2009 33rd Annual IEEE International Computer Software and Applications Conference

System Safety Requirements As Control Structures

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

Along with the popularity of software-intensive systems, the interactions between system components and between humans and software applications are becoming more and more complex. This results in system accidents related to system safety issues. System accidents are different to failures related to component reliability. System safety is not well addressed, because functional requirements and safety requirements are separately handled in practice. In this paper, we consider safety requirements as control structures that restrict system behaviors at meta-model level. We propose the formalism of interface C-Systems, short for "interface control systems". In this framework, functional requirements and safety requirements are separately formalized as interface automata and controlling automata respectively, as what we are doing in practice. The controlling automaton may guarantee safety requirements at design-time or run-time. Then the global system is a safe specification. The underlying mechanism differs from that of model checking. It explicitly separates the tasks of product engineers and safety engineers, and provides a new top-down methodology for designing and modeling a system with safety constraints, and for automatically composing a safe system that conforms to safety requirements. In practice, this methodology may be also used for safety checking, incident reporting and service restoration.

1. Introduction

Computer technology has created a quiet revolution in most fields of engineering. Software applications are widely used to control critical systems, which were traditionally controlled by humans. Software systems provide accurate and non-stopping services, and improve our productivity. However, they also introduce 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). There are two failure modes in these multi-component systems: *component level* and *system level* failures. The two modes are corresponding to the following two important issues related to dependability:

1. *Component reliability*, which is defined as the capability that a component satisfies its specified behavioral requirements. If a component is unreliable, then the system is also generally unreliable (without fault tolerance systems). This is currently the most recognized aspect in industry.

2. *System safety*, which is defined as the absence of accidents — events involving an unacceptable loss [2]. A recent challenge is the *system accident*, caused by increasing *coupling* among system components (software, control system, electromechanical and human), and their *interactive complexity* [3][4]. In contrast, accidents arising from component failures are termed *component failure accidents*.

Thus, *system safety* and *component reliability* are different elements of dependability. They are system property and component property, respectively [3]. People are now constructing intellectually unmanageable software systems that go beyond human cognitive limits. This allows potentially *unsafe interactions* to be undetected. System accidents often result from hazardous interactions among perfectly functioning components.

As an example, a system accident occurred in a batch chemical reactor in England [5]. The design of the system is shown in Fig. 1. 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 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.

Some other system accidents in avionics are also due to uncontrolled interactions between components [6]. 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 [6]. Ariane 5 adopted the

0730-3157/09 $25.00 © 2009 IEEE
DOI 10.1109/COMPSAC.2009.50

IEEE computer society

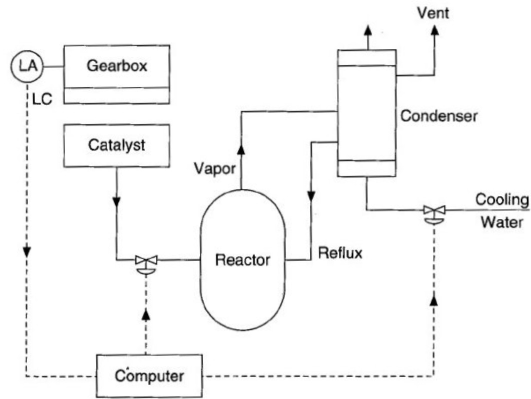

Figure 1. A Chemical Reactor Design

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 redefined and checked. Due to the overflow of input value computation, the IRS stopped working [7]. Then, the signaled error was interpreted as a launcher attitude, and led the control system to rotate the tailpipe at the end stop [8].

In these accidents, the components are reliable in terms of satisfying their specified requirements, but the systems are not safe as a whole. Thus, we may ask "why system safety issues are not addressed correctly?" We answer this question starting from an analysis on the process of defining system requirements.

As we know, system requirements include the following two classes [9] [10]:

1. *Core functional requirements*, which are requirements pertaining to all functions that are to be performed by the target system. Functional requirements specify the functions of components that compose a system.

2. *System safety requirements*, which are requirements about the safe operation of the target system. They focus on safety constraints specifying authorized system behaviors and components interactions, where a *safety constraint* specifies a specific safeguard [11]. A system may have different safety constraints under different contexts or critical levels. For example, avionic software systems are imposed more strict constraints than entertainment software applications.

