Abstract—This paper is motivated by the challenge to traditional development process of embedded systems from the evolution of dependability requirements, which leads to manual analysis and revision of system designs at design-time or post-implementation at a high cost, especially when the target system is complex or large. This paper proposes a complementary methodology, namely the model monitoring approach, to fill in the gap between the evolution of dependability requirements and traditional development process. The novel approach models functional and dependability requirements separately, and contains two alternative implementation techniques: model monitoring and model generating. The paper illustrates the methodology with examples and comparison with the model checking approach, to show better support of the evolution throughout the life-cycle at a lower cost.

Keywords—requirements evolution, dependability, reliability, safety, Büchi automata, model checking, model monitoring

I. INTRODUCTION

This research is motivated by the challenge to traditional development process from the change and evolution of dependability requirements. The changes are common both at design-time and post-implementation, especially for the system whose life period is long, e.g., aircrafts, nuclear plants, critical embedded electronic systems, etc. Unfortunately, the changes always cause high expenditure of rechecking and revising the system, especially when the system is too complex to be clearly analyzed manually or so large that the revision is not trivial. Moreover, the changes may not only lead to modify a small portion of the system, but to revise the entire design. We are searching for a technique supporting changeful dependability requirements at a lower cost.

It is well known that the evolution of requirements is common and challenges the practice of system developing and maintenance [1]. The evolution may be a result of different types of changes [2]. The changes may result from the experience of using the system after implementation and distributing, dynamic and turbulent environments, requirements elicitation, new regulations, etc. As a result, engineers must take into account these changes and revise their system post-implementation, of course at a high cost. The evolution of requirements threw down the gauntlet to the traditional development process. We must support the evolution throughout the entire life-cycle from various aspects [3]. We are interested in the innovation on the system development methodology for better support of the evolution.

Requirements include the following two classes:

1. Functional requirements, pertaining to all functions that are to be performed by the target system.

2. Dependability requirements, which are requirements about the dependable operation of the target system. Dependability requirements contain safety requirements, security requirements, reliability requirements, etc. For example, safety requirements [4] focus on safety constraints specifying authorized system behaviors and components interactions, where a safety constraint specifies a specific safeguard [5]. A system may have different dependability requirements under different contexts or critical levels. For instance, control software is imposed more strict constraints than entertainment software, even if both are embedded in aircrafts.

This research is interested in the evolution of dependability requirements, the latter one of the two classes above. There are two common causes of the changes to dependability requirements.

First, dependability requirements may change at design-time. The two types of requirements are defined by different groups in industrial practice, i.e., system engineers and dependability engineers, respectively. At the beginning stage of system developing, dependability engineers often find it very difficult to produce a complete set of dependability requirements. Thus, they add emergent dependability requirements during the development process, along with their increasing knowledge on the design.

Second, dependability requirements may change post-implementation. Some dependability requirements were unknown before the system is developed and used in real environment. For example, people always need to learn new safety requirements from historical events [6], [7]. Moreover, safety regulations change several times during the life-cycle as people requires a safer system. However, it will be expensive to modify the design after we learn these requirements, since the product has been released.

In the development process, dependability requirements are always modeled as property specification. As a result, the changes discussed above will cause the evolution and
change of property specification.

In the past thirty years, model checking [8] achieved great success as a verification technique for ensuring property specification. It has been widely used to verify reactive systems, by specifying and checking properties written in temporal logic formulas [9].

The major disadvantages of model checking techniques under the evolution of property specification are the following two. First, the analysis of counterexamples and revision of designs are not automated. If the system is complex, it is difficult to locate the faults and revise the design without introducing new faults. As a result, the verification process is iterated until no fault is detected, thus increases the cost. Second, once new properties are introduced or existing properties are modified, the whole design or implementation (product) must be revised or redistributed at a high cost even impossible, especially when the system is very large.

Motivated by the need of improving the drawbacks, we propose the model monitoring approach to fill in the gap between the evolution of property specification and traditional verification process. The novel approach models the functional requirements and dependability requirements separately. Then two alternative techniques, namely model monitoring and model generating, can be applied to ensure the properties, and improve the two mentioned disadvantages of model checking in the context of evolution.

