

Towards Better Support for the Evolution of Safety Requirements

Zhe Chen [†]

[†] LATTIS & LAAS-CNRS, Université de Toulouse
135 Avenue de Rangueil
31077 Toulouse, France
zchen@insa-toulouse.fr

Gilles Motet ^{†,‡}

[‡] Foundation for an Industrial Safety Culture
6 Allée Emile Monso
31029 Toulouse, France
gilles.motet@insa-toulouse.fr

ABSTRACT

The research is motivated by the challenge from the evolution of safety requirements, which leads to revision of system designs at design-time or post-implementation at a high cost. This paper proposes complementary techniques, namely model monitoring and model generating, to better support the evolution throughout the life-cycle at a lower cost.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications;
D.2.4 [Software Engineering]: Software/Program Verification—*reliability*

General Terms

Design, Reliability

Keywords

requirements evolution, safety

1. INTRODUCTION

This research is motivated by the challenge to traditional verification process from the change and evolution of safety 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. We are searching for a technique supporting changeable safety 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 [3]. We must support the evolution throughout the entire life-cycle from various aspects [2]. We are interested in the system development methodology for better support of the evolution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE 2010 Cape Town, South Africa.

Copyright 2010 ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

As we know, requirements include two subtypes: *functional requirements* and *safety requirements*, which are requirements about the safe operation of the target system. This research is interested in the evolution of safety requirements, the latter one of the two classes. There are two common causes of the changes to safety requirements.

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

Second, safety requirements may change post-implementation. Some safety requirements were unknown before the system is developed and used in real environment. People always need to learn new safety requirements from historical events. 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.

Since the safety requirements will be modeled as correctness properties for model checking, the changes discussed above will cause the evolution of correctness properties.

Due to page limitation, the reader is assumed to be familiar with the automata-theoretic model checking technique [1] (the overall process is shown in Fig. 1).

The major disadvantages of model checking techniques under the evolution of correctness properties 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 fault 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 correctness properties are introduced or existing correctness 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, this paper proposes complementary techniques, namely *model monitoring* and *model generating*, to fill in the gap between the evolution of safety requirements and traditional verification process. The novel techniques model the functional requirements and correctness properties separately. Then two alternative modes can be applied to ensure the correctness properties.

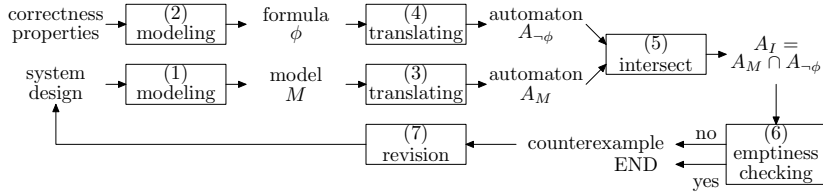


Figure 1: The Process of Model Checking

For the first mode, namely *model monitoring*, the implementation of correctness properties is separated from the implementation of functional requirements. A controlling system realizing the correctness properties controls the behavior of the target system. The two systems constitute a correct system from a global view. The interest of this approach is that we only need to revise the controlling system to guarantee correctness, when the safety requirements change. Since the controlling system is generally smaller than the overall system, the cost of analysis and revision will be much lower.

For the second mode, namely *model generating* or *implicit model monitoring*, a new system specification satisfying the correctness properties can be automatically generated. We need only implement directly the generated correct specification. The computation can be automated, thus the cost of modifying the design will be lower.

The two modes improve the two mentioned disadvantages of model checking in the context of evolution.

This paper is organized as follows. In Section 2, an example is treated to demonstrate our new approach and its difference from model checking. In Sections 3 and 4, the formal foundation and framework of model monitoring and model generating are introduced, respectively. We discuss related work in Section 5, and conclude in Section 6.

2. EXAMPLE: OVEN AND MICROWAVE OVEN

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 can 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 oven. We must impose additional safety constraints (a case of evolution of safety requirements). 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.

We aim at ensuring the correctness property “when the oven is heating, the door must be closed”, i.e., the LTL formula $\phi \stackrel{\text{def}}{=} \mathbf{G}(h \rightarrow c)$. For clarity and avoiding complex figures, we use this simple formula.

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.

Then we construct a controlling automaton A'_ϕ . 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 Δ is the set of transitions of the controlled automaton A_M . The controlling automaton can be constructed from the correctness property directly, or by translating the automaton A_ϕ , which is translated from ϕ using the translation from LTL formulas to automata (resulting in an alphabet-level controlling automaton).

