \rightarrow A$ is the transition relation. $(\mathcal{P}_1, e, \mathcal{P}_2) \in \psi$, if there exists an adaptation relation $(\mathcal{P}_1, s, e, \mathcal{P}_2, s') \in \varphi$
- $\mathcal{L}$: $A \rightarrow 2^{\Psi}$ is a labelling function

When model checking, a GLTS is constructed for each mLTL formula on $\mathcal{AP}$. As defined in Sect. 4.2, the satisfaction of mLTL on the mode trace is calculated based on the satisfaction of its LTL properties on the state sequence of intervals. The LTL formula set (Ψ) will first be extracted from κ. For an mLTL formula, we extract an LTL formula χ from the $[\chi]$ part, and $\Box v$ from the $@v$ part of the mLTL formula. Then, each LTL formula will be checked on each program of $\mathcal{AP}$. The labelling function $\mathcal{L}$ indicates that which LTL formula is satisfied on the non-adaptive program. For instance, $\langle \mathcal{P}, \{\chi_1, \chi_2, \chi_3\} \rangle \in \mathcal{L}$ means that LTL formulae χ_1, χ_2 and χ_3 hold on the non-adaptive program $\mathcal{P}$.

After normalization, the non-adaptive program has the starting and ending state. All state traces of the non-adaptive program start from the starting state and end at the ending state. The starting state could be explicitly defined in the input languages of model checker, for example FSP of LTSA. To ensure that the ending state is the final state of each state trace of the non-adaptive program, we should also explicitly indicate the ending state for LTL model checking on the non-adaptive programs. For this purpose, we add a state s_i' same as the ending state s_i and a τ transition from s_i to s_i' and a self-loop τ transition on state s_i'. This ensures that the ending state s_i is repeated infinitely at the end of the state sequence.

5.3 LTL-based model checking

LTL-based model checking approach transforms the problem of mLTL model checking on an adaptive program $\mathcal{AP}$ to LTL on the GLTS of $\overrightarrow{\mathcal{AP}}$. We first give the transforming function of the mLTL formula, and prove that the satisfaction of the mLTL formula on $\mathcal{AP}$ is equivalent to the satisfaction of the generated LTL formula on GLTS of $\overrightarrow{\mathcal{AP}}$. Finally, we give the algorithm of mLTL model checking.

$$\begin{aligned}
\mathcal{T}(\mathbf{true}) &= \mathbf{true} \\
\mathcal{T}([\chi]) &= p(\chi) \\
\mathcal{T}(@v) &= p(\Box v) \\
\mathcal{T}(\kappa_1 \wedge \kappa_2) &= \mathcal{T}(\kappa_1) \wedge \mathcal{T}(\kappa_2) \\
\mathcal{T}(\neg \kappa_1) &= \neg \mathcal{T}(\kappa_1) \\
\mathcal{T}(\bigcirc_m \kappa_1) &= \bigcirc \mathcal{T}(\kappa_1) \\
\mathcal{T}(\kappa_1 \mathcal{U}_m \kappa_2) &= \mathcal{T}(\kappa_1) \mathcal{U} \mathcal{T}(\kappa_2) \\
\mathcal{T}(\Diamond_m \kappa_1) &= \Diamond \mathcal{T}(\kappa_1) \\
\mathcal{T}(\Box_m \kappa_1) &= \Box \mathcal{T}(\kappa_1) \\
\mathcal{T}(\kappa_1 \vee \kappa_2) &= \mathcal{T}(\kappa_1) \vee \mathcal{T}(\kappa_2) \\
\mathcal{T}(\kappa_1 \rightarrow \kappa_2) &= \mathcal{T}(\kappa_1) \rightarrow \mathcal{T}(\kappa_2) \\
\mathcal{T}(\kappa_1 \leftrightarrow \kappa_2) &= \mathcal{T}(\kappa_1) \leftrightarrow \mathcal{T}(\kappa_2)
\end{aligned}$$

Fig. 6 Transforming function of mLTL formulae

The transforming function of mLTL is defined as $\mathcal{T}(\kappa)$ in Fig. 6. It transforms an mLTL formula to an LTL formula. Here, a predicate $p()$ is defined. The predicate $p(\chi)$ (where χ is an LTL formula) is **true** on $\mathcal{P}$ in the GLTS, if LTL formula $\chi \in \mathcal{L}(\mathcal{P})$.

Theorem 1 *An mLTL formula κ holds on an adaptive program $\mathcal{AP}$, if and only if $\mathcal{T}(\kappa)$ holds on the GLTS of $\overrightarrow{\mathcal{AP}}$.*

Proof We should prove that $\mathcal{AP} \models \kappa$, iff $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models \mathcal{T}(\kappa)$, where Ψ is the LTL property set of κ. According to Lemma 1, $\mathcal{AP} \models \kappa$, iff $\overrightarrow{\mathcal{AP}} \models \kappa$. We only prove that $\overrightarrow{\mathcal{AP}} \models \kappa \Rightarrow \Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models \mathcal{T}(\kappa)$