In the practice of industry, these two types of requirements are defined by different groups, i.e., *system designers* and *safety engineers*, respectively. Thus, they are always separately defined at the stage of designing. As a result, two important causes of system accidents are raised.

Cause 1: Safety requirements are not completely considered and implemented during system design, although they are well identified and defined. The first reason for this might be that the designers do not understand the safety requirements clearly. The second reason might be that it is hard to implement the safety requirements, because the system complexity might be out of control.

Cause 2: Some safety requirements were unknown before the system is developed and used in real environment. We always need to learn new safety requirements from historical events [3]. However, it will be expensive to modify our designs after we learn these requirements, since the product has been released.

We are searching for a methodology which can eliminate these two causes. As Leveson mentioned, "most software related accidents have been system accidents" [1], thus people need to model and constrain interactions of system components to validate the absence of *dysfunctional interactions*. The STAMP (Systems-Theoretic Accident Model and Processes) [1] mentioned, these accidents result from inadequate *control* or enforcement of *safety-related constraints* of the systems. However, the roles of *control* and *safety-related constraints* have not been well understood yet in practice.

In this paper, we will consider and formalize safety requirements as control structures that restrict system behaviors at meta-model level. A system model A and its control structure $\hat{A}$ will be separately developed, as what we are doing in practice. Then, we can use the control structure in two modes to prevent system accidents, corresponding to the two causes of these accidents, as follows:

Mode 1 (model generating): the two models A and $\hat{A}$ are combined at design-time to deduce a new safe specification by using an automated tool. Then we implement the new specification. Since it is automated, Cause 1 concerning personal misunderstanding of designers and complexity of systems can be eliminated.

Mode 2 (model monitoring): the two models are separately implemented, but maybe at different stages of lifecycle. The system A is controlled at runtime by $\hat{A}$ which reports unsafe actions of the system. That means, we can incrementally add new safety constraints to the control structure $\hat{A}$ after we learn these constraints. Thus, Cause 2 can be eliminated, since we can improve system safety without modifying the system A.

In both of the two cases, safety requirements are *design-oriented*. That is, they are integrated into the design. This is different to the traditional approach – safety requirements are used to verify a system design – where safety requirements are more *testing-oriented*.

This paper is organized as follows. The framework of our methodology is presented in Section 2. To illustrate the idea, a preliminary introduction on interface automata appears in Section 3. *The interface C-System* which contains an interface automaton and a controlling automaton is introduced in Sections 4 and 5, where examples are used to illustrate how to formalize safety rules and combine them with the system specification. In Section 6, we discuss how to use interface C-Systems in practice. In Section 7, we compare our work

to classical model checking techniques. Section 8 concludes the paper.

2. The Framework of our Methodology

As shown in Fig. 2, a system A is composed of n components $\{A_i\}_{1\leq i\leq n}$. The system receives inputs $(\Sigma^I)^*$, and produces outputs $(\Sigma^O)^*$. Our framework is shown in Fig. 3. Safety requirements are implemented as a control structure $\hat{A}$, which can detect unsafe actions of A. The two are "combined" to deduce a new system C, which is globally a safe specification.

In particular. our methodology consists of the following steps:

1) Specifying the system model A (i.e., system behavior), including specifications of its components, internal and external interactions, e.g., using interface automata.
2) Specifying the control structure $\hat{A}$ (i.e., safety constraints), using a certain formal technique, e.g., controlling 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. Modes 1 and 2 may be applied to generate a new safe specification, or implement a globally safe system including a control structure, respectively.

At the first step, as we mentioned in [12], system behavior specifies an *operational semantics*, which defines what a system is able to do. System behavior modeling is achieved 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 actions of the system behavior.

At the second step, the control structure formalizing safety constraints 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, testing 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 constraints.

At the third step, in order to ensure system safety, we "combine" a system model with its control structure. Then we ensure that the system is globally safe with its control structure specifying safety requirements. Notice that the word "combine" may mean a "hard-composition" (i.e., Mode 1) or a "soft-composition" (i.e., Mode 2). The differences between the two in practice will be discussed in Section 6.