The constructed automaton A_ϕ is shown in Fig. 4. A_ϕ is translated into an alphabet-level controlling automaton A'_ϕ , by replacing each boolean expression φ by $\Delta(\varphi)$, which is the set of names of the transitions labeling with the terminal that corresponds to a truth assignment that satisfies φ . For example, each transition of A'_ϕ in Fig. 5 is labeled with a set of names of transitions of A_M , where

$$\begin{aligned} \Delta(-h) &= \{p_1, p_2, p_3, p_4, p_8, p_9, p_{13}, p_{14}, p_{15}\} \\ \Delta(c) &= \{p_2, p_4, p_5, p_6, p_7, p_8, p_{14}\} \end{aligned}$$

Then we compute the meta-composition C of A_M and the controlling automaton A'_ϕ , denoted by $C = A_M \cdot A'_\phi$. The automaton C starts from the composite state (q_0, r_0) , a transition is allowed if and only if it is allowed by both A_M and A'_ϕ . 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 ϕ . We can recover it to a Kripke model M' of Fig. 6 by removing the initial state. It is easy to see the model M' satisfies the required property.

The approach of *model monitoring* is the following one. We do not directly implement M' . Instead, A_M and A'_ϕ are implemented separately, and they constitute a correct system from a global view. Specifically, A'_ϕ can be realized as a control system that monitors the behavior of the system implementing A_M . If A_M tries to apply a certain action that violates the correctness property, the control system can detect it and call some predefined functions to alert, block, or recover from the unsafe action.

Another mode, namely *model generating* or *implicit model monitoring*, implements directly the automatically generated model M' as a correct design of microwave ovens.

The approach of model checking. The automata-theoretic approach [4][5] (see Fig. 1) translates the negation of ϕ into a Büchi automaton $A_{-\phi}$, then computes the intersection $A_I = A_M \cap A_{-\phi}$. The double Depth First Search algorithm is called to decide the emptiness of A_I on-the-fly. Since $L(A_I) \neq \emptyset$, the algorithm reports a counterexample.

Finally, engineers must manually analyze the original design with the guide of the reported counterexample, locate the errors in the design, and revise the errors. 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

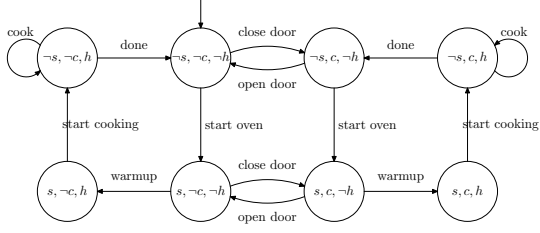


Figure 2: The Kripke Model M of Oven

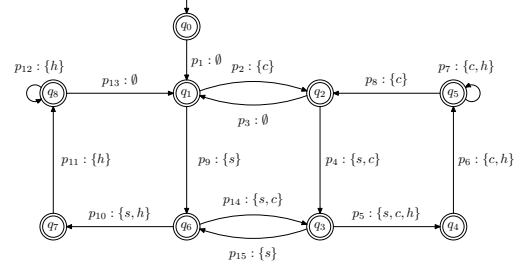


Figure 3: The Büchi Automaton A_M of Oven

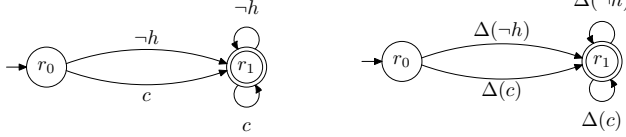


Figure 4: The Büchi Automaton A_ϕ of ϕ

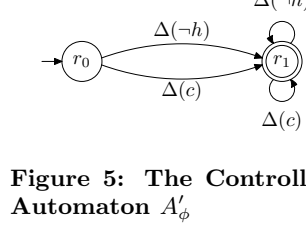


Figure 5: The Controlling Automaton A'_ϕ

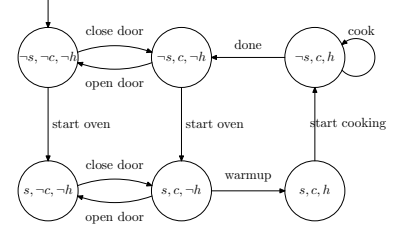


Figure 6: The Kripke Model M' of Microwave Oven

and revision will be repeated, until $L(A_I) = \emptyset$. Note that it is difficult to analyze the counterexample and locate the fault, if the system 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.

Comparison with model checking. Technically, we use a controlling automaton A'_ϕ rather than the automaton $A_{-\phi}$ 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 correctness 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. 5. The global system in Fig. 6 satisfies the new safety property. Note that A'_ϕ is usually much smaller than the overall system, it is easier and cheaper to modify only the controlling component when the safety requirements evaluate at the later stages of life-cycle.

Model generating can automatically generate a new correct design (Fig. 6) by computing the meta-composition of the automata in Fig. 3 and Fig. 5. Higher efficiency can be achieved since the computation can be automated.

We remark here that A'_ϕ of Fig. 5 is an alphabet-level controlling automaton, i.e., all the transitions associated with the same terminal of A_M appears together on the transitions between any two states of A'_ϕ . In fact, all the controlling automata translated from LTL formulas are alphabet-level controlling automata. The alphabet-level feature and the translation from LTL formula are not necessary for constructing controlling automata. We will see in the sequel 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.