- If $\kappa = [\chi]$, $\overrightarrow{\mathcal{AP}} \models [\chi]$ means for each state trace t of $\overrightarrow{\mathcal{AP}}$, $t \models [\chi]$. Thus, t' mode trace $mt \models_m [\chi]$. Definition of GLTS implies that there exists a mode trace mt in $\overrightarrow{\mathcal{AP}}$ for each mode trace of $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}}$ correspondingly. Since $mt \models_m [\chi]$, we have $\widetilde{i_1} \models_{fin} \chi$, where i_1 is the first interval of mt. The state sequence of first interval of each mode trace in $\overrightarrow{\mathcal{AP}}$ satisfy the LTL formula χ. That is to day the behavior of the non-adaptive program of the first mode on the mode traces satisfies χ. Thus, $p(\chi)$ holds on the first state of all state traces in $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}}$. We conclude that $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models p(\chi)$.
- If $\kappa = @v$, $\overrightarrow{\mathcal{AP}} \models @v$ means the state sequence of first interval of each mode trace in $\overrightarrow{\mathcal{AP}}$ satisfy the LTL formula $\Box v$. Proved as the same as $[\chi]$, proposition $p(\Box v)$ holds on the first state of each state traces in $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}}$. Thus, $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models @v$.
- If $\kappa = \kappa_1 \wedge \kappa_2$, $\overrightarrow{\mathcal{AP}} \models \kappa_1 \wedge \kappa_2$ means that $\overrightarrow{\mathcal{AP}} \models \kappa_1$ and $\overrightarrow{\mathcal{AP}} \models \kappa_2$. Thus, $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models \mathcal{T}(\kappa_1)$ and $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models \mathcal{T}(\kappa_2)$.
- If $\kappa = \neg\kappa_1$, $\overrightarrow{\mathcal{AP}} \models \neg\kappa_1$ means that κ_1 does not hold on the first mode of all mode traces. Thus, $\mathcal{T}(\kappa_1)$ does not hold on first state of all state traces in $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}}$. $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models \neg\mathcal{T}(\kappa_1)$.
- If $\kappa = \bigcirc_m\kappa_1$, $\overrightarrow{\mathcal{AP}} \models \bigcirc_m\kappa_1$ means that κ_1 holds from the second mode in all mode traces. Thus, $\mathcal{T}(\kappa_1)$ should hold from the second state in all state traces of $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}}$. $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models \bigcirc\mathcal{T}(\kappa_1)$.
- If $\kappa = \kappa_1\mathcal{U}_m\kappa_2$, $\overrightarrow{\mathcal{AP}} \models \kappa_1\mathcal{U}_m\kappa_2$ means that κ_1 holds on modes until κ_2 holds in each mode trace. Thus, $\mathcal{T}(\kappa_1)$ should hold on states until $\mathcal{T}(\kappa_2)$ holds in each state trace of $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}}$. $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models \mathcal{T}(\kappa_1)\mathcal{U}\mathcal{T}(\kappa_2)$.

$\mathcal{T}(\Diamond_m\kappa_1)$, $\mathcal{T}(\Box_m\kappa_1)$, $\mathcal{T}(\kappa_1 \vee \kappa_2)$, $\mathcal{T}(\kappa_1 \rightarrow \kappa_2)$ and $\mathcal{T}(\kappa_1 \leftrightarrow \kappa_2)$ will also be proved because these operators can be expressed by the above basic operators. $\Upsilon^{\Psi}_{\overrightarrow{\mathcal{AP}}} \models \mathcal{T}(\kappa) \Rightarrow \overrightarrow{\mathcal{AP}} \models \kappa$ could be proved similarly. □

According to the three steps of mLTL model checking mentioned above, the algorithm of mLTL model checking based on LTL is presented in Algorithm 1.

At first, the adaptive program is normalized (line 2). All LTL formulae are extracted from κ and stored in the array *ltlpropset* (line 3). All non-adaptive programs are also

Algorithm 1: mLTL model checking algorithm

```
Data: an mLTL formula $\kappa$, an adaptive program $\mathcal{AP}$.
Result: YES or print the error trace.
1  begin
2      $\overrightarrow{\mathcal{AP}} \longleftarrow normalize(\mathcal{AP})$
3      $ltlpropset \longleftarrow exractLTL(\kappa)$
4      $programs \longleftarrow extractPrograms(\overrightarrow{\mathcal{AP}})$
5      for $p \in programs$ do
6          for $f \in ltlpropset$ do
7              $R(p, f) \longleftarrow modelcheck(p, f)$
8      $glts \longleftarrow contructGLTS(\mathcal{AP}, R)$
9      $ltl \longleftarrow transform\_mLTL(\kappa)$
10     $modelcheck(glts, ltl)$
```

extracted from $\mathcal{AP}$, processed and stored in the array $programs$ (line 4). Then, these LTL formulae will be checked on each non-adaptive program (lines 5–7). The LTL model checking is available in LTSA, and the checking result is a boolean matrix R that indicates the labelling function $\mathcal{L}$ of the GLTS. The GLTS is constructed based on the matrix R and $\overrightarrow{\mathcal{AP}}$ (line 8), and κ is transformed to an LTL formula ltl (line 9). Finally, ltl is checked on $glts$ (line 10).

Since the number of non-adaptive programs in $\overrightarrow{\mathcal{AP}}$ is not only dependent on the number of non-adaptive programs and adaptations in $\mathcal{AP}$, but also on the structure of programs and adaptations, we analyze the complexity of mLTL model checking on the normalized adaptive program. Assume a normalized adaptive program $\overrightarrow{\mathcal{AP}}$ contains n programs $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, and m adaptations. The complexity of mLTL model checking is calculated based on LTL model checking problem. As discussed in [26,44], the time and space complexity of LTL model checking χ on transition system TS are $\mathcal{O}(2^{|\chi|} * |TS|)$. We assume all programs are of similar size $|\mathcal{P}| : |\mathcal{P}| \approx |\mathcal{P}_i|$, for all i. We denote $|\overrightarrow{\mathcal{AP}}|$ as the size of the GLTS of $\overrightarrow{\mathcal{AP}}$. If the size of the property set of κ is k and the average size of LTL formula in κ is $|\chi|$, the time complexity of our algorithm is $\mathcal{O}(n * k * 2^{|\chi|} * |\mathcal{P}| + 2^{|\mathcal{T}(\kappa)|} * |\overrightarrow{\mathcal{AP}}|)$ and the space complexity is $\mathcal{O}(max(2^{|\chi|} * |\mathcal{P}|, 2^{|\mathcal{T}(\kappa)|} * |\overrightarrow{\mathcal{AP}}|))$.