We remark here that another precondition is that we can find all safety constraints in a system. This is an issue of "risk identification" [13], which is beyond the scope of this paper. This work deals with "risk treatment" by reducing the accident risk [13].

To implement this framework, we must provide a formalism for modeling safety requirements. And we also need to carefully define the composition of a system model and its control structure. In the following sections, we will introduce such means based on interface automata and controlling automata.

3. Preliminary: Interface Automata

To model component-based concurrent systems with different input, output and internal actions, the theory of interface automata [14] extends I/O automata [15][16], which extends classical automata theory [17].

Unlike I/O automata, an interface automaton is not required to be input-enabled (i.e., some inputs may be recognized as illegal in some states) and only allows the composition of two automata (I/O automata allow the composition of infinite automata), and a synchronization of one output and one input action results a hidden action after the composition.

Definition 1: An **interface automaton** (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**.
- $S \subseteq Q$ is a set of **start states**, where $|S| \leq 1$. □

In the graph notation, a transition $p_k : (q, a, q') \in \delta$ 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.

The composition of two composable automata allows the automata to synchronize on shared actions, and asynchronously interleave all other actions.

Definition 2: Two interface automata A and B are **composable** if

- $\Sigma^H_A \cap \Sigma_B = \emptyset$
- $\Sigma^I_A \cap \Sigma^I_B = \emptyset$
- $\Sigma^O_A \cap \Sigma^O_B = \emptyset$
- $\Sigma^H_B \cap \Sigma_A = \emptyset$

We let $shared(A, B) = \Sigma_A \cap \Sigma_B$. □

Definition 3: If A and B are composable interface automata, their **product** $A \otimes B$ is the interface automaton defined by

- $Q_{A\otimes B} = Q_A \times Q_B$

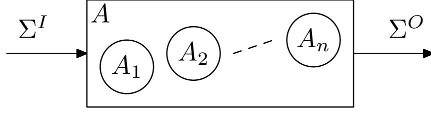

Figure 2. Traditional Framework

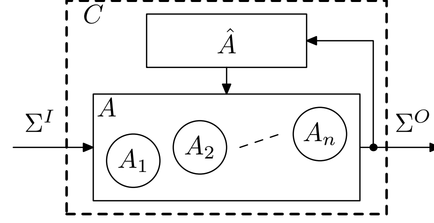

Figure 3. Our Framework

- $\Sigma^I_{A\otimes B} = (\Sigma^I_A \cup \Sigma^I_B) - shared(A,B)$
- $\Sigma^O_{A\otimes B} = (\Sigma^O_A \cup \Sigma^O_B) - shared(A,B)$
- $\Sigma^H_{A\otimes B} = \Sigma^H_A \cup \Sigma^H_B \cup shared(A,B)$
- $\delta_{A\otimes B} = \{\ p_i : ((v,u),a,(v',u)) \mid p_i : (v,a,v') \in \delta_A \wedge a \notin shared(A,B) \wedge u \in Q_B\}$
  $\cup \{p_j : ((v,u),a,(v,u')) \mid p_j : (u,a,u') \in \delta_B \wedge a \notin shared(A,B) \wedge v \in Q_A\}$
  $\cup \{p_{ij} : ((v,u),a,(v',u')) \mid p_i : (v,a,v') \in \delta_A \wedge p_j : (u,a,u') \in \delta_B \wedge a \in shared(A,B)\}$
- $S_{A\otimes B} = S_A \times S_B$. □

Note that the name of the transition p_{ij} of $A \otimes B$ may contain the names of two original transitions $p_i \in \delta_A$ and $p_j \in \delta_B$.

In the product $A \otimes B$, there may be **illegal states**, where one component is able to send an output $a \in shared(A,B)$ and the other is not able to receive a.

The composition of two interface automata A, B is obtained by restricting the product of the two automata to the set $Cmp(A,B)$ of **compatible states**, which are the states from which there exists a legal environment that can prevent entering illegal states.