This paper is organized as follows. A standard model checking technique related to our work is recalled in Section II. Then an example is used to illustrate the model checking approach and our new approach in Section III. The comparison shows the differences between them. In Sections IV and V, the formal foundation and framework of the model monitoring approach are introduced. We discuss related work in Section VI, and conclude in Section VII.

II. PRELIMINARY: MODEL CHECKING

In this section, we will briefly recall the notations and fundamentals of standard automata-theoretic model checking technique. An exhaustive investigation on this subject could be found in the monograph [8].

The first step of model checking is constructing a formal model $M$ that captures the behavior of system. We always use the Kripke structure.

**Definition 1:** Let $AP$ be a set of atomic propositions. A Kripke structure $M$ over $AP$ is a tuple $(S, S_0, R, L)$, where $S$ is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, $L : S \rightarrow 2^{AP}$ is a labeling function that labels each state with the set of atomic propositions true in that state.

The second step is to specify the property to be verified $\phi$ in a certain temporal logic, typically LTL (Linear Temporal Logic), CTL (Computation Tree Logic), CTL* [9]. We use the path quantifiers $A$ ("for all computation paths") and $E$ ("for some computation path"), and the temporal operators $X$ ("next state"), $F$ ("in the future"), $G$ ("globally"), $U$ ("until") and $R$ ("release").

Given the model $M$ and the formula $\phi$, the model checking problem is to decide whether $S_0 \subseteq \{ s \in S \mid M, s \models \phi \}$. There are two popular solutions: symbolic approach and automata-theoretic approach. The automata-theoretic approach translates the formula $\phi$ into an automaton for checking. The SPIN [10] is a typical implementation.

![Figure 1. The Process of Model Checking](image)

For the convenience of comparison, we are concerned with the automata-theoretic approach. The overall process of model checking is shown in Fig. 1 where the steps are numbered. This method is based on Büchi automata [11], [12], [13].

**Definition 2:** A (nondeterministic) Büchi automaton is a tuple $A = (Q, \Sigma, \Delta, q_0, F)$, where $Q$ is a finite set of states, $\Sigma$ is a finite alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is a set of transitions, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of accepting states.

A run of $A$ on an $\omega$-word $v = v(0)v(1)\ldots \in \Sigma^\omega$ is a sequence of states $\rho = \rho(0)\rho(1)\ldots \in Q^\omega$ such that $\rho(0) = q_0, (\rho(i), v(i), \rho(i + 1)) \in \Delta$ for $i \geq 0$. Let $inf(\rho)$ be the set of states that appear infinitely often in the run $\rho$, then $\rho$ is a successful run if and only if $inf(\rho) \cap F \neq \emptyset$. A accepts $v$ if there is a successful run of $A$ on $v$. The $\omega$-language recognized by $A$ is $L(A) = \{ v \in \Sigma^\omega \mid A \text{ accepts } v \}$.

If a language $L = L(A)$ for some Büchi automaton $A$, then $L$ is Büchi recognizable. Büchi recognizable $\omega$-languages are called regular $\omega$-languages. The expressive power of regular $\omega$-languages includes that of LTL [12], although Büchi automata are syntactically simple. Thus, we can translate LTL formulas into Büchi automata.

At the third and fourth steps, the modeled system $M$ and the property $\phi$ are both represented in Büchi automata, respectively.

A Kripke structure $M = (S, R, S_0, L)$ is translated into an automaton $A_M = (S \cup \{ q_0 \}, \Sigma, \Delta, \{ q_0 \}, S \cup \{ q_0 \})$, where $q_0 \not\in S$ and $\Sigma = 2^{AP}$. We have $(s, a, s') \in \Delta$ for $s, s' \in S$ if and only if $(s, s') \in R$ and $a = L(s')$, and $(q_0, a, s) \in \Delta$ if and only if $s \in S_0$ and $a = L(s)$.

The negation of the property $\neg \phi$ in LTL is translated into an automaton $A_{\neg \phi}$ over the same alphabet $2^{AP}$ [14]. $L(A_{\neg \phi})$ includes exactly the $\omega$-words satisfying $\neg \phi$. Note that each edge of $A_{\neg \phi}$ is annotated with a boolean expression that represents several sets of atomic propositions, where each set corresponds to a truth assignment for $AP$ that satisfies the boolean expression. For example, let $AP = \{ a, b, c \}$, an edge labeled $a \land b$ matches the transitions...
labeled with \{a, b\} and \{a, b, c\}. We denote by \(\Sigma(a \land b) = \{\{a, b\}, \{a, b, c\}\}\).

