On testing and automatic mending of safety PLC code

On testing and automatic mending of safety PLC code

Adnan Khan, Martin Fabian

Chalmers University of Technology, Department of Electrical Engineering, 41296 Göteborg, Sweden

Abstract

This paper presents an approach to automatically amend an erroneous model of an implementation using a safety specification as the basis to ensure safety. Industrially, safety PLCs are common to ensure safe operations. However, before its commissioning, the implemented safety code must be tested for faults caused by spurious transitions and missing safety transitions. Spurious transitions are implemented events that are not prescribed by the safety specification, while missing safety transitions are unimplemented safety events that are prescribed by the safety specification. The presence of these faults can result in material or human damage. The proposed approach requires the model of an implementation to be trace equivalent with the given safety specification only in terms of traces composed of safety events, which is captured by the notion of *safe-IOCOS*. If the implementation emits other than the specified safety events then the implementation is not safe-IOCOS and requires amendment. This is achieved by removing the spurious transitions and adding the missing safety events in the implementation using synthesis techniques from the supervisory control theory. The *infimal controllable superlanguage* is used to compute the *infimal safety extension*, which adds the missing safety transitions. It is shown how the resulting model of an implementation after amendment is both safe-IOCOS and controllable with respect to the specification.

Keywords:
Automata, Supervisory control theory, Input-output conformance testing, infimial controllable super-language, Safety, Discrete event system

1. Introduction

Computer controlled machines are increasingly being introduced for commercial purposes to improve efficiency and precision of manufacturing and production systems. This has resulted in factory environments where humans work in collaboration with robots, as well as products like self driving cars. These machines pose a significant threat to humans if proper safety measures are not implemented.

In production systems, most of the control is event-based and the nature of these events are typically discrete, which enables them to be modelled as *discrete event systems* [1]. In production systems, valve open, motor-off, robot-stop are some examples of discrete events, and these events are typically executed by *programmable logic controllers* (PLCs) synchronously.

Discrete event systems evolve with respect to occurring events asynchronously, while occupying a specific state at each time instant, where certain conditions are valid. Formally, the interaction of such systems can be described by different variants of synchronous composition [2].

The application of synchronous composition can be seen in the framework of *supervisory control theory* (SCT) [3]. In SCT, supervisors are *synthesized* automatically based on an uncontrolled model of the plant and a given specification such

Email addresses: adnan.khan@chalmers.se (Adnan Khan), fabian@chalmers.se (Martin Fabian)

that, the property of *controllability* must be satisfied.

Critical scenarios in automated systems occur as consequences of events, e.g. pressing the emergency stop button in a production setup, tripping of a production system due to high boiler pressure etc. As inputs, these *safety events* trigger actions typically implemented in a *safety PLC*, and as a consequence generate output safety events to guarantee safe system behavior. For instance, pressing the emergency stop should immediately activate the emergency stop sequence.

However, during non-critical scenarios, the nominal behavior, as represented by sequences of *nominal events*, should be active. This behavior is typically implemented in a standard PLC that does not have as strict requirements on its correctness as the safety PLC.

The partition between nominal events and safety events is not as clear-cut as might seem at first, though. Some events take both roles as they are necessary for the safety PLC to determine when a critical situation is about to arise. For example, if a boiler trips on high pressure, the pressure value under normal conditions acts as a nominal event. However, at the same time the pressure value is being monitored by the safety PLC to ensure that the boiler trips if it exceeds the set point. Such events we call *semi-nominal*, and they are typically shared between by standard PLC and safety PLC.

After the implementation of an actual physical system, testing is typically carried out. One of the approaches frequently used in the formal methods community is called *model-based testing* (MBT) [4]. The term model-based testing is interpreted and used in a variety of manner in the formal methods community for the generation of test cases. One of the interpretations deals with black-box testing that subjects an implementation to undergo various tests with respect to a given specification model. These test runs reveal the model of the implementation, which is then compared with the given specification to identify faults. A version of this approach is called the *input-output conformance simulation relation* (IOCOS) [5].