Definition 4: If A and B are composable interface automata, their **composition** $A||B$ is the interface automaton defined by

- $Q_{A||B} = Cmp(A,B)$
- $\Sigma^I_{A||B} = \Sigma^I_{A\otimes B}$
- $\Sigma^O_{A||B} = \Sigma^O_{A\otimes B}$
- $\Sigma^H_{A||B} = \Sigma^H_{A\otimes B}$
- $\delta_{A||B} = \delta_{A\otimes B} \cap (Cmp(A,B) \times \Sigma_{A||B} \times Cmp(A,B))$
- $S_{A||B} = S_{A\otimes B} \cap Cmp(A,B)$. □

4. Interface C-Systems on Single Components

In this section, we start from a simple case – modeling safety constraints on a single component. Then we will progress to multi-component systems, which involve more complex behaviors.

In the example of the batch chemical reactor (c.f. Fig. 1), the computer system behavior is modeled using an interface automaton A of Fig. 4(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 the catalyst flow (p_1), then opening the water flow (p_2), etc., resulting in an infinite execution trace $p_1p_2p_1p_2$.... To respond to abnormal signals as soon as possible, the states q_0, q_1 both have a transition labeled l, which leads to a state that can sound an alarm (p_5) and stop the process (p_6). 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 *controlling automata* to formalize safety constraints. Then, we combine a controlling automaton with the system automaton.

Definition 5: A **controlling automaton** $\hat{A}$ over an interface 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**, such that $\hat{\Sigma} = \delta$. That is, $\hat{\Sigma}$ consists of all the names of transitions in δ of A.
- $\hat{\delta} \subseteq \hat{Q} \times \hat{\Sigma} \times \hat{Q}$ 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 5 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.

In the example, to prevent accidents, we need to impose 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 controlling automaton $\hat{A}$

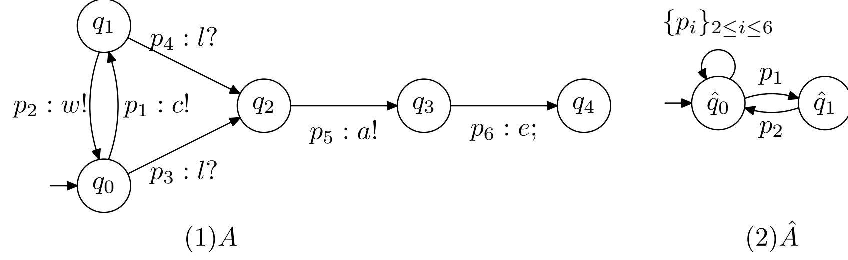

Figure 4. Automata of the Reactor Control System

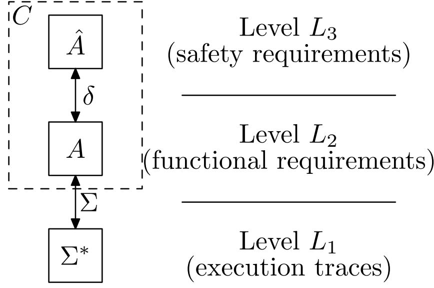

Figure 5. A 3-levels Overview

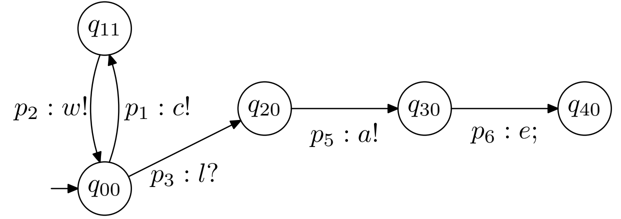

Figure 6. The Meta-Composition C

of Fig. 4(2). When we express 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 controlling automaton $\hat{A}$, and automatically generate a system model C satisfying the safety requirement.

Definition 6: The **meta-composition** C of an interface automaton $A = (Q, \Sigma, \delta, S)$ and a controlling automaton $\hat{A} = (\hat{Q}, \hat{\Sigma}, \hat{\delta}, \hat{S})$ over A is a tuple:

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

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) $(\hat{q}_j, p_k, \hat{q}_n) \in \hat{\delta}$.