6 Implementation and experiments

In the previous sections, we discuss how to model, specify and verify adaptive programs. A Modelling and Analysis Tool for Adaptive Programs (MATAP) has been developed and this section will be devoted to its implementation. We will use the ACP case to illustrate how to develop global specification of adaptive software.

We choose LTSA as the LTL model checker for adaptive programs. LTSA is a verification tool that mechanically checks that the specification of a system satisfies the properties required of its behaviour. A system in LTSA is modelled as a set of interacting finite state machines. LTSA uses the FSP process calculus to specify behaviour models and Fluent Linear Temporal Logic (FLTL) to define properties. A *fluent* is a

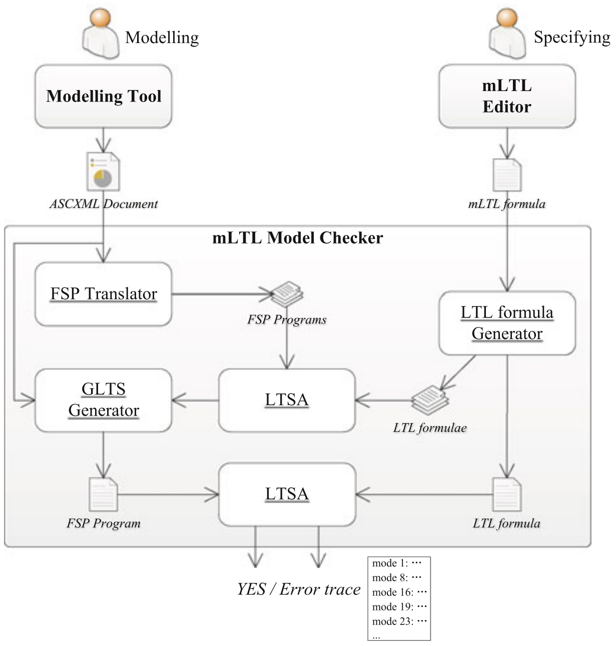


Fig. 7 MATAP architecture

property of the world that holds after it is initiated by an action and ceases to hold when terminated by another action. We use fluent to define propositions hold on states of the adaptive program.

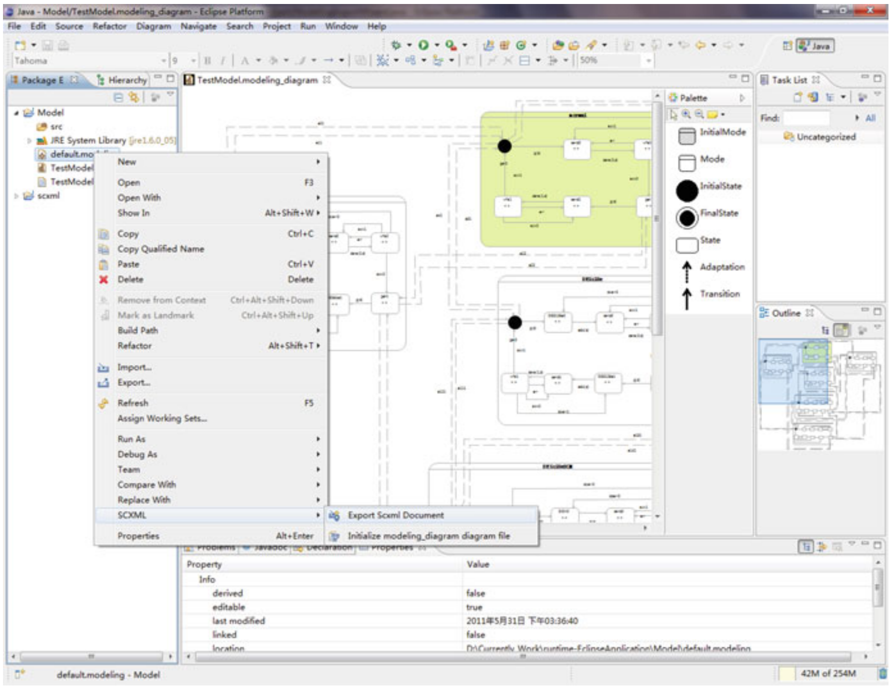
6.1 Architecture

The architecture of MATAP is shown in Fig. 7. MATAP is developed based on Eclipse[2] platform, and consists of a visual modelling tool for adaptive programs, an mLTL formula editor for specifying global properties and an mLTL model checker.

To improve extensibility and interoperability, the modelling tool exports a XML document for model checking. We extend the W3C's State Chart XML (SCXML)[3] by adding adaptation elements and define the Adaptation SCXML (ASCXML) to store adaptive programs. Our mLTL model checker is based on LTSA which uses the FSP process calculus to specify behaviour models and supports LTL model checking. From the FSP description, LTSA generates a LTS model. MATAP uses LTSA for LTL model checking in our mLTL model checking algorithm. According to the mLTL model checking algorithm, the *FSP Translator* extracts non-adaptive programs from the normalized adaptive program and translates them into FSP language. The *LTL formula Generator* extracts LTL property set from an mLTL formula and transforms the mLTL formula to an LTL formula by the transforming function as defined in Fig. 6.

[2] http://www.eclipse.org/.

[3] http://www.w3.org/TR/scxml/.