IOCOS is a finer relation compared to the *input-output conformance* (IOCO) [6], due to an extra requirement on the input events in addition to the requirement on the output events posed by IOCO. For the implementation to be IOCOS with respect to the specification, the implementation is required to have a subset of the specified outputs and a superset of the specified inputs.

However, [7] concluded that IOCOS is not detailed enough for testing safety as some behaviors can go untested due to the subset requirement on the implemented outputs and the superset requirement on the implemented inputs. This gave rise to a even stronger relation called *safe-IOCOS* [8].

The safe-IOCOS relation requires the implementation to emit exactly the same safety events (both inputs and outputs) prescribed by the specification. If the implementation emits less than the specified safety events or safety events that are not specified then the implementation is not safe-IOCOS and needs to be amended.

Compared to model-checking [9], where certain properties of a system are verified for correctness, testing does not guarantee absence of faults. And it is carried out to raise the confidence of the implementer on the implemented system. Furthermore, in testing we test an actual system, which is typically a black-box and builds the model based on the executed tests. However, in model-checking, typically we test the model of the system and not a physical system.

Testing an implementation to uncover faults related to safety is important. The implementation can be faulty either due to missing safety events or spurious events and to uncover them a suitable testing approach must be used.

Industrially, these implementations are typically standard PLCs and safety PLCs in a closed-loop setting with a physical production system. These industrial implementations, are now increasingly being tested using a simulation model [10, 11, 12] instead of a real physical system. However, in some cases black-box testing approaches are still used [13].

Furthermore, the process of amending the implementation after uncovering faults is an equally

important step that is typically carried out manually. This procedure of correction in itself is susceptible to mistakes made by engineers, which can have devastating consequences for faults related to safety properties.

1.1. Contribution

In this paper, it is shown how testing an implementation using safe-IOCOS is better for uncovering faults compared to IOCOS to ensure safety. Then, an approach to algorithmically amend a faulty implementation, using *the infimal safety extension* with respect to a specification is presented. The amendment is carried out to achieve trace equivalence for the traces associated with the safety events in both the implementation and the specification. To show the efficiency of the presented approach, an example modelled in the tool *Supremica* [14] is given. Furthermore, the relationship between controllability and safe-IOCOS is established.

1.2. Outline

This paper is structured as follows. In Section 2, the formal definitions required to describe the IOCOS and safe-IOCOS are detailed. Section 3 gives an overview of the IOCOS testing relation and its shortcomings. Section 4 introduces the safe-IOCOS relation. Section 5 gives detail regarding the supervisory control theory. Section 6 gives a brief overview on synthesis. In Section 7, some preconditions for the amendment of a faulty implementation is given. In Section 8, the solution based on the infimal safety extension is presented. In Section 9, a formal proof for the presented approach is detailed. Finally, Section 10 concludes the paper and presents some future work directions.

2. Preliminaries

In this section, some formalism and definitions typically used to represent *finite-state machine* (FSM), IOCOS, and safe-IOCOS are detailed.

Consider two disjoint sets of input actions I and output actions O, $I \cap O = \emptyset$. The output actions consists of nominal actions O_n, semi-nominal actions O_{sn}, and safety actions O_x, such that it holds, $O_{sn} \subseteq O_n$, $O = O_n \cup O_x$, and, $O_x \cap O_n = \emptyset$. These output actions are initiated by the system under test and are expressed with an exclamation mark, such as $!x \in O$. Similar to output actions, input actions also consist of nominal actions I_n, semi-nominal actions I_{sn}, and safety actions I_x such that it holds, $I_{sn} \subseteq I_n$, $I = I_n \cup I_x$, and, $I_x \cap I_n = \emptyset$. The input actions are signals to the system such as $a \in I$.

The main modeling formalism used in this paper is the *finite-state machine*.