We say that A and $\hat{A}$ constitute an **interface control system** (or simply **interface C-System**). □

The symbol $\vec{\cdot}$ is called *meta-composition operator*, and read "meta-compose". Its left and right operands are an automaton and a controlling automaton, respectively. Notice that an interface C-System is equivalent to the meta-composition C of an interface automaton and a controlling automaton.

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. 4, thus we get the automaton $C = A \vec{\cdot} \hat{A}$ of Fig. 6 where q_{ij} denotes $(q_i, \hat{q}_j)$.

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

Theorem 7: Given $A, \hat{A}$ and the meta-composition C, an **execution trace** $t_\Sigma \in \Sigma^*$ is recognized by C iff, t_Σ is recognized by A, and its **transition trace** $t_\delta \in \delta^*$ is recognized by $\hat{A}$. □

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

Thanks to $\hat{A}$, the hazardous execution traces, for example $cwclae$, which exists in A, will be eliminated, because its transition trace $p_1p_2p_1p_4p_5p_6 \notin L(\hat{A})$ (the language of $\hat{A}$). The comparison between A of Fig. 4(1) and C of Fig. 6 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 developed a formal and automated method for eliminating hazardous transitions.

5. Interface C-Systems on Multi-Components

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

To illustrate the principle, we use an example concerning a system composed of two components with interactions: a candy vending machine and a customer. We hope that, since this class of examples is so popular in the literatures of formal methods (e.g., Hoare's Communicating Sequential Processes (CSP) and I/O automata [15]), they will provide an interesting illustration of our idea. The candy machine A_m,

specified in Fig. 7(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. 7(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 $A_{mu} = A_m || A_u$ of Fig. 7(3), where q_{ij} denotes the composite state (m_i, u_j), $p_{i,j}$ denotes two synchronized transitions $\{p_i, p_j\}$. A transition of the composition may be composed of two transitions of components. For example, $p_{1,13} : s$ is a synchronization of $p_1 : s!$ and $p_{13} : s?$, which belong to A_m and A_u, respectively. Generally, a transition of $A = P||Q$ may be composed of one or two transitions of its components, where two transitions constitute a synchronization.

In the context of meta-composition, a composite transition is allowed if and only if both of its sub-transitions are allowed by its controlling automaton. Thus, we define the meta-composition operator as follows:

Definition 8: The **meta-composition** (or interface C-System) C of a composition $A = P||Q$ and a controlling automaton $\hat{A} = (\hat{Q}, \hat{\Sigma}, \hat{\delta}, \hat{S})$ over A is a tuple:

$$C = A \overrightarrow{\cdot} \hat{A} = (Q_A \times \hat{Q}, \Sigma_A, \delta', S_A \times \hat{S}) \qquad (2)$$

where $p_{\mathcal{I}} : ((v, u, q), a, (v', u', q')) \in \delta'$ ($p_{\mathcal{I}}$ contains a set of transitions $\{p_k\}_{k \in \mathcal{I}}$) iff,

(1) $p_{\mathcal{I}} : (((v, u), a, (v', u')) \in \delta_A$, and

(2) $\forall k : k \in \mathcal{I} \bullet (q, p_k, q') \in \hat{\delta}$. □

Notice that the specification of the example allows a hazardous situation: the greedy user repeatedly pushes the buttons without giving the machine a chance to dispense a candy bar (e.g., the transition labeled $p_{5,11} : b_1$ 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 buttons: "the transitions p_{11}, p_{12} are not allowed, when interactions occur between the machine and the user". Differing from the previous example, this type of constraints needs to synchronize the actions of the machine and of the user.

Formalizing the constraints, the semantics of the controlling automaton A_c of Fig. 8(1) is: whenever the user pushes a button (p_9, p_{10}), she or he cannot push it again (p_{11}, p_{12}), but can only wait for a candy bar.

Combining the whole system A_{mu} with its constraint A_c, we get the system $C = (A_m || A_u) \overrightarrow{\cdot} A_c$ in Fig. 8(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.

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.

6. Interface C-Systems in Practice

We proposed separately modeling core functional requirements and safety requirements, using interface automata and controlling automata, respectively. Since the industrial is familiar with the implementation of automata-based designs, we only need to make some comments on the role and the implementation of controlling automata in this section.

