2009 Fourth International Conference on Systems

Modeling System Safety Requirements Using Input/Output Constraint Meta-Automata

Zhe Chen [†]
[†] *Laboratory LATTIS, INSA, University of Toulouse*
135 Avenue de Rangueil, 31077 Toulouse, France
Email: zchen@insa-toulouse.fr

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

Abstract

Most recent software related accidents have been system accidents. To validate the absence of system hazards concerning dysfunctional interactions, industrials call for approaches of modeling system safety requirements and interaction constraints among components and with environments (e.g., between humans and machines). This paper proposes a framework based on input/output constraint meta-automata, which restricts system behavior at the meta level. This approach can formally model safe interactions between a system and its environment or among its components. This framework differs from the framework of the traditional model checking. It explicitly separates the tasks of product engineers and safety engineers, and provides a top-down technique for modeling a system with safety constraints, and for automatically composing a safe system that conforms to safety requirements. The contributions of this work include formalizing system safety requirements and a way of automatically ensuring system safety.

1. System Accidents

Computer technology has created a quiet revolution in most fields of engineering, also introduced new failure modes that are changing the nature of accidents [1]. To provide much more complex automated services, industrials have to develop much more complicated computer systems which consist of numerous components and a huge number of actions (both internal and interactive). A recent challenge is the *system accident*, caused by increasing *coupling* among system components (software, control system, electromechanical and human), and their *interactive complexity* [2][3]. In contrast, accidents arising from component failures are termed *component failure accidents*.

System safety and *component reliability* are different. They are system property and component property, respectively [2]. *Reliability* is defined as the probability that a component satisfies its specified behavioral requirements, whereas *safety* is defined as the absence of accidents — events involving an unacceptable loss [4]. People are now constructing intellectually unmanageable software systems that go beyond human cognitive limits, and this allows potentially unsafe interactions to be undetected. Accidents often result from hazardous interactions among perfectly functioning components.

There are several examples of system accidents.

The space shuttle *Challenger* accident was due to the release of hot propellant gases from a field joint. An O-ring was used to control the hazard by sealing a tiny gap in the field joint created by pressure at ignition. However, the design did not effectively impose the *required constraints* on the propellant gas release (i.e., it did not adequately seal the gap), leading to an explosion and the loss of the shuttle and its crew [5].

The self-destructing explosion of Ariane 5 launcher was resulted from the successive failures of the active inertial reference system (IRS) and the backup IRS [5]. Ariane 5 adopted the same reference system as Ariane 4. However, the profile of Ariane 5 was different from that of Ariane 4 — the acceleration communicated as input value to IRS of Ariane 5 was higher. Furthermore, the *interactions between IRS and other components* were not checked and redefined. Due to the overflow of input value computation, the IRS stopped working [6]. Then, the signaled error was interpreted as a launcher attitude, and led the control system to rotate the tailpipe at the end stop [7].

The loss of the Mars Polar Lander was attributed to a *misapprehensive interaction* between the onboard software and the landing leg system [8]. The landing leg system was expected and specified to generate noise (spurious signals) when the landing legs were deployed during descent. However, the onboard software interpreted these signals as an indication that landing occurred (as specified in their requirements) and shut the engines down, causing the spacecraft to crash into the Mars surface.

A system accident occurred in a batch chemical reactor in England [9]. The computer controlled the input flow of cooling water into the condenser and the input flow of catalyst into the reactor by manipulating the valves. The computer was told that if any component in the plant gets abnormal, it had to leave all controlled variables as they were

978-0-7695-3551-7/09 $25.00 © 2009 IEEE
DOI 10.1109/ICONS.2009.24

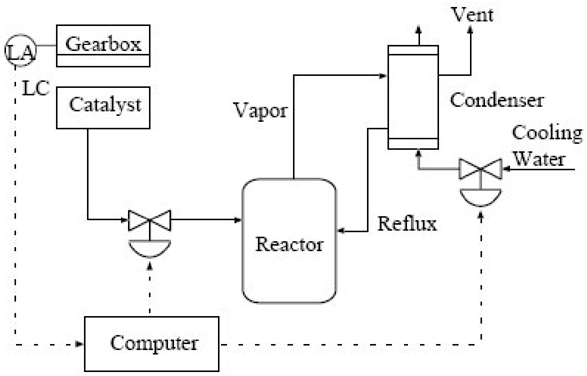


Figure 1. A Chemical Reactor Design

and to sound an alarm. On one occasion, the computer just started to increase the cooling water flow, after a catalyst had been added into the reactor. Then the computer received an abnormal signal indicating a low oil level in a gearbox, and it reacted as its requirements specified: sounded an alarm and maintained all the control variables with their present condition. Since the water flow was kept at a low rate, then the reactor overheated, the relief valve lifted and the contents of the reactor were discharged into the atmosphere. The design of the system is shown in Fig. 1.