The mathematical foundation of checking is the following one. The system \(A_M\) satisfies the specification \(\phi\) when \(L(A_M) \subseteq L(A_\phi)\). Therefore, one checks whether \(L(A_M) \cap L(A_\phi) = \emptyset\), since \(L(A_\phi) = \Sigma^* - L(A_\phi)\). If the intersection is not empty, any behavior in it corresponds to a counterexample.

At the fifth step, we compute the automaton \(A_I\) accepting \(L(A_M) \cap L(A_\phi)\) (denoted by \(A_I = A_M \cap A_\phi\)). Finally, at the sixth step, the last task is to check emptiness of the intersection \(A_I\). A memory efficient algorithm, namely double DFS (Depth First Search) [15], was developed by extending Tarjan’s DFS. If \(L(A_I) = \emptyset\), then the system \(M\) satisfies the property \(\phi\). Or else, a counterexample \(v \in L(A_I)\) will be reported, and guide the revision of the original design (step 7).

On one hand, revisions of the design may bring in new faults. On the other hand, model checkers always produce only one counterexample each time, indicating a single fault. Thus, the iterative process of model checking, counterexample analysis and revision will be repeated, until \(L(A_I) = \emptyset\). Note that it is difficult to locate the fault if the design is complex or large. As a result, due to the complexity and size of the system, the cost of manual analysis of counterexamples and revision is high.

III. EXAMPLE: OVEN AND MICROWAVE OVEN

In this section, we will illustrate the model checking approach and our model monitoring approach by treating the same example. Then we compare the two approaches to show intuitively the differences between them.

Let us consider the behaviors of an oven and a microwave oven. They have similar operations: start oven, open door, close door, heat, etc. One significant difference is that we can use an oven with its door open. We should only use a microwave oven with its door closed for avoiding the damaging effects of radiation. Suppose we have a design of an oven, we want to reuse this design for producing microwave ovens. We must impose additional constraints (a case of evolution of property specification). For example, we add a constraint: “the door must be closed when heating”.

Figure 2 extracts the Kripke model \(M\) of a design of oven. The atomic propositions are: \(s\) (start), \(c\) (door is closed) and \(h\) (heat), i.e., \(AP = \{s, c, h\}\). Each state is labeled with both the atomic propositions that are true and the negations of the propositions that are false in the state.

A. The Model Checking Approach

We aim at checking the property “when the oven is heating, the door must be closed”, i.e., the LTL formula \(\phi \overset{def}{=} G(h \rightarrow c)\). For clarity and avoiding complex figures, we use this simple formula. The approach can be automated and scaled up for more complex models and formulas. We are concerned with the automata-theoretic model checking which is syntactically similar to our approach (but there is no semantic similarity).

First, \(M\) is translated into a Büchi automaton \(A_M\) of Fig. 3 by adding an initial state \(q_0\). Each transition is labeled with its name \(p_i\) and the associated terminal \(a \in \Sigma = 2^{AP}\). Each state is an accepting state.

Second, the negation of \(\phi\) is translated into a Büchi automaton \(A_{\neg \phi}\) of Fig. 4. For \(\neg \phi\), we have

\[
\neg \phi = \neg G(h \rightarrow c) = \neg (\text{False} R (\neg h \lor c)) = \text{True} U (h \land \neg c)
\]

Thus, \(A_{\neg \phi}\) of Fig. 4 is constructed recursively from \(\text{True} U (h \land \neg c)\). Each transition is labeled with a boolean expression, representing a set of terminals that corresponds to a truth assignment for \(AP\) that satisfies the boolean expression. Specifically, let \(\Sigma(\varphi)\) be the set of terminals making \(\varphi\) true, we have \(\Sigma(h \land \neg c) = \{(h), \{s, h\}\}\). \(\Sigma(\text{True}) = 2^{AP}\). We may represent \(A_{\neg \phi}\) by labeling its transitions with terminals in \(\Sigma\) instead of boolean expressions, i.e., the automaton of Fig. 5, with a similar style to Fig. 3.