Mode 1 in practice. The first use of this mode is to deduce a new safe system specification. As we mentioned in Section 1, this automated composition can be applied to complex systems where Cause 1 exists.

Another use is safety checking. We combine A and $\hat{A}$, get a composite automaton C. If $L(C) = L(A)$, then we conclude that the system satisfies the safety constraints. If $L(C) \neq L(A)$, then the core functional requirements must be revisited to take the safety requirements into account.

Mode 2 in practice. The first use of this mode is safety incident reporting [11]. In particular, engineers construct a controller implementing $\hat{A}$. The controller may be a surveillance system, that is, another system traces the behaviors of the target system and reports errors if there is some sequences of events that are strings not accepted by $L(\hat{A})$. This use also guarantees a safe system $C = A \overrightarrow{\cdot} \hat{A}$ as a whole. This utilization is typically useful for adding safety controls to existing systems, since we cannot modify its inner structure after production or it is hard and expensive to do so. For example, if the chemical reactor system behaves as the pattern $p_1 p_2 p_1 p_4 ...$, then $\hat{A}$ will try the transition p_4 in the state $\hat{q}_1$, and then the surveillance system implementing $\hat{A}$ will report an error.

The second use is real-time service restoration, which means that system services are promptly restored after being lost due to an accident [11]. For example, if the chemical reactor system behaves as $p_1 p_2 p_1 p_4 ...$, then $\hat{A}$ will try the transition p_4 in the state $\hat{q}_1$. However, only p_2 is allowed in $\hat{q}_1$. Thus the surveillance system might execute p_2 (or remind human operators to do so) to avoid the accident. Since the state of $\hat{A}$ is switched to $\hat{q}_0$ after p_2, the succeeding sequence of actions $p_4 p_5 p_6$ becomes legal, and therefore $p_1 p_2 p_1 p_4 p_5 p_6$ is restored to be a safe execution.

The choice of using Mode 1 or Mode 2 depends on the project in hand. Mode 1 removes directly the traces which violate safety constraints from system specifications. Thus the resulting system will be efficient, but decreasing the generative power. Whereas Mode 2 is more flexible, we can modify constraints at a low cost.

Multiple safety constraints in practice. If we have n constraints formalized as the controlling automata $\hat{A}_1, \hat{A}_2, ..., \hat{A}_n$. Then we can combine these automata to an intersection $\hat{A}$ such that $L(\hat{A}) = \bigcap_{k=1}^{k=n} L(\hat{A}_k)$, according to classical automata theory [17]. The automaton $\hat{A}$ can be used as the controlling automaton that implicitly includes the semantics of n constraints.

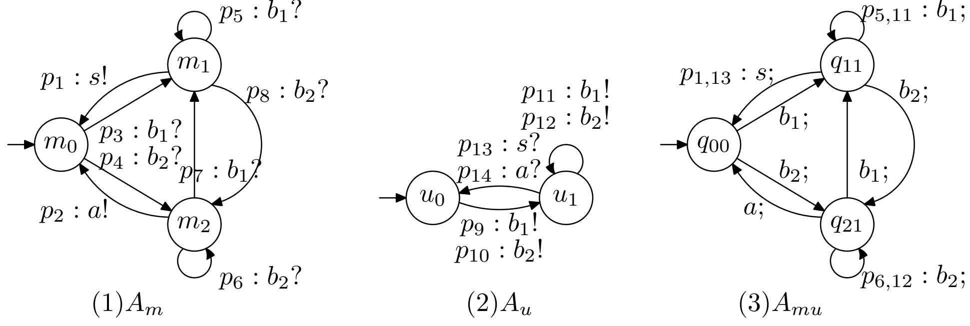

Figure 7. Automata of the Candy Machine System

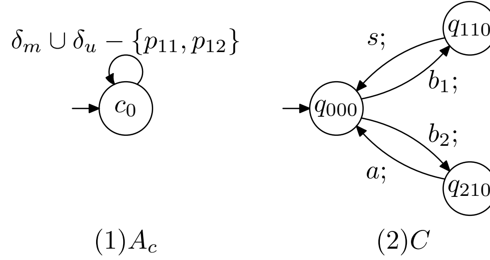

Figure 8. A Safety Constraint of the Candy Machine System

7. On the Relationship with Model Checking

The most popular technique for system safety verification is *model checking* [18]. Hundreds of checking patterns are collected for system engineers [19], and specific uses in safety engineering [10].