In all these accidents, the components are reliable in terms of satisfying their specified requirements, but the systems are not safe as a whole. As Leveson mentioned in STAMP (Systems-Theoretic Accident Model and Processes) [1], these accidents result from inadequate *control* or enforcement of *safety-related constraints* of the systems.

Since most software related accidents have been system accidents [1], people need to model and constrain interactions of system components to validate the absence of *dysfunctional interactions*. One of the challenges to system safety is that *the ambiguity in safety requirements* may lead to reliable unsafe systems. Another challenge is *the lack of formal techniques* for describing safety rules and interactions between components (including interactions between humans and machines), which makes automated verifications difficult. In this paper, we will propose a formal framework for modeling system safety rules. The framework is based on a new concept of I/O constraint meta-automata.

This paper is organized as follows: the framework of our approach is proposed in Section 2. The formal technique based on I/O constraint meta-automata is introduced in Section 3. An example is used to illustrate how to formalize safety constraints and combine it with a system specification. In Section 4, we discuss how to apply this approach to a system that consists of multiple components. In Section 5, we compare our works to classic verification techniques, such as model checking, and conclude the paper.

2. The Framework of Our Approach

The most popular technique of system safety verification is *model checking* [10]. In this framework, we have two steps in verifying a system. At first, we formalize system behavior as a model (e.g., a transition system, a Kripke model [11]). At the second step, we specify the features that we aim at validating, and use a certain checking algorithm to search for a counterexample which is an execution trace violating the specified features. If the algorithm finds such a counterexample, we have to modify the original system to ensure safety requirements, or else the verification succeeds.

Unlike the model checking, our framework takes another way. It consists of the following steps:

1) Modeling system behavior, including specifications of its components, internal and external interactions.
2) Modeling system safety constraints using a certain formal technique, e.g., I/O constraint meta-automata in this paper.
3) Combining these two models to deduce a safe system model, that is, a system model whose behavior is in accordance with its safety constraints.

As we mentioned in [12], the system behavior specifies an *operational semantics*, which defines what a system is able to do. Modeling system behavior is mainly performed by *product engineers* (designers), such as programmers and developers. In the example of the chemical reactor control system, the actions "opening the catalyst flow", "opening the cooling water flow" and "sounding an alarm" are system behaviors.

In the second step, the model of safety requirements specifies a *correctness semantics*, which defines what a system is authorized to do. This process is the duty of *safety engineers* whose responsibility is to assure system safety. Safety engineers may consist of requirement engineers, test engineers, managers from higher socio-technical levels who define safety standards or regulations [1], etc. In the example of the chemical reactor system, the constraint "opening the catalyst flow must be followed by opening the cooling water flow" is an instance of system safety requirements.

In the third step, in order to ensure system safety, we combine the system model with its safety constraints model. Then we can check if the system is safe under the constraints specifying safety requirements. However, the precondition of such a formal checking is that we must formalize safety requirements. And we also need to carefully define the composition of a system model and its constraints model. In the next section, we will introduce such an approach based on I/O constraint meta-automata.

We remark here that an other precondition is that we can find all safety constraints in a system. However, this is an issue of "risk identification" [13], which is outside the scope of this paper. This work deals with "risk treatment", which is a different phase to risk identification, according to the ISO Standard 31000 [13].

3. Modeling System Safety Constraints Using I/O Constraint Meta-Automata

The theory of input/output automata [14][15] extends classic automata theory [16] for modeling concurrent systems with different input, output and internal actions. I/O automata and the variants are widely used in modeling distributed systems [17].