Definition 1. *A (deterministic)* finite-state machine *(FSM) is a 5-tuple,* $\langle Q, \Sigma, i, \rightarrow, M \rangle$ *where*

- Q *is a finite non-empty set of states;*
- $\Sigma = I \cup O$ *is a finite set of events, these represent observable actions of the FSM;*
- $i \in Q$ *is the initial state;*
- $\rightarrow \subseteq Q \times \Sigma \times Q$ *is the transition function;*
- $M \subseteq Q$ *is the set of marked states.*

In addition to Σ, we defined the sets of nominal events $\Sigma_n = I_n \cup O_n$, semi-nominal events $\Sigma_{sn} = I_{sn} \cup O_{sn}$, and safety events $\Sigma_x = I_x \cup O_x$.

The transition function describes the possible evolution of the FSM from a source state $p \in Q$ to a target state $q \in Q$ associated with an event $a \in \Sigma$. We use infix notation $p \xrightarrow{a} q$ to denote that $\langle p, a, q \rangle \in \rightarrow$.

The transition function is generally a partial function, defined only for a subset of $Q \times \Sigma$. An event $a \in \Sigma$ is said to be *enabled* in a state $q \in Q$ if $p \xrightarrow{a} q$ is defined for some $q \in Q$. We write $p \xrightarrow{a}$ to denote this.

Furthermore, a *trace* t is a finite sequence of symbols of Σ, i.e. $t \in \Sigma^*$, including the empty trace ϵ. We can extend the transition function to traces in Σ^* as $p \xrightarrow{\epsilon} p \ \forall p \in Q$, and $p \xrightarrow{t\,a} r$ if $p \xrightarrow{t} q$ and $q \xrightarrow{a} r$ for some $q \in Q$ and $a \in \Sigma$.

Definition 2. *For an FSM* $S = \langle Q_S, \Sigma, i_S, \rightarrow_S, M_S \rangle$*, its set of traces are the ones defined from its initial state,*

$$\textbf{\textit{traces}}(S) = \{t \in \Sigma^* \mid i_S \xrightarrow{s}\} \tag{1}$$

Definition 3. *For an FSM* $S = \langle Q_S, \Sigma, i_S, \rightarrow_S, M_S \rangle$ *the state reached* after *a trace* t *is*

$$\textbf{\textit{after}}(S, t) = \{q \in Q_S \mid i_S \xrightarrow{t} q\}. \quad (2)$$

Definition 4. *For an FSM* $S = \langle Q_S, \Sigma, i_S, \rightarrow_S, M_S \rangle$*, its set of* marked traces *are the ones defined from its initial state reaching to the marked states.*

$$\textbf{\textit{traces}}_m(S) = \{t \in \Sigma^* \mid i_S \xrightarrow{t} q \in M_S\}. \quad (3)$$

Definition 5. *The set of all outgoing events enabled at a state* q *is*

$$\textbf{\textit{act}}(q) = \{\sigma \in \Sigma \mid q \xrightarrow{\sigma}\}. \quad (4)$$

The set of all outputs and inputs enabled at a state q is then given by $\mathbf{outs}(q) = \mathbf{act}(q) \cap O$ and $\mathbf{ins}(q) = \mathbf{act}(q) \cap I$, respectively.

3. Input-output conformance simulation relation

Testing is typically carried out to uncover faults in an implementation with respect to a specification model. During testing, a tester carries out several experiments on the implementation that is typically a black-box [15].

This means that the tester gives different inputs to the implementation with respect to the given specification and observes the outputs emitted by the implementation. Based on the observed behavior, the verdict of pass or fail is given.

Definition 6. *For a given FSM* G *and* K*, a relation* $R \subseteq (Q_G \times Q_K) \cup (Q_G \times Q_K)$ *is an iocos-relation if for any* $\langle p, q \rangle \in R$ *it holds that:*

1. $\textbf{\textit{ins}}(q) \subseteq \textbf{\textit{ins}}(p)$
2. $\forall a \in \textbf{\textit{ins}}(q)$ *and* $\forall p' : p \xrightarrow{a} p'$*, there* $\exists q' : q \xrightarrow{a} q'$ *such that* $\langle p', q' \rangle \in R$
3. $\forall !x \in \textbf{\textit{outs}}(p)$ *and* $\forall p' : p \xrightarrow{!x} p'$*, there* $\exists q' : q \xrightarrow{!x} q'$ *such that* $\langle p', q' \rangle \in R$

Definition 7. *IOCOS simulation relation* $= \cup \{R \subseteq (Q_G \times Q_K) \cup (Q_G \times Q_K) \mid R$ *is an iocos-relation*$\}$.