Then, we compute the intersection \(A_I = A_M \cap A_{\neg \phi}\). The double DFS algorithm is called to decide the emptiness of \(A_I\) on-the-fly. A subgraph of the intersection constructed by the algorithm is shown in Fig. 6. A state \(t_{ij}\) denotes a composite state \((q_i, r_j)\). The first DFS (solid transitions) starts from \(t_{00}\) to \(t_{73}\), then the second DFS (dashed transitions) is started at \(t_{73}\). A successor \(t_{83}\) is detected on the stack of the first DFS, then the algorithm reports a counterexample. Thus, \(L(A_I) \neq \emptyset\).

Finally, engineers must manually analyze the original design with the guide of the reported counterexample, locate the faults in the design, and revise the design. The iterative process of model checking, counterexample analysis and revision is executed, until \(L(A_I) = \emptyset\). Note that if the system is complex or large, the analysis of counterexample and revision would be hard, because it is difficult to locate the fault. As a result, due to the complexity and size of the system, the cost of manual analysis of counterexamples and revision is high.

B. The Model Monitoring Approach

We also need the automaton \(A_M\) of Fig. 3. However, we need a controlling automaton \(A_{\phi}\) instead of \(A_{\neg \phi}\). A controlling automaton is a Büchi automaton having an alphabet that equals the set of transitions of the controlled automaton \(A_M\), i.e., \(A_{\phi} = (Q', \Sigma', \Delta', q'_0, F')\) with \(\Sigma' = \Delta\) where \(\Delta\) is the set of transitions of the controlled automaton \(A_M\). The controlling automaton can be constructed from the property specification directly, or by translating the automaton \(A_{\phi}\) (resulting in an alphabet-level controlling automaton).

For this example, we use \(A_{\phi}\) which is translated from \(\phi\) using the translation from LTL formulas to automata. For \(\phi\),
we have
\[
\phi = \mathbf{G}(h \rightarrow c) = \mathbf{F} \mathbf{a} \mathbf{s} \mathbf{e} \mathbf{R}(-h \lor c)
\]

The constructed automaton \(A_\phi\) is shown in Fig. 7.

We translate \(A_\phi\) into an alphabet-level controlling automaton \(\hat{A}_\phi\), by replacing each boolean expression \(\varphi\) by \(\Delta(\varphi)\), which is the set of names of the transitions labeling the terminal that corresponds to a truth assignment that satisfies \(\varphi\). For example, each transition of \(\hat{A}_\phi\) in Fig. 8 is labeled with a set of names of transitions of \(A_M\), where
\[
\Delta(-h) = \{p_1, p_2, p_3, p_4, p_8, p_9, p_{13}, p_{14}, p_{15}\}
\]
\[
\Delta(c) = \{p_2, p_4, p_5, p_6, p_7, p_8, p_{14}\}
\]

Then we compute the meta-composition \(C\) of \(A_M\) and the controlling automaton \(\hat{A}_\phi\), denoted by \(C = A_M \mathbf{e} \hat{A}_\phi\). The automaton \(C\) of Fig. 9 starts from the initial state \(t_{00}\), a transition is allowed if and only if it is allowed by both \(A_M\) and \(\hat{A}_\phi\). Note that the hazardous transitions \(p_{10}, p_{11}, p_{12}\) and the unreachable transition \(p_{13}\) are eliminated. The model \(C\) satisfies the property \(\phi\). We can recover it to a Kripke model \(M'\) of Fig. 10 by removing the initial state. It is easy to see the model \(M'\) satisfies the required property.

The model monitoring approach contains two alternative techniques to implement the meta-composition operator as follows.

One alternative, namely model monitoring or explicit model monitoring for emphasis, implements \(A_M\) and \(\hat{A}_\phi\) separately, which constitute an overall correct system. Specifically, \(\hat{A}_\phi\) can be realized as a controlling system that monitors the behavior of the system implementing \(A_M\) at runtime. If \(A_M\) tries to apply a certain action that violates the properties, the control system can detect it and call some
achieved since the computation can be automated.
Another alternative, namely model generating or implicit
model monitoring, implements directly the automatically
generated model $M'$ as a correct model of microwave ovens,
rather than separating the two implementations. In this case,
$\bar{A}_\phi$ monitors implicitly the behavior of $A_{M'}$.

C. Technical Comparison

Technically, we use a controlling automaton $\bar{A}_\phi$ rather
than the automaton $A_{\neg \phi}$ specifying $\neg \phi$. We use meta-
composition rather than intersection and emptiness checking.