Definition 1: An **input/output automaton** (also called an I/O automaton or simply an automaton) is a tuple $A = (Q, \Sigma^I, \Sigma^O, \Sigma^H, \delta, S)$, where:

- Q is a set of **states**.
- $\Sigma^I, \Sigma^O, \Sigma^H$ are pairwise disjoint sets of **input, output and internal actions**, respectively. Let $\Sigma = \Sigma^I \bigcup \Sigma^O \bigcup \Sigma^H$ be the set of **actions**.
- $\delta \subseteq Q \times \Sigma \times Q$ is a set of **labeled transitions**, such that for each $a \in \Sigma^I$ and $q \in Q$ there is a transition $p_k : (q, a, q') \in \delta$ (**input-enabled**).
- $S \subseteq Q$ is a nonempty set of **start states**. ☐

In the graph notation, a transition $p_k : (q, a, q')$ is denoted by an arc from q to q' labeled $p_k : a$, where p_k is the name of the transition. To discriminate explicitly the different sets of actions in diagrams, we may suffix a symbol "?", "!" or ";" to an input, output or internal action, respectively.

In the example of the batch chemical reactor, the computer system behavior is modeled using an I/O automaton A of Fig. 2(1). The automaton A includes a set of input actions $\Sigma^I = \{l\}$ (low oil signal), a set of output actions $\Sigma^O = \{c, w, a\}$ (opening catalyst flow, opening water flow, sounding an alarm, respectively), and a set of internal actions $\Sigma^H = \{e\}$ (ending all operations). The normal operational behavior includes opening catalyst flow (p_1), then opening water flow (p_2), etc., resulting in an infinite execution trace $p_1 p_2 p_1 p_2 \ldots$. To respond to abnormal signals as soon as possible, all the states have a transition labeled l, which leads to a state that can sound an alarm (p_6) and stop the process (p_8). Unfortunately, this design leads to hazardous behaviors: $(cw)^* clae$, that is, after a sequence of opening catalyst and water flows $(cw)^*$, then the catalyst flow is opened c when an abnormal signal is received l, then an alarm is sounded a. So water is not added after the catalyst flow is opened. This sequence of events leads to the accident mentioned in Section 1.

Note that this hazard is due to the uncontrolled sequences of transitions — p_1 must be followed by p_2 and not by p_4. To solve this problem, we need to specify the authorized sequences (satisfying safety constraints) on the transitions δ and not on the actions Σ. Thus, these constraints are not at the behavioral model level, but at the meta-model level. We propose the concept of *constraint meta-automata* to formalize safety constraints. Then, we combine a meta-automaton with the system automaton.

Definition 2: A **constraint meta-automaton** (or simply meta-automaton) $\hat{A}$ over an I/O automaton $A = (Q, \Sigma, \delta, S)$ is a tuple $\hat{A} = (\hat{Q}, \hat{\Sigma}, \hat{\delta}, \hat{S})$, where:

- $\hat{Q}$ is a set of **states** disjoint with Q.
- $\hat{\Sigma}$ is a set of **terminals** that consists of all the transition names in δ of A.
- $\hat{\delta}$ is a set of **labeled transitions**.
- $\hat{S} \subseteq \hat{Q}$ is a nonempty set of **start states**. ☐

Note that the transitions δ of A are terminals of $\hat{A}$, so we say that $\hat{A}$ is at the meta level of A. Figure 3 illustrates the 3 levels in our framework. Let Σ^* be a set of execution traces of actions, A describes the behavior on Σ. $\hat{A}$ specifies the behavior on the A-transitions ($\hat{\Sigma} = \delta$), that is, a behavior on the behavior of A. This meta-behavior expresses safety requirements.

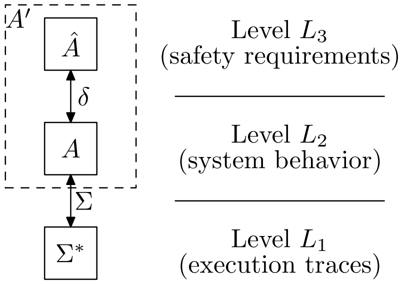


Figure 3. A 3-levels Overview

In the example, to prevent accidents, we need to bind the safety constraint "opening catalyst must be followed by opening water," that is, "whenever the transition $p_1 : c$ occurs, the transition $p_2 : w$ must occur after that." This constraint can be formalized as a constraint meta-automaton $\hat{A}$ of Fig. 2(2). When we design this constraint, we only specify the sequence of transitions p_1, p_2 at the meta-model level, and we concern little about the implementation of the system at the model level. The next step is to compose the system automaton A with its constraint meta-automaton $\hat{A}$, and automatically generate a system model A' satisfying the safety requirement.