Definitions 6 and 7 from [5] define IOCOS over all pairs of states, even state-pairs that are in practice unreachable. However, in a practical setting, only the state-pairs that are actually reachable in the implementation and defined by the specification are of interest. Thus below we give a different definition of IOCOS, proven in [16] to be equivalent to the original definition.

Definition 8. *For an implementation* G *and a specification* K *with initial states* $p_0 \in Q_G$ *and* $q_0 \in Q_K$*, respectively, we say that* G *is IOCOS with respect to* K*, denoted* G *IOCOS* K*, if* $\langle p_0, q_0 \rangle \in IOCOS$ *[16].*

The above mentioned definitions for the simulation relation IOCOS from [5] are more general than necessary for this paper, as we here deal with deterministic FSM only. Therefore, an equivalent definition of G *IOCOS* K for deterministic FSM can be given.

Definition 9. *For deterministic implementation* G *and specification* K*,* G *is said to be IOCOS with respect to* K*, denoted* G *IOCOS* K*, if*

$$\forall t \in \textbf{\textit{traces}}(G) \cap \textbf{\textit{traces}}(K) : p_0 \xrightarrow{t} p, q_0 \xrightarrow{t} q : \textbf{\textit{outs}}(p) \subseteq \textbf{\textit{outs}}(q) \wedge \textbf{\textit{ins}}(q) \subseteq \textbf{\textit{ins}}(p)$$

If IOCOS is used to test safety, there is a possibility that safety events that are missing in the implementation but are prescribed by the specification can go untested [8]. These faults are called *missing safety events* that may typically occur in cases where a safety action is required to be implemented at several locations in a control program e.g. in a PLC code.

Another possible problem is if some *spurious events* (nominal or safety) that are not prescribed by the specification are present in the implementation. These events can also go untested if the IOCOS testing relation is used and can wreak havoc by leading to unsafe situations in practical settings.

To highlight the above mentioned problems of the IOCOS testing relation, an example illustrated in Fig. 1 with two implementations $G1$, and

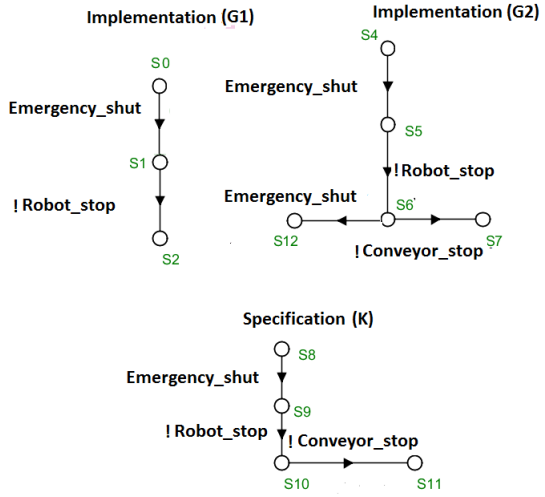

Figure 1: Implementation with missing *!Conveyor_stop* event ($G1$, top left). Implementation with spurious *Emergency_shut* event ($G2$, top right). The specification K is at the bottom.

$G2$ are tested individually with respect to a specification K. This example is based on a typical production cell that contains a robot and a conveyor belt. In terms of operations, both machines should be stopped when the emergency shutdown is activated per the specification K.

First, the implementation $G1$ is tested with respect to specification K. In $G1$, when the input event *Emergency_shut* is triggered, only the *!Robot_stop* output is activated. However, according to the specification K, after the occurrence of the input event *Emergency_shut*, both *!Robot_stop* and *!Conveyor_stop* should be activated. Thus, based on the IOCOS testing relation, the implementation $G1$ is IOCOS with respect to the specification K as the implementation emits a subset of the specified outputs. Thus, the missing output *!Conveyor_stop* remains covered in $G1$.