Property	Value
Info	
derived	false
editable	true
last modified	2011年5月31日 下午03:36:40
linked	false
location	D:\Currently Work\runtime-EclipseApplication\Model\default.modeling

Fig. 8 Modelling tool

Each extracted LTL formula will be checked on each FSP program generated by *FSP Translator*. The checking result (**true** or **false**) and the adaptive program will be used by *GLTS Generator* to construct the global labelled transition system and generate another FSP program. Finally, the transformed LTL formula will be checked on this FSP program, and mLTL model checker will print the result, *YES* or *Error trace*.

6.2 Modelling and specifying adaptive programs

SCXML is a general-purpose event-based state machine language in support of Statecharts[21]. SCXML describes states, events, and transitions. States represent the status of the system. Events represent what happens. Transitions move between states, and are triggered by events. An open source Java implementation of SCXML is available from Apache.[4] We add three elements to SCXML, "$\langle initialmode \rangle$", "$\langle mode \rangle$" and "$\langle adaptation \rangle$". The *initialmode* defines the initial program of an adaptive program, and the *mode* defines the other non-adaptive programs. The *adaptation* defines the adaptations among non-adaptive programs. The original SCXML elements are under *initialmode* or *mode* elements. The modelling tool as shown in Fig. 8 has been developed to model adaptive programs. Modes, states, transitions and adaptations can be modelled visually, and an ASCXML document can be exported finally.

[4] http://commons.apache.org/scxml/.

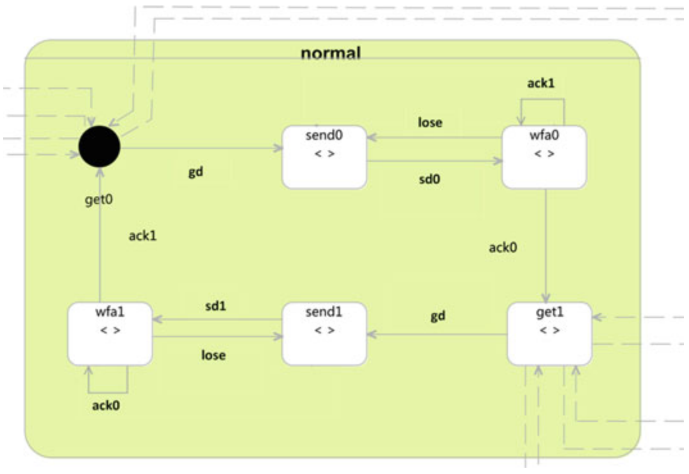


Fig. 9 Behavior of SENDER in normal mode

Next, we use MATAP to illustrate modeling the non-adaptive and adaptive program of ACP. For implementing mLTL model checking via LTSA, we also use the case to show automatic translation of non-adaptive program to FSP.

The non-adaptive program is defined as a finite state machine, and the adaptive program consists of a set of non-adaptive programs and adaptations among states of them. In the MATAP modelling tool, we provide "Mode", "InitialMode", "InitialState", "State", "Transition", and "Adaptation" primitives to visual model the non-adaptive programs and adaptations. The propositions hold on the state can also be defined on it. The behavior model of SENDER in the *normal* mode is illustrated in Fig. 9. The dotted lines are adaptations between *normal* and other non-adaptive programs.

When generating FSP program from the non-adaptive program in normalized adaptive programs, we map the action, state and transition to action, process name and action prefix in FSP respectively. The generated FSP program of the $normal_3$ in which get0 is the starting state and get1 is the ending state in Fig. 9 is as follow. Since `GET1` is the ending state, a new state `GET1_` and the `tao`(τ) transition are inserted automatically.

```
SENDER_normal3 = (gd -> SEND0),
SEND0 = (sd0 -> WFA0),
WFA0 = (ack0 -> GET1 | lose -> SEND0 | ack1 -> WFA0),
GET1 = (gd -> SEND1 | tao -> GET1_),
GET1_ = (tao -> GET1_),
SEND1 = (sd1 -> WFA1),
WFA1 = (ack1 -> SENDER_normal3 | loses -> SEND1 | gack0 -> WFA1).
```

There are two propositions "$INPUT$" and "$SENT$" that mean the data is read and ready to be sent, and the data is sent successfully respectively. $INPUT$ holds on state send0 and send1, $SENT$ on get1 after the action $ack0$ and get0 after the action $ack1$. In FSP, LTS models are essentially based on actions or events. FSP introduces *fluent* as a mean of describing abstract states of LTS models. Logical properties can be specified in terms of fluents and analyzed using LTSA. Since the action gd (get data) change the current to state send0 or send1 on which the proposition $INPUT$ holds.

A fluent $INPUT$ is generated as follow. It means that when the action gd occurs the proposition $INPUT$ will become $true$ immediately, and when other actions occur it will become $false$ immediately.

```
set ACTIONS = {gd, sd0, sd1, ack0, ack1, lose, tao}
fluent INPUT = <{gd}, ACTIONS\{gd}> initially 0
fluent SENT = <{ack0, ack1}, ACTIONS\{ack0, ack1}>  initially 0
```

After modeling the adaptive program, we should specify them by mLTL formulae. We define the concrete syntax for mLTL operators and use the same syntax of FLTL in LTSA for LTL operators. `Gm`, `Fm`,`Xm` and `Um` are used to present $\Box_m$, $\Diamond_m$, $\bigcirc_m$ and $\mathcal{U}_m$ respectively.

The global properties of ACP from Eq. 9 to 15 are specified as follows:

```
Gm(@ normal  ->  [](INPUT  ->  <> SENT)])
Gm(@ normal  ->  [] <> INPUT])
Gm([](INPUT  ->  <> SENT)])
Gm([![] <>INPUT]  ->  Xm[[] <>INPUT])
Fm([](INPUT  ->  <>DATACOMPRS) ] )
Gm([] ((DES64ENCD -> !<>DES128ENCD) && (DES128ENCD -> !<>DES64ENCD))])
Gm([] (INPUT  ->  <>ENCRYPERR)]  ->  Fm([] (INPUT  ->  <>SENT)]))
```

6.3 Model checking using LTSA

When the adaptive program and its specification have been developed, the mLTL model checker will take them as input and checking properties on the program automatically. We use the property `Gm(@normal -> [](INPUT-><> SENT)])` to illustrate the process of mLTL model checking.