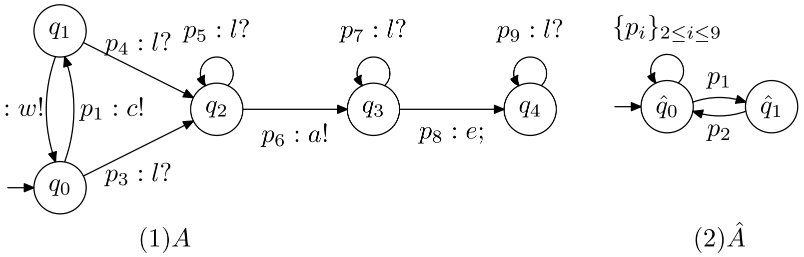
Definition 3: The **meta-composition** A' of an I/O automaton $A = (Q, \Sigma, \delta, S)$ and a constraint meta-automaton $\hat{A} = (\hat{Q}, \hat{\Sigma}, \hat{\delta}, \hat{S})$ over A is a tuple:

$$A' = A \overrightarrow{\cdot} \hat{A} = (Q \times \hat{Q}, \Sigma, \delta', S \times \hat{S}) \quad (1)$$

where $p_k : ((q_i, \hat{q}_j), a, (q_m, \hat{q}_n)) \in \delta'$ iff,
(1) $p_k : (q_i, a, q_m) \in \delta$, and
(2) $(\hat{q}_j, p_k, \hat{q}_n) \in \hat{\delta}$. ☐

The symbol $\overrightarrow{\cdot}$ is the *meta-composition operator*, and read as "meta-compose". Its left and right operands are an automaton and a constraint meta-automaton, respectively.

Notice that $\delta = \{p_k\}_{k \in \mathcal{K}}$ plays a key role in associating transitions of A and terminals of $\hat{A}$. For our example, we combine the automata A and $\hat{A}$ of Fig. 2, thus we get the automaton $A' = A \overrightarrow{\cdot} \hat{A}$ of Fig. 4 where q_{ij} denotes $(q_i, \hat{q}_j)$.



$(1)A$ $(2)\hat{A}$

Figure 2. Automata of the Reactor Control System

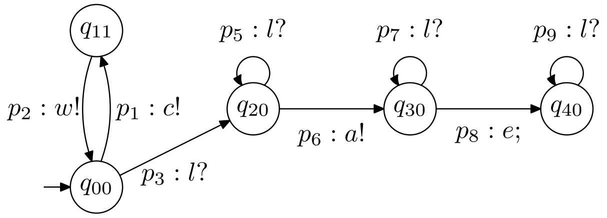


Figure 4. The Meta-Composition A'

The meta-composition contains exactly all the paths satisfying the constraint in the system. Formally, we have the following theorem (the proof is omitted for its simpleness and intuitiveness from the definition):

Theorem 4: Given $A, \hat{A}$ and the meta-composition A', an **execution trace** $t_\Sigma \in \Sigma^*$ is in A' iff, t_Σ is in A, and its **transition trace** $t_\delta \in \delta^*$ is in $\hat{A}$. $\square$

Obviously, the set of traces of A' is a subset of the traces of A. Formally, let $L(A)$ be the set of traces of A (also the language of A), we have $L(A') \subseteq L(A)$.

Thanks to $\hat{A}$, the hazardous execution traces, for example $cwclae$, which exists in A, will be eliminated, because its transition trace $p_1p_2p_1p_4p_6p_8 \notin L(\hat{A})$ (the language of $\hat{A}$). The comparison between A of Fig. 2(1) and A' of Fig. 4 highlights the hazardous transition p_4 of A. However, in general, this diagnosis is much more complex and cannot be achieved manually, since a real system A has too many states to be expressed clearly on a paper. That is why we should provide a formal and automated method for eliminating hazardous transitions.

4. Modeling Multi-Component Systems with Safety Constraints

Our approach can also be applied to the systems that are made up of several components, whose safety constraints are related to several components.

As a preliminary, we redefine the composition of I/O automata, which was introduced in [14].