Traditionally, in order to validate the absence of system hazards, industrials identify system safety requirements, and use model checking to verify if system behaviors conform to safety requirements. In this framework, we have three steps in verifying a system. At first, we formalize system behavior as a model (e.g., a finite-state transition system, a Kripke model [20]). At the second step, we specify the safety constraints that we aim at validating using temporal logics [19]. At the third step, a certain checking algorithm is applied 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 design to ensure safety constraints.

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. 5). 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 C is definitely a safe execution. The two techniques are complementary.

Model checking and our approach use safety requirements as testing-oriented and design-oriented guidelines, respectively. Model checking may be used to reduce the design fault likelihood, and our approach can be applied to avoid behavior that are not in accordance with some critical safety requirements. That means, we can also use model checking techniques to verify interface C-Systems to increase our confidence in the composed design.

8. Conclusion

We proposed the formalism of interface C-Systems. In this framework, system behaviors and safety requirements can be separately modeled using interface automata and controlling automata respectively, since this is what we are actually doing in practice. As we illustrated using 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. Thanks to this framework, our methodology can be used for safety checking, incident reporting and service restoration.

The essential ideas of our approach are the separation and formalization of the system specification A (core functional requirements) and the control structure $\hat{A}$ (safety requirements). The automaton A handles inputs to produce

outputs using activities depending on the states, whereas the controlling automaton $\hat{A}$ treats activities to produce the set of acceptable activities depending on safety requirements.

This paper continues our work on *C-Systems* (short for "formal language control systems"). In [21], we actually defined the *input/output C-System*. The concept of controlling automata was proposed in [22]. The *context-free C-System* was proposed in [23] for restricting the use of modeling languages, in order to ensure guidelines and consistency rules of UML. This paper extends [22] in the sense that we identify more precisely the position and application modes of C-Systems in safety engineering, through a detailed analysis of safety requirements.

In the future, we will also study the formalization of parameterized constraints. Another direction is empirical case study on applying this formalism in large and complex systems.

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, *Safeware: System Safety and Computers*. Addison-Wesley, Reading, MA, 1995.

[3] N. Leveson, "Applying systems thinking to analyze and learn from events," in *Workshop NeTWorK 2008: Event Analysis and Learning From Events*, 2008, available from http://sunnyday.mit.edu/papers/network-08.doc.

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

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

[6] 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.

[7] 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)*. ACM, 2006, pp. 570–576.

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

[9] K. Allenby and T. Kelly, "Deriving safety requirements using scenarios," in *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE 2001)*. IEEE Computer Society, 2001, pp. 228–235.

[10] F. Bitsch, "Safety patterns - the key to formal specification of safety requirements," in *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2001)*, ser. Lecture Notes in Computer Science, U. Voges, Ed., vol. 2187. Springer, 2001, pp. 176–189.

[11] D. Firesmith, "Engineering safety requirements, safety constraints, and safety-critical requirements," *Journal of Object Technology*, vol. 3, no. 3, pp. 27–42, 2004.

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

[13] ISO/DIS, *ISO/DIS 31000: Risk Management – Principles and Guidelines*. International Standards Organization, 2009.

[14] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proceedings of the 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2001)*, 2001, pp. 109–120.

[15] 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.

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

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

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

[19] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, 1999, pp. 411–420.

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

[21] Z. Chen and G. Motet, "Modeling system safety requirements using input/output constraint meta-automata," in *Proceedings of the 4th International Conference on Systems (ICONS'09)*. IEEE Computer Society, 2009, pp. 228–233.

[22] Z. Chen and G. Motet, "Formalizing safety requirements using controlling automata," in *Proceedings of the Second International Conference on Dependability (DEPEND'09)*. IEEE Computer Society, 2009.

[23] Z. Chen and G. Motet, "A language-theoretic view on guidelines and consistency rules of UML," in *Proceedings of the Fifth European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009)*, ser. Lecture Notes in Computer Science. Springer, 2009.