The process of checking the mLTL formula consists of four steps.

1. Generating an FSP program FSP_i for each non-adaptive programs. The model checker first normalizes the adaptive program and generates an FSP program for each non-adaptive programs as discussed previously.
2. Extract LTL formulae LTL_i from the mLTL formula κ, and transform κ to an LTL formula LTL_κ. From the mLTL formula, two LTL formulae are extracted, $\Box v$ and `[](INPUT -><> SENT)`. The formula `[](INPUT -><> SENT])` will be checked on each FSP program. For the formula $\Box v$, a special fluent is used in the FSP program generated from the GLTS which will be discussed in step (3). Then the mLTL formula is transformed to an LTL formula `[](AT_NORMAL -> LTL1)` defined by FLTL, where `AT_NORMAL` and `LTL1` are asserts generated in step (3).
3. Generating an FSP program FSP_G from GLTS. An FSP program will be generated from the GLTS of the adaptive program. After being normalized, the adaptive program ACP has 24 non-adaptive programs and 144 adaptations among them. According to modes and adaptations in GLTS, the action, the process name and the action prefix could be generated in the FSP program. The proposition on each mode of GLTS is a set of LTL formulae that means these formulae hold the the mode. After LTL formula checking on each FSP program, the LTL formula set on each mode of GLTS is calculated.

The whole adaptation set and single adaptation set for each mode are generated firstly as follows.

```
/*set of all adaptations*/
set ADAPTATIONS = {e11, e12, e21, e22, e31, e32, e41,
                   e42, e51, e52, e61, e62, e71, e72,
                   e81, e82, e91, e92, e101,e102,e111,
                   e112,e121,e122,e131,e132,e141,e142,
                   e151,e152,e161,e162,e171,e172,e181,e182}

/*set of adaptations which change the adaptive
program into a mode*/
set ADAPT2NORMAL_1   = {e21,e22,e31,e32,e61,e62}
```

After adaptations in `ADAPT2NORMAL_1`, the ACP transits into the behavioral mode `NORMAL_1`. Thus, fluent and asserts could be generated to denote the mode name. For instance, the program `NORMAL_1`, `NORMAL_2`, `NORMAL_3`, and `NORMAL_4` have the same name `NORMAL`. We can see that the mLTL formula $@normal$ can be expressed directly if we have the assert to denote the mode name.

```
fluent AT_NORMAL_1 = <ADAPT2NORMAL_1,
       ADAPTATIONS\ADAPT2NORMAL_1> initially 0

assert AT_NORMAL   = (AT_NORMAL_1 || AT_NORMAL_2 || AT_NORMAL_3
                      || AT_NORMAL_4)
```

Then, a fluent could be generated for each LTL formula LTL_i by the result of LTL_i checking on each FSP_i. The fluent means that on which mode the LTL_i formula holds. We use a *set* to describe adaptations that transit other modes to the modes on which the LTL_i formula holds. Since `[](INPUT − ><> SENT)` holds on all FSP program FSP_i, its fluent is as follow:

```
set LTL1SET = {e11,e12,e21,e22,e31,e32,e41,e42,e51,e52,e61,e62,e71,e72,
                   e81,e82,e91,e92,e101,e102,e111,e112,e121,e122,e131,e132,
                   e141,e142,e151,e152,e161,e162,e171,e172,e181,e182}
fluent LTL1 = (LTL1SET, ADAPTATIONS\LTL1SET) initially 0
```

4. Checking LTL_κ on FSP_G. The LTL_κ `assert PROP1 = ([](AT_NORMAL -> LTL1))` is checked on the FSP program FSP_G.

6.4 Experiments

In this subsection, we illustrate the performance of our approach by using the global properties of *ACP* from Eqs. 9 to 15 to model check the *ACP* adaptive program. We have successfully verified all of the above properties. All experiments were evaluated on a Lenovo Thinkpad computer with 1.20GHz Intel Core 2 Duo Processor and 4GB of RAM, running Windows 7.

The normalization of ACP has 24 non-adaptive programs and 144 adaptations among them. The GLTS of normalized ACP is shown in Fig. 10. The verification result, execution time and memory usage of each global property is listed in Table 1.

In order to study the scalability of our approach, we synthesized a series of normalized adaptive programs by duplicating the non-adaptive programs n times (where n is 2,4,8) in the normalized *ACP* program. We measured the execution time and memory consumption of each global property on the synthesized adaptive programs. Fig. 11a shows the experimental results of the average execution time of model checking for the above seven global properties, where the x-axis represents the number of non-adaptive programs in the normalized adaptive programs and the y-axis represents the