Now, the second implementation $G2$ is tested with respect to K. The input event *Emergency_shut* after getting triggered activates the *!Robot_stop* and *!Conveyor_stop* outputs. After assessing each state of $G2$, the IOCOS testing relation is valid with respect to K as each state emits a subset of the specified outputs and a superset of the inputs. However, if we analyze $G2$, the event *Emergency_shut* in state $S6$ is a spurious event as it is no specified by the specification K. This event is not uncovered by the IOCOS testing relation and can lead to safety critical situations.

4. Testing with safe-IOCOS

The problems with the IOCOS relation highlighted in Sec. 3 gives rise to a new relation called safe-IOCOS is proposed [8]. The safe-IOCOS simulation relation requires the implementation to be trace equivalent only for the traces composed of safety events. However, in terms nominal events, the implementation does not have any restrictions.

Definition 10. *For two deterministic FSMs G and K with equal set of events, G is said to be safe-IOCOS with respect to K if*

$$\forall t \in \boldsymbol{traces}(G||K) : \\ \boldsymbol{act}(\boldsymbol{after}(G,t)) \cap \Sigma_x = \boldsymbol{act}(\boldsymbol{after}(K,t)) \cap \Sigma_x \quad (5)$$

For an implementation G and specification K, the safe-IOCOS definition (Def. 10) is interpreted as the implementation G conforms to a specification K, if for all common traces between the specification and implementation, the safety inputs and the safety outputs possible from the state reached by the implementation after a trace are equal to the possible safety inputs and safety outputs events from the state reached by the specification after the same trace. If this equality relation between the respective sets of inputs and the outputs exist, the implementation is safe-IOCOS with respect to the specification for the executed trace.

Now, we apply safe-IOCOS on the example in Fig. 1 to test $G1$ and $G2$ with respect to K.

In $G1$, the application of safe-IOCOS with respect to K uncovers the missing safety output *!Conveyor_stop*. As according to Def. 10, safe-IOCOS require $G1$ to have equality for safety events and it can be seen in Fig. 1 that K has two safety outputs, *!Robot_stop* and *!Conveyor_stop* after the input event *Emergency_shut*. While, $G1$ has only *!Robot_stop*. This means, the implementation $G1$ is not safe-IOCOS with respect to K.

Similarly, in $G2$, when safe-IOCOS is applied, the spurious transition *Emergency_shut* in state

$S6$ in uncovered. Thus, $G2$ is not safe-IOCOS with respect to K due to the equality requirements for safety events expressed in Def. 10.

The example illustrated in Fig. 1 and Def. 10 is for a general case, where the specification K represents the given specification and contains safety events only. However, in practice the event partitioning in the specification differs from the stated example, which needs to be explained in relation to the implementation in physical setting.

In practice, the implementation G is a controlled plant that consists of a real physical system P and a controller S in a closed-loop setting. The job of the controller is to keep the plant within the global specification K that is a composition of nominal specification K_n and safety specification K_x. The event partitioning of both K_n and K_x that follows is inspired by the splitting method of [17].

The *nominal* specification K_n deals with the nominal behavior and contains only the nominal events Σ_n. The implementation is allowed to execute various combinations of these nominal events Σ_n.

Some of the nominal events in K_n are also required by the safety specification to ensure safety. These events are called semi-nominal events such that $\Sigma_{sn} = \Sigma_n \cap \Sigma_{K_x}$. The role of the semi-nominal events is to bind the nominal specification K_n with the safety specification K_x to ensure safety.

For the safety specification K_x, its event set is $\Sigma_{K_x} = \Sigma_x \cup \Sigma_{sn}$. Thus, the safety specification shares the semi-nominal events Σ_{sn} with the nominal specification, K_n.

The safety events, Σ_x only belong to the safety specification i.e. $\Sigma_x \cap \Sigma_n = \emptyset$ and requires careful attention, because if the specified safety events are missing in the implementation then it can either lead to machine damage or human accidents.