3. THE BAC SYSTEM

DEFINITION 1. 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 $\Delta_1 = \{p_i\}$ where p_i is a name of transition, a *controlling Büchi 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 ω -language*. \square

DEFINITION 2. A *Büchi automaton control system* (simply BAC system) includes an automaton A_1 and a controlling automaton A_2 , denoted by $A_1 \vec{\tau} A_2$.

A run of $A_1 \vec{\tau} A_2$ on an ω -word $v = v(0)v(1)\dots \in \Sigma_1^\omega$ contains:

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

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 $\text{inf}(\rho_1)$ and $\text{inf}(\rho_2)$ be the sets of states that appear infinitely often in the sequences ρ_1 and ρ_2 , respectively. Then the run is successful if and only if $\text{inf}(\rho_1) \cap F_1 \neq \emptyset$ and $\text{inf}(\rho_2) \cap F_2 \neq \emptyset$. $A_1 \vec{\tau} A_2$ accepts v if there is a successful run on v . The *global ω -language* recognized by $A_1 \vec{\tau} A_2$ is $L(A_1 \vec{\tau} A_2) = \{v \in \Sigma_1^\omega \mid A_1 \vec{\tau} A_2 \text{ accepts } v\}$. \square

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

THEOREM 3. 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 $L(A) = L(A_1 \vec{\tau} A_2)$ as follows:

$$A = A_1 \vec{\tau} A_2 = (Q_1 \times Q_2 \times \{0, 1, 2\}, \Sigma_1, \Delta, (q_1, q_2, 0), Q_1 \times Q_2 \times \{2\})$$

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 condi-

tions:

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

Proof. The transitions Δ guarantees a transition in Δ_1 is allowed by Δ_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. 2, it is easy to see A accepts exactly $L(A_1 \dot{\neg} A_2)$. \square

Theorem 3 ensures that the meta-composition can be also implemented as a reactive system in model generating.

Let us consider a special family of controlling automata.

DEFINITION 4. 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 $(q_i, p_k, q_j) \in \Delta_2$.

The system $A_1 \dot{\neg} A_2$ is an **alphabet-level Büchi automaton control system (A-BAC)**. \square

Obviously, the A-BAC system is a special case of 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 Δ_1 must appear together between any two states of A_2 .

The example in the previous section uses only the alphabet-level controlling automaton (Def. 4) for the convenience of comparison. As we explained, the special case is not necessary. We can define directly the controlling automaton.

4. THE PROCESS OF MODEL MONITORING AND MODEL GENERATING

In the process, the system model A_1 and the controlling model A_2 are specified separately, where A_2 contains *correctness semantics*, which defines what a system is authorized to do. Then we have the following two choices of implementing the meta-composition to deduce a correct system:

1. 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 safety requirements. Note that we do not really implement $A = A_1 \dot{\neg} A_2$. Instead, A_1 and A_2 constitute implicitly a global system that is equivalent to A . If safety 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: A_1 and A_2 are combined at design-time to generate a new correct model $A = A_1 \dot{\neg} A_2$. Then we implement this model. If safety 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 both of the two cases, safety requirements are *design-oriented*. The properties are implemented directly or integrated into the design. This is different to model checking, where safety requirements are more *testing-oriented* (used to verify a system design).

5. RELATED WORK

Our approach was compared with the automata-theoretic approach for model checking developed by Holzmann, Vardi et al. [4][5] in Section 2. We provided an example to show the difference between our approach and model checking. The comparison shows that there is no semantic similarity between them, although they are syntactically similar due to the formalism of Büchi automata. Furthermore, model monitoring is often significantly more practical than model checking, since it explores one computational path at a time.

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

It is important to note that model monitoring/generating and model checking have their own advantages, and should be used complementarily. That is, invariable safety requirements should be ensured through model checking, while changeful ones should be implemented through model monitoring/generating.

6. CONCLUSION AND FUTURE WORK

In this paper, model monitoring and model generating are proposed to fill in the gap between model checking and the evolution of safety requirements, which are common at design-time and post-implementation in practice.

This research comes from and will contribute to industrial practice. Thus, as future work, we are cooperating with industrial projects to obtain more empirical results and further develop the methodology. Of course, another future direction is exploring the theoretic properties of the BAC system.

7. REFERENCES

- [1] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.
- [2] N. A. Ernst, J. Mylopoulos, Y. Yu, and T. Nguyen. Supporting requirements model evolution throughout the system life-cycle. In *Proceedings of the 16th IEEE International Requirements Engineering Conference (RE 2008)*, pages 321–322. IEEE Computer Society, 2008.
- [3] S. Harker, K. Eason, and J. Dobson. The change and evolution of requirements as a challenge to the practice of software engineering. In *Proceedings of IEEE International Symposium on Requirements Engineering (RE 1993)*, pages 266–272. IEEE Computer Society, 1993.
- [4] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [5] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.