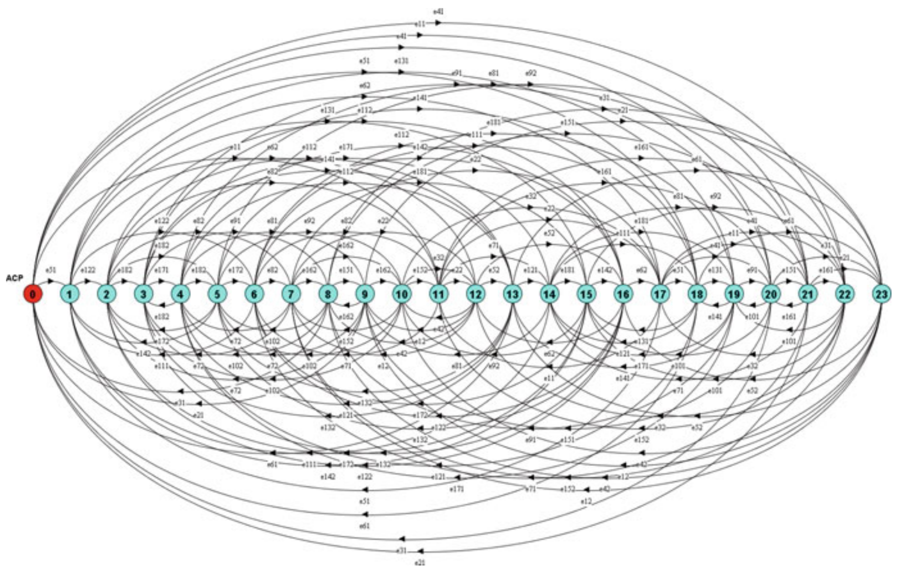


Fig. 10 The GLTS of normalized ACP

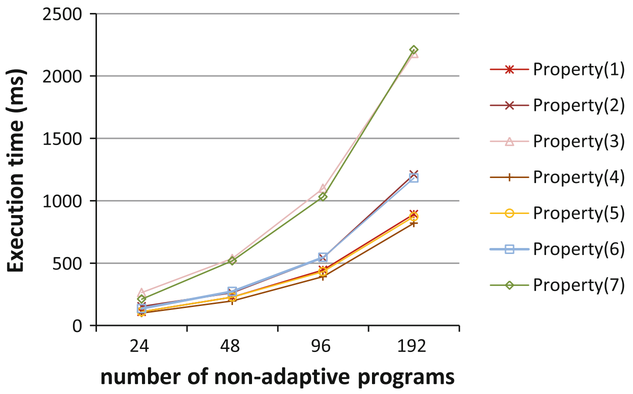
Table 1 Performance of mLTL model checking on ACP

Global property	Verification result	Execution time (ms)	Memory usage (MB)
Property(1) (Eq. 9)	TRUE	110	36
Property(2) (Eq. 10)	FALSE	153	33
Property(3) (Eq. 11)	TRUE	100	35
Property(4) (Eq. 12)	FALSE	264	33
Property(5) (Eq. 13)	TRUE	110	37
Property(6) (Eq. 14)	TRUE	136	34
Property(7) (Eq. 15)	TRUE	211	37

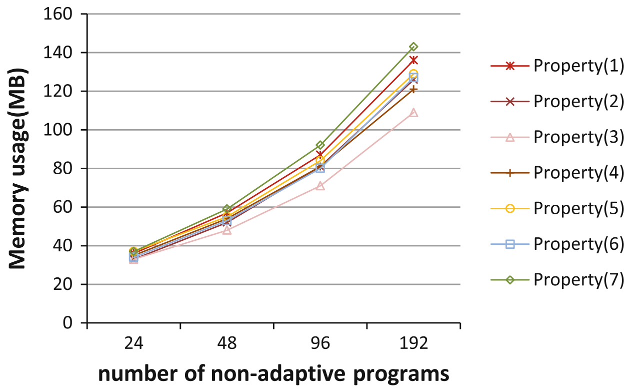
time elapsed during executions. From the results we noticed that the time consumed by our approach is linear to the number of non-adaptive programs in a normalized adaptive program. Since there are two LTL formulae in the property (4) and (7), the execution time of the two properties are about two times of other properties. Figure 11b shows the memory usage where the x-axis represents the number of non-adaptive programs in the normalized adaptive program and the y-axis represents the memory occupied during model checking. From these results, we can see that the memory usage of our approach is approximately linear to the number of non-adaptive programs in a normalized adaptive program.

7 Related work

Numerous techniques have been proposed to address various issues in formalizing adaptation. Bradbury et al. [12] discussed various approaches based on graphs, process algebras, logic and other formalisms are used to specify adaptive systems. Graph-based



(a) Execution time changes with number of non-adaptive programs



(b) Memory usage changes with number of non-adaptive programs

Fig. 11 Performance of mLTL model checking

approaches [5,25,42] use graph rewriting rules to specify dynamism. Process Algebra approaches use a variety of process algebras such as Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP), and π-calculus to describe the semantics of interactions and reconfigurations of components and connectors [1,15,27,39]. Architectural Description Language (ADL) [34] models adaptive software as components and connectors, and adaptation as reconfiguration of connections. Dynamic reconfiguration enables changing the architectures of adaptive software at runtime. Generally, these approaches have focused on structural adaptation. A few efforts, including those that use process algebras to specify the behavior of adaptive programs [1,15,22], have formally specified the behavioral changes of adaptive programs.

Zhang et al. [47–49] proposed a model-based development approach for dynamically adaptive software. An adaptive program is modelled as the composition of a finite number of steady-state programs and the adaptations among these programs.

Three commonly occurring basic adaptation models help to construct the adaptations of the adaptive program from one domain to another. A-LTL [47], an adaptation-based extension to linear temporal logic specifies the above three commonly used adaptation semantics. By specification composition, they specify the whole requirements for an adaptive program. Their modular verification approach [49] not only reduces the verification complexity by a factor of n, where n is the number of steady-state programs encompassed by the adaptive program, but also further reduces verification cost by supporting verification of incrementally developed adaptive software. mLTL model checking has the same space complexity. Since the normalized adaptive program has n multiple non-adaptive programs than the adaptive program, the time complexity of mLTL model checking is n multiple than the modular approach for A-LTL.