Model checking leads to revise the original design by
manually analyzing the counterexample violating the new
property. Thus the cost is high, especially when the system
is so large and complex that the counterexamples cannot be
effectively analyzed.

Model monitoring uses the automaton of Fig. 3, and adds
a controlling system implementing the automaton of Fig. 8.
The global system in Fig. 9 satisfies the new property. Note
that $\bar{A}_\phi$ is usually much smaller than the overall system, it is
easier and cheaper to modify only the controlling component
when the property specification evolve at the later stages of
life-cycle.

Model generating can automatically generate a new cor-
correct design (Fig. 10) by computing the meta-composition of
the automata in Fig. 3 and Fig. 8. Higher efficiency can be
achieved since the computation can be automated.

We remark here that $\bar{A}_\phi$ of Fig. 8 is an alphabet-level
controlling automaton, i.e., all the transitions associated with
the same terminal of $A_{M'}$ appear together on the transitions
between any two states of $\bar{A}_\phi$. In fact, all the controlling
automata translated from LTL formulas are alphabet-level
controlling automata. The feature of alphabet-level control
and the translation from LTL formulas are not necessary
for constructing controlling automata. We will show that
controlling automata are more flexible and can be defined
directly. These unnecessary features are just used for the
comparison with model checking in this example.

IV. THE BüCHI AUTOMATON CONTROL SYSTEM

In this section, we will introduce the definition and
properties of the Büchi automaton control system, which
is the formal foundation of the model monitoring approach,
and interpret some theoretical results.

A. The Formalism

A Büchi automaton control system consists of a Büchi
automaton and a Büchi controlling automaton.

Definition 3: Given a controlled Büchi automaton (or
simply automaton) $A_1 = (Q_1, \Sigma_1, \Delta_1, q_1, F_1)$, with a set of
transitions $\Delta_1 = \{p_i\}_{i \in I}$ where $p_i$ is a name of transition,
a Büchi controlling automaton (or simply controlling
automaton) over $A_1$ is a tuple $A_2 = (Q_2, \Sigma_2, \Delta_2, q_2, F_2)$
with $\Sigma_2 = \Delta_1$. $L(A_2)$ is a controlling $\omega$-language.

Note that each transition has a name. For example, assume
$\Delta_1(q_1, a) = \{q_2, q_3\}$, we denote by $p_i : (q_1, a, q_2) \in \Delta_1$ and
$p_j : (q_1, a, q_3) \in \Delta_1$ with two names $p_i, p_j$.

Definition 4: A Büchi automaton control system (sim-
ply BAC system) includes an automaton $A_1$ and a control-
ling automaton $A_2$, denoted by $A_1 \Rightarrow A_2$.

A run of $A_1 \Rightarrow A_2$ on an $\omega$-word $v = v(0)v(1)\ldots \in \Sigma^*$
contains:

- a sequence of states $\rho_1 = \rho_1(0)\rho_1(1)\ldots$
- a sequence of transitions $\sigma = \sigma(0)\sigma(1)\ldots$
- a sequence of controlling states $\rho_2 = \rho_2(0)\rho_2(1)\ldots$

such that $\rho_1(0) = q_1$, $\sigma(i) : (\rho_1(i), v(i), \rho_1(i + 1)) \in \Delta_1$
for $i \geq 0$, and $\rho_2(0) = q_2$, $\rho_2(j), \sigma(j), \rho_2(j + 1) \in \Delta_2$
for $j \geq 0$. Let $\inf(\rho_1)$ and $\inf(\rho_2)$ be the sets of states
that appear infinitely often in the sequences $\rho_1$ and $\rho_2$,
respectively. Then the run is successful if and only if
$\inf(\rho_1) \cap F_1 \neq \emptyset$ and $\inf(\rho_2) \cap F_2 \neq \emptyset$. $A_1 \Rightarrow A_2$ accepts
$v$ if there is a successful run on $v$. The global $\omega$-language
recognized by $A_1 \Rightarrow A_2$ is $L(A_1 \Rightarrow A_2) = \{v \in \Sigma^* | A_1 \Rightarrow A_2$
accepts $v\}$.

The symbol $\Rightarrow$ is called “meta-composition”, denoting the
left operand is controlled by the right operand.