Practically, the controller S is typically manually constructed, but could also be *synthesized* by means of some formal approach, such as the Supervisory Control Theory (SCT) [3]. This same theoretical approach can, as will be shown, also aid in amending an implementation when faults have been uncovered by safe-IOCOS testing.

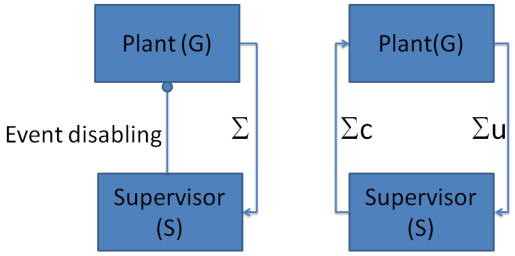

Figure 2: Asymmetric (left) and symmetric (right) supervisor feedback loops.

5. Supervisory Control Theory

The SCT takes a control-theoretic approach on discrete event systems. The main idea behind this theory is the automatic synthesis of a controller, called a *supervisor* S, for an uncontrolled system, the *plant* P, so that the controlled system G adheres to a given *specification*.

The plant models all the possible behavior of the uncontrolled system, while the specification models the desired controlled behavior. The task of synthesis is now to automatically calculate a supervisor, given the plant and the specification, such that when this supervisor controls the plant the closed-loop system exhibits a behavior that is guaranteed to remain within the specification, while always being able to complete some desired task.

The supervisor effects its control on the plant by dynamically disabling events from occurring. Thus, there is an asymmetric feedback-loop between the plant and the supervisor, see Fig 2 (left), the plant generates events, while the supervisor at each global state disables a subset of the events to guarantee that the closed-loop system never goes outside the specified behavior. However, not all events are subject to disablement by the supervisor; *controllable* events can be disabled, while *uncontrollable* events cannot. This must be considered when synthesizing the supervisor; the synthesis procedure must produce a supervisor that is *controllable* with respect to the plant and the uncontrollable events.

In addition, some global states are of significant interest, and are therefore *marked* by the

specification. At least one of these marked states must always be reachable from any state in the closed-loop system,. This poses a requirement on the supervisor to be *non-blocking*.

Furthermore, it is common to require that the supervisor should disable events only when enabling such events would inevitably lead to violating the controllability or non-blocking properties. This is captured by the requirement of the supervisor to be *minimally restrictive*[1]. It is known that a minimally restrictive, controllable and non-blocking supervisor always exists, is computable in the regular case, and is unique [3, 1].

Though [3] originally took a formal language-theoretic viewpoint, for calculation purposes it is beneficial to model the plant, the specification, and the supervisor as finite-state machines [1].

In the following it is assumed that plant, specification, and supervisor all have the same event set, Σ, and that the controllable and uncontrollable events, Σ_c and Σ_u, respectively, partition Σ.

The interaction between the supervisor and the plant can be modeled by *synchronous composition*, where an event can occur only if it is simultaneously enabled in both.

Definition 11. *For two FSMs A and B with the same set of events, their* synchronous composition *is $A||B = \langle Q_A \times Q_B, \Sigma, \langle i_A, i_B\rangle, \rightarrow_{A||B}, M_A \times M_B\rangle$, where*

$$\langle p, q\rangle \xrightarrow{\sigma}_{A||B} = \langle p \xrightarrow{\sigma}, q \xrightarrow{\sigma}\rangle \tag{6}$$

if both $p \xrightarrow{\sigma}$ and $q \xrightarrow{\sigma}$ are defined, else undefined.

From Def. 11, it follows that:

$$\forall t \in \mathbf{traces}(A||B) = \mathbf{traces}(A) \cap \mathbf{traces}(B).$$

Note that events that in either FSM label no transitions will be globally disabled by the composition; that is, they are disabled from ever occurring.

Now controllability can be formally defined.

[1]This paper does not treat minimally restrictiveness, so this property is not formally defined.