Goal-based modeling [20,24,36,46] has also been used for specifying dynamically adaptive software. A common feature of these works is that they assume that all adaptation choices are known and enumerated at design time. Hence, unanticipated adaptations are difficult to specify and analyze. Self-adaptive systems are dynamic in nature and allow for changes in the manner in which they adapt throughout their deployment lifetime. As a result of this dynamism, some behavioral modes reached during adaptation may be undesirable and harmful to the overall system goals. Georgas et al. [18] improved the dependability of such systems by developing facilities for adaptation recording, enhancing the configuration visibility over the entire system lifetime, and providing user-driven support for architectural recovery from undesirable configurations. However, mLTL does not require all possible alternative adaptations. The unknown behavior of adaptive programs at runtime may be monitored and the execution traces are generated and analyzed by a model checker to verify its conformance to the formal specification [6]. The runtime verification of adaptive software [19] may be used to solve the model checking problem. Similarly, the adaptive program defined in this paper can be generated at runtime, and mLTL can be used to define formal specification.

For the purpose of modelling and verification of software, different models of state machine have been proposed. The statecharts are hierarchical finite state machines [21], in which vertices can be ordinary states, or superstates which are FSMs themselves. The superstates offer a convenient mechanism to specify systems in a stepwise refinement manner and FSMs need to be specified only once and can then be reused in different contexts. Various approaches [2,16,17,35,43,45] have been proposed to the formal verification of statecharts using model checking. Most approaches rely on translating the hierarchical structure into the flat representation of the input language of the model checker. In this paper, our model has only two levels and the global specification distinguishes different FSMs. Maraninchi and Rémond [30,31] proposed an automata model in real-time systems considering the independent running modes, which are similar to the behavioral modes in this paper. Model checking of their automata does not support specifying mode related specifications either.

8 Conclusion

In this paper, we propose an approach to formal development of global specifications for dynamically adaptive programs. mLTL does not require all possible alternative adaptations to be specified during designing systems. mLTL integrated with LTL and A-LTL could specify the adaptive software effectively.

We believe that this paper offers several contributions in the domain of adaptive software by providing an approach to specify and verify the global properties for future ultra large scale systems. Global specification provides a novel way to specify adaptive software and thus enables a new perspective. It is simple and sufficiently effective for continuous evolution and unknowable behavior when designing complex software systems in the future.

It is noticed that several issues require further investigations. We may extend FSP language to support modelling of dynamic adaptation for intuitionistic description and develop an extension of LTSA for mLTL model checking. Model checking algorithm in this paper is not so efficient by the reason of separated verification of non-adaptive programs on LTL formulae, and more efficient model checking algorithm needs to be developed. Our approach only focuses on the behavioral aspect of adaptive programs, and the importance of expressing adaptations at the architectural level is thus realized. Applying our approach with an appropriate ADL representation will provide a more comprehensive solution to the development of adaptive programs.

Acknowledgments This work has been supported in part by the National Natural Science Foundation of China (NSFC) under the Grant No.61003017 and the Project of National Laboratory of Software Development Environment under the Grant No.SKLSDE-2010ZX-05.

References

1. Allen R, Douence R, Garlan D (1998) Specifying and analyzing dynamic software architectures. In: Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering, Springer Verlag, p 21
2. Alur R, Yannakakis M (2001) Model checking of hierarchical state machines. ACM Trans Program Lang Syst 23(3):273–303
3. Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secur Comput 1(1):11–33
4. Baier C, Katoen JP (2008) Principles of model checking. MIT Press, Cambridge
5. Baresi L, Heckel R, Thöne S, Varro D, Varró D, Milano PD (2004) Style-based refinement of dynamic software architectures. In: Proceeding of 4th Working IEEE/IFIP Conference on Software Architecture, IEEE, pp 155–164
6. Barringer H, Goldberg A, Havelund K, Sen K (2004) Program monitoring with ltl in eagle. In: Proceedings of 18th IEEE International Parallel and Distributed Processing Symposium, IEEE Computer Society, Washington, DC, p 264
7. Bauer A, Leucker M, Schallhart C (2010) Comparing ltl semantics for runtime verification. J Logic Comput 20(3):651–674
8. Benedetti M, Cimatti A (2003) Bounded model checking for past ltl. In: Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'03, Springer, Berlin/Heidelberg, pp 18–33
9. Bertrand D, Déplanche AM, Faucou S, Roux OH (2008) A study of the aadl mode change protocol. In: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, Washington, DC, pp 288–293
10. Biyani KN, Kulkarni SS (2008) Assurance of dynamic adaptation in distributed systems. J Parallel Distrib Comput 68(8):1097–1112
11. Bodden E (2004) A lightweight ltl runtime verification tool for java. In: the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA '04, ACM, New York, pp 306–307