Let $\mathcal{N} = \{n_1, ..., n_k\} \subseteq \mathbb{N}$ be a finite set with cardinality k, and for each $n \in \mathcal{N}$, S_n be a set. Then we define: $\prod_{n\in\mathcal{N}} S_n \overset{\text{def}}{=} \{(x_{n_1}, x_{n_2}, ..., x_{n_k}) \mid (\forall j \in \{1, ..., k\} \bullet x_{n_j} \in S_{n_j}) \wedge \mathcal{N} = \{n_1, ..., n_k\} \wedge (\forall j_1, j_2 \in \{1, ..., k\} \bullet j_1 < j_2 \rightarrow n_{j_1} < n_{j_2})\}$. We define the function of projection $\overrightarrow{s}[j]$ to denote the j-th component of the state vector $\overrightarrow{s}$: $\forall j \in \{1, ..., k\}$, $(x_{n_1}, x_{n_2}, ..., x_{n_k})[j] = x_{n_j}$.

Definition 5: A finite collection of I/O automata $\{A_n\}_{n\in\mathcal{N}}$ is said to be **strongly compatible** if $\forall i, j \in \mathcal{N}$, $i \neq j$, we have

(1) $\Sigma_i^O \cap \Sigma_j^O = \emptyset$, and

(2) $\Sigma_i^H \cap \Sigma_j = \emptyset$. $\square$

Definition 6: The **composition** $A = \prod_{n\in\mathcal{N}} A_n$ of a finite collection of strongly compatible I/O automata $\{A_n\}_{n\in\mathcal{N}}$ is an I/O automaton $(\prod_{n\in\mathcal{N}} Q_n, \Sigma^I, \Sigma^O, \Sigma^H, \delta, \prod_{n\in\mathcal{N}} S_n)$ iff,

- $\Sigma^I = \bigcup_{n\in\mathcal{N}} \Sigma_n^I - \bigcup_{n\in\mathcal{N}} \Sigma_n^O$,
- $\Sigma^O = \bigcup_{n\in\mathcal{N}} \Sigma_n^O$,
- $\Sigma^H = \bigcup_{n\in\mathcal{N}} \Sigma_n^H$, and
- for each $\overrightarrow{q}, \overrightarrow{q}' \in \prod_{n\in\mathcal{N}} Q_n$ and $a \in \Sigma$, $p_{\mathcal{I}} : (\overrightarrow{q}, a, \overrightarrow{q}') \in \delta$ iff $\forall j : 1 \leq j \leq |\mathcal{N}| \wedge n_j \in \mathcal{N}$,
  1) if $a \in \Sigma_{n_j}$ then $\exists i : i \subseteq \mathcal{I} \bullet p_i : (\overrightarrow{q}[j], a, \overrightarrow{q}'[j]) \in \delta_{n_j}$;
  2) if $a \notin \Sigma_{n_j}$ then $\overrightarrow{q}[j] = \overrightarrow{q}'[j]$ and $\forall i : p_i \in \delta_{n_j} \bullet i \cap \mathcal{I} = \emptyset$. $\square$

Notice that the name of a transition of A may contain a set of names of original transitions $p_{\mathcal{I}} = \{p_i\}_{i\subseteq\mathcal{I}}$, where i may be a set or a single element.

We use an example derived from [14], concerning a system composed of two components with interactions: a candy vending machine and a customer. The candy machine A_m, specified in Fig. 5(1), may receive inputs b_1, b_2 indicating that buttons 1 and 2 are pushed, respectively. It may output s, a, indicating candy dispensation actions, SKYBARs and ALMONDJOYs, respectively. The machine may receive several inputs before delivering a candy. A greedy user A_u, specified in Fig. 5(2), can push buttons b_1, b_2 or get a candy s, a. The greedy user does not wait for a candy bar before pressing a button again.

The composition of the machine behavior and the user behavior is defined by $A_{mu} = A_m \cdot A_u$ of Fig. 5(3), where q_{ij} denotes the composite state (m_i, u_j), $p_{i_1,...,i_k}$ denotes a set of transitions $\{p_{i_1}, p_{i_2}, \ldots, p_{i_k}\}$. A transition of the composition may be composed of several transitions of components. For example, $p_{1,15} : s$ is a synchronization of $p_1 : s!$

and $p_{15} : s?$, which belong to A_m and A_u, respectively. Formally, a transition of $A = \prod_{n \in \mathcal{N}} A_n$ may be composed of i transitions of components, where $1 \leq i \leq |\mathcal{N}|$.