Definition 12. *Given a plant P and uncontrollable events Σ_u, a supervisor S is* controllable *w.r.t. to P and Σ_u if*

$$\boldsymbol{traces}(P||S)\Sigma_u \cap \boldsymbol{traces}(P) \subseteq \boldsymbol{traces}(P||S) \tag{7}$$

Def. 12 says that the states reached after the execution of a common trace in P and S, the uncontrollable events enabled by the plant P must be a subset of the uncontrollable events enabled by the supervisor. Put another way, S can never disable an uncontrollable event enabled by P in a state reached by a common trace.

Definition 13. *Given a plant P and a supervisor S, the supervisor is non-blocking if*

$$\overline{\boldsymbol{traces}_m(P||S)} = \boldsymbol{traces}(P||S). \tag{8}$$

Def. 13 says that the generated controller S is non-blocking if the closed-loop system of the given plant P and the supervisor S is such that it can always reach a marked state.

The authors in [18] re-interpret the original supervisory control theory formulation into an *input/output interpretation*, where Σ_u are regarded as outputs from the plant and inputs to the supervisor, while Σ_c are outputs from the supervisor and inputs to the plant. This forms a symmetric relation between the plant and the supervisor, see Fig. 2 (right), which according to [18] is better suitable to model real systems, where events do not occur spontaneously but only as responses to commands. This changes the role of a supervisor from being a passive safety device that stops bad things from happening (while allowing good things), to being an active entity commanding actions of the plant.

It is shown by [18] that the input/output interpretation does not really change anything when it comes to synthesis and controllability, the same requirements are still valid and it is only a matter of interpretation. In this interpretation, controllability means that the supervisor must at each global state be ready to accept as inputs the plant outputs enabled in that state. Similarly, there is an *inverse controllability* where the supervisor

may not generate outputs (inputs to the plant) that the plant cannot accept in its current state. However, due to the way synthesis is performed, inverse controllability is trivially satisfied, and need not concern us further.

6. Synthesis to amend a faulty implementation

In the SCT, if the plant does not conform to the specification then synthesis is carried out to ensure the property of controllability and nonblocking. Synthesis is an algorithmic process to generate supervisors. The supervisor's role is to keep the plant safe by dynamically restricting the events generated by the plant. This event restriction is modeled by synchronous composition, where an event is enabled if and only if both the plant and the supervisor simultaneously enable it.

Since the plant is considered immutable, synthesis works by either constricting or expanding the synchronous composition of the plant and specification models. In the former case this is done by computing the *supremal controllable sublanguage*, $\mathbf{sup\,C}(G||K)$, and in the latter case by computing the *infimal controllable superlanguage*, $\mathbf{inf\,C}(G||K)$.

For computing $\mathbf{sup\,C}(G||K)$, transitions leading to unsafe states by uncontrollable events are iteratively removed until a fix-point is reached. It is known [3] that a minimally restrictive solution to this always exists and is computable. However, the result may be the degenerate *null* supervisor that restricts the plant so much as to not allowing it to do anything. This indicates that practically no useful safe solution exists.

In the case of computing $\mathbf{inf\,C}(G||K)$, the model of the plant and specification operating in synchrony is instead expanded within the scope of the plant. This allows more behavior while in practice breaking the specification, though in a minimal way.

Both computation approaches i.e. supremal controllable sublanguage and the infimial controllable superlanguage basically alters the specification and not plant. However, for the proposed case, it is the implementation that is a controlled plant and requires amendment. Therefore, the specification should be kept intact and the synthesis should compute the new modified implementation model. To achieve this, the roles of the implementation and the specification needs to be reversed so that the algorithm modifies the implementation instead of the specification.

Now the question is, which computation approach should be used to amend the implementation model. The answer probably lies in the very nature of the approaches discussed above. The supremal controllable sublanguage computation can help in removing spurious transitions as it basically constricts the specification. However, as mentioned above, if no safe solution exist then the resultant language will be null, which is not of any use in practical settings.