12. Bradbury JS, Cordy JR, Dingel J, Wermelinger M (2004) A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, ACM, New York, pp 28–33
13. Broy M, Leuxner C, Sitou W, Spanfelner B, Winter S (2009) Formalizing the notion of adaptive system behavior. In: Proceedings of the (2009) ACM symposium on Applied Computing. ACM, New York, pp 1029–1033
14. Bruni R, Corradini A, Gadducci F, Lluch Lafuente A, Vandin A (2012) A conceptual framework for adaptation. In: Lara J, Zisman A (eds) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol 7212. Springer, Berlin/Heidelberg, pp 240–254
15. Canal C, Pimentel E, Troya JM (1999) Specification and refinement of dynamic software architectures. In: Proceedings of the TC2 First Working IFIP Conference on Software Architecture. Kluwer B.V., Deventer, pp 107–126
16. Chan W, Anderson RJ, Beame P, Burns S, Modugno F, Notkin D, Reese JD (1998) Model checking large software specifications. IEEE Trans Softw Eng 24(7):498–520
17. Clarke EM, Heinle W (2000) Modular translation of statecharts to smv. Carnegie-Mellon University School of Computer Science, Tech. rep., Pittsburgh
18. Georgas JC, van der Hoek A, Taylor RN (2005) Architectural runtime configuration management in support of dependable self-adaptive software. ACM SIGSOFT Softw Eng Notes 30:1–6
19. Goldsby HJ, Cheng BH, Zhang J (2008a) Models in software engineering. In: Chap AMOEBA-RT: Run-Time Verification of Adaptive Software. Springer, Berlin/Heidelberg, pp 212–224
20. Goldsby HJ, Sawyer P, Bencomo N, Cheng BHC, Hughes D (2008b) Goal-based modeling of dynamically adaptive system requirements. In: Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, IEEE Computer Society, ECBS '08, Washington, DC, pp 36–45
21. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274
22. Kramer J, Magee J (1998) Analysing dynamic change in software architectures: a case study. In: Proceedings of the 4th International Conference on Configurable Distributed Systems, IEEE Computer Society, Washington, DC, p 91
23. Kramer J, Magee J (2007) Self-managed systems: an architectural challenge. In: Future of software engineering. IEEE Computer Society, Washington, DC, pp 259–268
24. Lapouchnian A, Yu Y, Liaskos S, Mylopoulos J (2006) Requirements-driven design of autonomic application software. In: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, CASCON '06, IBM Corp., Riverton
25. Le Métayer D (1998) Describing software architecture styles using graph grammars. IEEE Trans Softw Eng 24(7):521–533
26. Lichtenstein O, Pnueli A (1985) Checking that finite state concurrent programs satisfy their linear specification. In: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, New Yrok, pp 97–107
27. Magee J, Kramer J (1996) Dynamic structure in software architectures. In: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering. ACM, New York, pp 3–14
28. Magee J, Kramer J (2006) Concurrency: state models & Java programs. Wiley, New York
29. Malek S, Edwards G, Brun Y, Tajalli H, Garcia J, Krka I, Medvidovic N, Mikic-Rakic M, Sukhatme GS (2010) An architecture-driven software mobility framework. J Syst Softw 83(6):972–989
30. Maraninchi F, Rémond Y (1998) Mode-automata: about modes and states for reactive systems. In: Proceedings of the 7th European Symposium on Programming. Springer, New York, pp 185–199
31. Maraninchi F, Rémond Y (2003) Mode-automata: a new domain-specific construct for the development of safe critical systems. Sci Comput Program 46(3):219–254
32. Martin L, Christian S (2009) A brief account of runtime verification. J Logic Algebr Program 78(5):293–303
33. McKinley PK, Sadjadi SM, Kasten EP, Cheng BHC (2004) Composing adaptive software. Computer 37(7):56–64
34. Medvidovic N, Taylor RN (2000) A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng 26(1):70–93
35. Mikk E, Lakhnech Y, Siegel M, Holzmann GJ (1998) Implementing statecharts in promela/spin. In: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques, IEEE Computer Society, WIFT '98, Washington, DC, pp 90

36. Morandini M, Penserini L, Perini A (2008) Modelling self-adaptivity: a goal-oriented approach. In: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, IEEE Computer Society, SASO '08, Washington, DC, pp 469–470
37. Morin B, Barais O, Nain G, Jezequel JM (2009) Taming dynamically adaptive systems using models and aspects. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, pp 122–132
38. Northrop L, Feiler P, Gabriel R, Goodenough J, Linger R, Kazman R, Schmidt D, Sullivan K, Wallnau K (2006) Ultra-large-scale systems-the software challenge of the future. Software Engineering Institute, Carnegie Mellon University, Tech. rep., Pittsburgh
39. Oquendo F (2004) π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Softw Eng Notes 29(3):1–14
40. Sadjadi SM, McKinley PK, Kasten EP (2003) Architecture and operation of an adaptable communication substrate. In: Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems, IEEE Computer Society, p 46
41. Salehie M, Tahvildari L (2009) Self-adaptive software: Landscape and research challenges. ACM Trans Auton Adap Syst 4(2):1–42
42. Taentzer G, Goedicke M, Meyer T (1998) Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In: Proceedings of 6th International Workshop on Theory and Application of Graph Transformations, Springer, Berlin, pp 179–193
43. Thums A, Schellhorn G, Ortmeier F, Reif W (2004) Interactive verification of statecharts. In: Ehrig H, Damm W, Desel J, Groe-Rhode M, Reif W, Schnieder E, Westkmper E (eds) Integration of Software Specification Techniques for Applications in Engineering, Lecture Notes in Computer Science, vol 3147. Springer, Berlin/Heidelberg, pp 355–373
44. Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification. In: Proceedings of 1st IEEE Symposium on Logic in Computer Science, IEEE Computer Society, Cambridge, pp 332–344
45. Varró D (2002) A formal semantics of uml statecharts by model transition systems. In: Corradini A, Ehrig H, Kreowski H, Rozenberg G (eds) Graph Transformation, Lecture Notes in Computer Science, vol 2505. Springer, Berlin / Heidelberg, pp 378–392
46. Yu Y, Lapouchnian A, Liaskos S, Mylopoulos J, Leite J (2008) From goals to high-variability software design. In: An A, Matwin S, Ras Z, Slezak D (eds) Foundations of Intelligent Systems, Lecture Notes in Computer Science, vol 4994. Springer, Berlin/Heidelberg, pp 1–16
47. Zhang J, Cheng BH (2006a) Model-based development of dynamically adaptive software. In: Proceedings of the 28th International Conference on Software Engineering, ACM, New York, pp 371–380
48. Zhang J, Cheng BH (2006b) Using temporal logic to specify adaptive program semantics. J Syst Softw 79(10):1361–1369
49. Zhang J, Goldsby HJ, Cheng BHC (2009) Modular verification of dynamically adaptive systems. In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, ACM, New York, pp 161–172