In the context of composite transitions, a composite transition is allowed iff one of its sub-transitions is authorized by its constraint meta-automaton. Thus, we define the meta-composition operator as follows:

Definition 7: The **meta-composition** A' of a composition $A = \prod_{n \in \mathcal{N}} A_n = (\prod_{n \in \mathcal{N}} Q_n, \Sigma, \delta, \prod_{n \in \mathcal{N}} S_n)$ and a constraint meta-automaton $\hat{A} = (\hat{Q}, \hat{\Sigma}, \hat{\delta}, \hat{S})$ over A is a tuple:

$$A' = A \overrightarrow{\cdot} \hat{A} = ((\prod_{n \in \mathcal{N}} Q_n) \times \hat{Q}, \Sigma, \delta', (\prod_{n \in \mathcal{N}} S_n) \times \hat{S}) \quad (2)$$

where $p_{\mathcal{I}} : ((\overrightarrow{q_i}, \hat{q}_j), a, (\overrightarrow{q_m}, \hat{q}_n)) \in \delta'$ iff,

(1) $p_{\mathcal{I}} : (\overrightarrow{q_i}, a, \overrightarrow{q_m}) \in \delta$, and

(2) $\exists k : k \in \mathcal{I} \bullet (\hat{q}_j, p_k, \hat{q}_n) \in \hat{\delta}$. $\square$

Notice that the specification of the example allows a hazardous situation: the greedy user repeatedly pushes a single button without giving the machine a chance to dispense a candy bar (the transition labeled $p_{5,13} : b_1$ of q_{11} does not allow the transition (q_{11}, s, q_{00}) to be fired). To prevent this situation, the following constraints forbid successive occurrences of pressing a single button:

- Whenever one of the transitions p_3, p_5, p_7 (action b_1) occurs, the next transition must be not p_3, p_5, p_7.
- Whenever one of the transitions p_4, p_6, p_8 (action b_2) occurs, the next transition must be not p_4, p_6, p_8.

Differing from the previous example, the constraint needs to synchronize the actions of the machine and of the user.

Formalizing the constraints, the semantics of the constraint meta-automaton A_c of Fig. 6(1) is: whenever the user pushes a button, she or he cannot push it again, but may push the other button to change the choice, or wait for a candy bar.

Combining the whole system A_{mu} with its constraint A_c, we get the system $A' = (A_m \cdot A_u) \overrightarrow{\cdot} A_c$ in Fig. 6(2), where q_{ijk} denotes the composite state (m_i, u_j, c_k). All of its execution traces satisfy the constraint, and thus prevent the hazardous situation.

This simple example is good for demonstrating the principle, avoiding indigestible diagrams of automata. Since we formally defined the *meta-composition operator*, it can be easily implemented to be an automated tool. Thus, it can be applied to more complex systems.

5. Conclusion

We propose modeling system safety requirements formally using I/O constraint meta-automata. As we illustrated using the examples, this approach can formally model safe interactions between a system and its environments, or among its components. This framework differs from the one of the traditional model checking. It explicitly separates the tasks of product engineers and safety engineers, and provides a technique for modeling a system with safety constraints, and for automatically composing a safe system that conforms to safety requirements.

The essential ideas of our approach are the separation and formalization of the system specification A (behavioral requirements) and the safety constraints $\hat{A}$ (safety requirements). The automaton A handles inputs to produce outputs using activities depending on the states, whereas the meta-automaton $\hat{A}$ treats activities to produce the set of acceptable activities depending on safety requirements.

Our framework has different objective and uses different approaches to those of model checking. Model checking techniques use *a bottom-up approach* — it verifies execution traces Σ^* at the lower level L_1 to prove the correctness and safety of the system model A at the middle level L_2 (see Fig. 3). However, our proposal uses *a top-down approach* — we model safety requirements as acceptable sequences of transitions (δ^*) at the higher level L_3 to ensure the correct use of A. Then any execution trace (at L_1) that conforms to the meta-composition A' is definitely a safe execution. So the two techniques are complementary. Model checking may be used to reduce the fault likelihood, and our approach can be applied to avoid behavior that are not in accordance with some critical safety requirements.

Both linear time logic and branching time logic have been proved to be useful in checking properties of traces of classic automata [10]. Since I/O automata are extended from classic automata, these existing techniques can be easily applied to I/O automata with little modifications.

In the future, we will apply the approach to the variants of I/O automata, e.g., timed, hybrid, probabilistic, dynamic [17]. Our approach can also be applied to the systems specified using classic automata [16], since I/O automata are specific extensions of traditional automata.