Two trivial types of controlling automata are empty con-
trolling automata and full controlling automata. The former
ones accept the empty controlling language which rejects
all the sequences of applied transitions, i.e., $L(A_2) = \emptyset$.
The latter ones accept full controlling languages that accept
all the sequences of applied transitions, i.e., $L(A_2) = \Delta_1^*$,
where $\Delta_1$ is the set of transitions of the controlled automaton
$A_1$. Note that the two types of languages are both regular
$\omega$-languages.

Let us consider the closure property of Büchi automata
under the operator meta-composition.

Theorem 5: Given two Büchi automata $A_1 = (Q_1, \Sigma_1, \Delta_1, q_1, F_1)$ and
$A_2 = (Q_2, \Sigma_2, \Delta_2, q_2, F_2)$ with
$\Sigma_2 = \Delta_1$. We can construct an automaton $A = A_1 \Rightarrow A_2$
such that $L(A) = L(A_1 \Rightarrow A_2)$ as follows:

$A = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma_1, \Delta_1, (q_1, q_2, 0), Q_1 \times Q_2 \times \{2\})$

where $((q_i, q_j, x), a, (q_m, q_n, y)) \in \Delta$ if and only if $p :$
$q_i, a, q_m \in \Delta_1$, $(q_j, p, q_n) \in \Delta_2$, and $x, y$ satisfy the
following conditions:

$y = \left\{ \begin{array}{ll}
0, & \text{if } x = 2 \\
1, & \text{if } x = 0 \text{ and } q_m \in F_1 \\
2, & \text{if } x = 1 \text{ and } q_n \in F_2, \text{ or if } q_m \in F_1 \text{ and } q_n \in F_2 \\
x, & \text{otherwise}
\end{array} \right.$

Proof: The transitions $\Delta$ guarantees a transition in $\Delta_1$
is allowed by $\Delta_2$. The third component of $Q_1 \times Q_2 \times \{0, 1, 2\}$
is responsible for guaranteeing that accepting states from
both $A_1$ and $A_2$ appear infinitely often. According to Def. 4, it is easy to see $A$ accepts exactly $L(A_1 \searrow A_2)$.

If all of the states of $A_1$ are accepting, the computation can be simplified as follows.

**Theorem 6:** Given two Büchi automata $A_1 = (Q_1, \Sigma_1, \Delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma_2, \Delta_2, q_2, F_2)$ with $\Sigma_2 = \Delta_1$. We can construct an automaton $A$ such that $L(A) = L(A_1 \searrow A_2)$ as follows:

$$A = A_1 \searrow A_2 = (Q_1 \times Q_2, \Sigma_1, \Delta, (q_1, q_2), Q_1 \times F_2)$$

where $((q_i, q_j), (a, (q_m, q_n))) \in \Delta$ if and only if $p : (q_i, a, q_m) \in \Delta_1$, $(q_j, p, q_n) \in \Delta_2$.

It is easy to see the generative power of BAC systems and the one of Büchi automata are equivalent. Formally, let $BA$ be the family of Büchi automata, and $L(X)$ be the family of languages recognized by a set $X$ of automata, we have the following theorem.

**Theorem 7:** $L(BA \searrow BA) = L(BA)$.

Proof: (i), $L(BA \searrow BA) \subseteq L(BA)$ follows immediately from Thm. 5.

(ii), $L(BA \searrow BA) \supseteq L(BA)$. Given a Büchi automaton $A_1$, we can have $L(A_1 \searrow A_2) = L(A_1)$ by using a controlling automaton $A_2$ accepting a full controlling language.

B. The Alphabet-level BAC System

Let us consider a special family of controlling automata.

**Definition 8:** Given two Büchi automata $A_1 = (Q_1, \Sigma_1, \Delta_1, q_1, F_1)$ and $A_2 = (Q_2, \Sigma_2, \Delta_2, q_2, F_2)$ with $\Sigma_2 = \Delta_1$. $A_2$ is an alphabet-level controlling automaton, if the following condition is satisfied: if $(q_i, p, q_j) \in \Delta_2$ where $p : (q_m, a, q_n) \in \Delta_1$, then for all the transitions $p_k : (q_x, a, q_y) \in \Delta_1$ associated with the terminal $a$, there exists $(q_i, p, q_j) \in \Delta_2$. The system $A_1 \searrow A_2$ is an alphabet-level Büchi automaton control system (A-BAC).