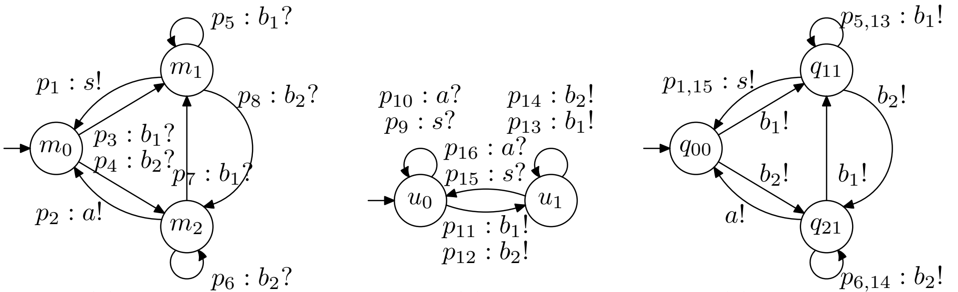
We will also study the formalization of parameterized constraints. To model parameterized systems more accurately, parameters of actions and value domains of variables should be considered. This is also the basis of studying reusability, substitutability and equivalence of components.

As we mentioned, identification of potential hazards is also a challenge in practice. Risk identification and treatment are both important phases in risk management [13]. This will be a good direction for future work.

References

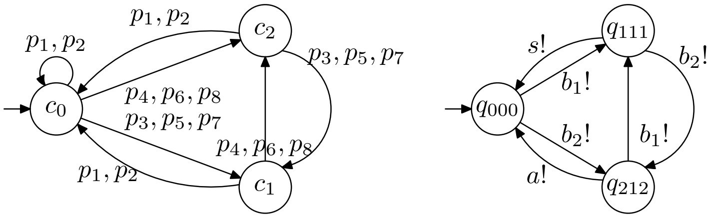
[1] N. Leveson, “A new accident model for engineering safer systems,” *Safety Science*, vol. 42, no. 4, pp. 237–270, 2004.

[2] N. Leveson, “Applying systems thinking to analyze and learn from events,” in *Workshop NeTWorK 2008*, 2008, available from http://sunnyday.mit.edu/papers/network-08.doc.



$(1)A_m$ $(2)A_u$ $(3)A_{mu}$

Figure 5. Automata of the Candy Machine System



$(1)A_c$ $(2)A'$

Figure 6. A Safety Constraint of the Candy Machine System

[3] C. Perrow, *Normal Accidents: Living with High-Risk Technologies*. Princetown University Press, USA, 1999.

[4] N. Leveson, *Safeware: System Safety and Computers*. Addison-Wesley, Reading, MA, 1995.

[5] N. Leveson, "Evaluating accident models using recent aerospace accidents," *Technical Report, MIT Dept. of Aeronautics and Astronautics*, 2001, available from http://sunnyday.mit.edu/accidents.

[6] T. Kohda and Y. Takagi, "Accident cause analysis of complex systems based on safety control functions," in *Proceedings of Annual Reliability and Maintainability Symposium (RAMS '06)*, 2006, pp. 570–576.

[7] J.-C. Geffroy and G. Motet, *Design of Dependable Computing Systems*. Kluwer Academic Publishers, 2002.

[8] J. S. R. Board, "Report on the loss of the mars polar lander and deep space 2 missions," *NASA Jet Propulsion Laboratory*, 2000.

[9] T. Kletz, "Human problems with computer control," *Plant/Operations Progress*, vol. 1, no. 4, 1982.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 2000.

[11] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems, Second Edition*. Cambridge University Press, 2004.

[12] G. Motet, "Risks of faults intrinsic to software languages: Trade-off between design performance and application safety," *Safety Science*, 2009, to appear.

[13] ISO/DIS, *ISO/DIS 31000: Risk management – Principles and guidelines on implementation*. International Standards Organization, 2008.

[14] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, vol. 2, no. 3, pp. 219–246, 1989, also available as MIT Technical Memo MIT/LCS/TM-373.

[15] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[16] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[17] N. A. Lynch, "Input/output automata: Basic, timed, hybrid, probabilistic, dynamic," in *Roberto M. Amadio, Denis Lugiez (Eds.): Proceedings of 14th International Conference on Concurrency Theory (CONCUR 2003), Lecture Notes in Computer Science 2761*. Springer, 2003, pp. 187–188.