Obviously, the A-BAC system is a special case of the BAC system. Thus, its generative power is not greater than the BAC system. We would like to say that the BAC system is of transition-level. The BAC system is more flexible, because it is not required that all the transitions associated with the same terminal in $\Delta_1$ must appear together between any two states of $A_2$.

Furthermore, it is worth noting that the BAC system can express some constraints outside the power of A-BAC systems. We use a simple example to show the difference between BAC and A-BAC systems in expressive power.

Suppose a nondeterministic vending machine $A_1$ (see Fig. 11) that receives coin $a$ and dispenses three types of goods $b, c, d$ (inputs are marked by $\cdot$, while outputs are marked by $!$). The machine may receive a coin $a$, then output $b, c, d$ in order. It may also output only $b, d$. The nondeterministic choice is made by the machine, depending on the luck. Obviously, the machine accepts the $\omega$-language $L(A_1) = (abd + abed)^\omega$.

To increase profit, people decide to modify the behavior by adding a new requirement: the dispensation of $bcd$ will be no longer allowed, only $bd$ can be dispensed. That is, the transition $p_3$ will not be allowed.

It is easy to implement a BAC system with the controlling module $A_2$ accepting $(p_1 + p_2 + p_4 + p_5)^\omega$. Thus, the new overall system accepts the $\omega$-language $L(A_1 \searrow A_2) = (abd)^\omega$.

However, there does not exist an A-BAC system satisfying the new requirement. The key problem is the nondeterminism in the state $q_1$. Assume there is such an alphabet-level controlling automaton $A_2$. Then $A_2$ can only specify whether $b$ can occur in a state which is reached after $a$ occurs. If no, then neither of $p_2$ and $p_3$ is allowed, i.e., no goods will be dispensed. If yes, then both of $p_2$ and $p_3$ are allowed. In this case, to avoid the occurrence of $abcd$, $A'$ may disable the transition dispensing $c$ in the next state. However, the machine may enter a dead state $q_1$ after choosing $p_3$ and dispensing $b$. As a result, the overall system dispenses only $b$ and entered a dead state. This does not conform to the requirement, since $d$ is not dispensed and the machine is no longer live. Therefore, $A_2$ does not exist.

Thus, we conclude that the BAC system is more flexible than the A-BAC system. We formalize the result as follows.

Let LTL-A-BCA be the family of alphabet-level Büchi controlling automata translated from LTL formulas, A-BCA be the family of alphabet-level Büchi controlling automata, BCA be the family of Büchi controlling automata. The following theorem characterizes their difference in expressive power.

**Theorem 9:** Given a Büchi automaton $A$,

(i) for any LTL-A-BCA $A_\phi$, there exists an A-BCA $\hat{A}_\phi$ such that $L(A \searrow \hat{A}_\phi) = L(A \searrow \hat{A}_\phi)$. The reverse does not hold.

(ii) for any A-BCA $\hat{A}_\phi$, there exists a BCA $\hat{A}$ such that $L(A \searrow \hat{A}_\phi) = L(A \searrow \hat{A})$. The reverse does not hold.

Proof: (i) follows from the fact that the expressive power of LTL is strictly included in that of regular $\omega$-languages [12]. (ii) follows from Def. 8 and the example above.

C. Interpretation

Theorem 5 ensures that the meta-composition can be also implemented as a reactive system in model generating. Theorem 7 ensures that any reactive system can be represented in the form of BAC systems. Since all of the states of $A_M$
constructed from $M$ are accepting states, theorem 6 can simplify the computation.  

Theorem 9 shows that the BAC system is more expressive than LTL model checking in specifying correctness properties, since model checking uses the LTL formula which is equivalent to LTL-A-BCA. The example in Section III uses only the alphabet-level controlling automaton translated from LTL (i.e., an LTL-A-BCA) for the convenience of comparison.

Theorem 9 shows also that the BAC system is more expressive than the tools checking regular properties. A regular property specifies a regular $\omega$-language. Therefore, these tools (e.g., SPIN [10]) use the correctness properties represented by a Büchi automaton, which is equivalent to an alphabet-level Büchi controlling automaton (A-BCA).

As we explained in Thm. 9, the feature of alphabet-level control and the translation from LTL formulas are not necessary, and restricts the expressive power of BAC systems. We can define directly the controlling automaton, c.f. the example before Thm. 9.

V. THE MODEL MONITORING APPROACH

The framework of the model monitoring approach is shown in Fig. 12. It consists of the following steps:

1. Model monitoring (or explicit model monitoring): $A_1$ and $A_2$ are separately implemented, but maybe at different stages of life-cycle. That is, $A_2$ can be added or modified at later stage of life-cycle. The system $A_1$ is controlled at runtime by $A_2$ which reports, blocks or recovers unsafe actions of the controlled system. We can incrementally add new correctness properties to the controlling system $A_2$ after we learn new requirements. Note that we do not really implement $A = A_1 \triangleright A_2$. Instead, $A_1$ and $A_2$ constitute a global system that is equivalent to $A$. If dependability requirements change, $A_1$ will not be modified. We only need to revise $A_2$, which is much easier and more efficient than model checking which leads to revise the whole system.

2. Model generating (or implicit model monitoring): $A_1$ and $A_2$ are combined at design-time to generate a new correct model $A = A_1 \oplus A_2$. Then we implement the model $A$ where $A_2$ implicitly monitors $A_1$. If dependability requirements change, we only need to revise $A_2$, then regenerate and implement a new specification $A'$. Because the computation of meta-composition can be automated, it is more efficient than manual analysis of counterexample and revision in model checking.

As a whole, we generally call the two alternative techniques the model monitoring approach. In both of the two cases, dependability requirements are design-oriented. The properties are implemented directly or integrated into the design. This is different to model checking, where dependability requirements are more testing-oriented (used to verify a system design).

VI. RELATED WORK

A standard automata-theoretic model checking technique, developed by Vardi, Holzmann et al. [15][16][14][10], was recalled in Section II. We provided an example to show the difference between our approach and model checking. The comparison in Section III shows that there is no semantic similarity between them, although they are syntactically similar due to the formalism of Büchi automata.

To conclude, the essence of our approach is the separation of functional requirements and dependability requirements, while model checking emphasizes integration. Thanks to the approach, people may achieve lower cost of revising designs when the dependability requirements change, because only the controlling component needs modifications.

As we discussed in Section IV, the interpretation of the theoretical results shows that the BAC system is more expressive than the model checking techniques that use LTL formulas or regular properties in specifying properties. Furthermore, model monitoring is often significantly more practical than model checking, since it explores one computational path at a time, while model checking suffers the state explosion problem because of the exhaustive search.

It is important to note that the model monitoring approach and model checking have their own advantages, and should be used complementarily.

The model monitoring approach separates the models of a system and its controlling system related to dependability requirements. Thus, the evolution of property specification only results in modifying the design of controlling system. Since the controlling system is always much smaller than the whole system, the cost of modification is lower. This approach is especially useful in ensuring changeful dependability requirements.
The model checking approach integrates dependability requirements into the system design through automated counterexample searching, manual analysis and revision. As a result, the evolution of property specification brings high cost of revising the design. Thus, model checking is more suitable for verifying invariable dependability requirements.

Therefore, invariable dependability requirements should be ensured through model checking, while changeful ones should be implemented through the model monitoring approach. Note that classifying dependability requirements into the two categories (i.e., invariable and changeful) is a trade-off process. The classification depends on the individual system and the experiences of designers. Discussions on this issue are beyond the scope of this paper.

Furthermore, model checking could be used to verify the global BAC system, in order to increase our confidence in the design of BAC systems.

The BAC system is also different from Ramadge and Wonham’s supervisory control [17], where the supervisor is similar to the alphabet-level controlling automaton. That means, our controlling automata, which specify properties on the transitions instead of the alphabet, have stronger expressive power in specifying constraints.

VII. Conclusion

In this paper, the model monitoring approach is proposed to fill in the gap between model checking and the evolution of dependability requirements. The model monitoring approach is based on a formalism named the Büchi automaton control system (BAC). We showed that the BAC system is more expressive than the model checking techniques that use LTL formulas or regular properties in specifying properties.

Our earlier work in [18] introduced the interface automaton control system (IN-AC system), which should be considered as a variant of the BAC system in the context of concurrency. The BAC system is theoretically more fundamental, and can clearly characterize the semantic difference between our approach and model checking through their syntactical similarity.

References