Furthermore, for cases where a state has spurious events along with valid events, computing the supremal controllable sublanguage is not suitable. This so, since the state associated with the spurious transition gets removed if the property of controllability is compromised, which leads to the elimination of valid transitions along with spurious ones.

This enables the use of infimial approach as it can help in the case of missing transitions because it expands the specification. And the last step of the infimial approach i.e. FSC [2] can help in the removal of spurious events. However, this requires to be validated.

7. Preconditions for amendment

In a typical manufacturing system, there are several production units working in parallel to carry out the programmed tasks. Usually, each machine in a production unit is involved in carrying out multiple operations [19]. For example a robot is involved in pick and place operations, a conveyor has start and stop operations etc. Now, these operations can be implemented individually or be a part of a larger segment in an implemented PLC code along with other operations.

To amend specific segments of an implementation for faults related to safety, each segment of the implementation must be amended with re-

spect to the safety specification related to that particular segment. This means that the safety specification must contain the information related to the safety behavior that should be implemented in a particular faulty segment, along with the information of the associated nominal behavior.

For example, if a segment in an implementation requires amendment for the safety behavior related to a proximity sensor, then each nominal operation associated with the proximity sensor logic is required to be prescribed in the safety specification.

The nominal behaviors that are not associated with the faulty segment of the implementation should be preserved. This means that the implementation is allowed to nominally do more compared to the prescribed nominal behavior in the safety specification.

Now to amend an implementation G with respect to a safety specification K_x, where both G and K_x are non-blocking, such that $\Sigma_{K_x} \subseteq \Sigma_G$ by computing $\mathbf{inf\,C}(G||K_x)$ the following assumptions will be made.

The first assumption is that the semi-nominal events in K_x are renamed if they are used in segments of the given implementation G that do not require amendment for safety. For example if an event $!x_1$ is used in the segment of G that is under scrutiny, then it is renamed to $!x_2$ in other segments of the implementation G.

Renaming of common semi-nominal events allows to amend automatically just the faulty segments of the implementation G. Thus, the rest of the implementation will remain in its original form.

The second assumption is that the safety specification K_x has a subset of semi-nominal events with respect to the implementation G. This means that the semi-nominal events enabled by K_x cannot be missing in G.

Definition 14. *Given an implementation G and a safety specification K_x, the semi-nominal events of K_x cannot be missing from G, that is:*

$$\forall t \in \boldsymbol{traces}(G||K_x) :$$
$$\boldsymbol{act}(\boldsymbol{after}(K_x, t)) \cap \Sigma_{sn} \subseteq \boldsymbol{act}(\boldsymbol{after}(G, t)) \cap \Sigma_{sn}$$

If the given G does not fulfill Def. 14, then the computation of $\mathbf{inf\,C}(G||K_x)$ may result in blocking. Due to this, missing safety events may not get added to the faulty state in G.

With the above assumptions, now the goal is to algorithmically amend a faulty implementation G that does not fulfill Def. 10 with respect to the safety specification K_x. However, to amend the faulty G with respect to K_x, first K'_x is created by adding the $\Sigma_G \setminus \Sigma_{K_x}$ events to K_x. This addition of events makes $\Sigma_{K'_x} = \Sigma_G$.

Now, the amendment should be carried out in such a manner that trace equivalence for the traces associated with safety behaviors, i.e. Σ_x, between the final amended model G'' and the safety specification K'_x is achieved. This partial trace equivalence for traces associated with Σ_x, makes G'' safe-IOCOS with respect to K'_x. In addition, G'' will be non-blocking, and controllable with respect to the safety specification K'_x.

8. Infimal safety extension to amend a faulty implementation

The general procedure for infimal controllable superlanguage [1] for any two FSMs expands the language of one FSM with respect to the other. This procedure is typically used to expand an arbitrary specification M with respect to an uncontrolled plant P. It consists of three steps.

In the first step of the procedure, a new state, called DEAD is introduced in the specification model M. After the addition of DEAD, all the undefined uncontrollable events of the active event set from each state are added to it.

In the second step, self-loops on all uncontrollable events are added to the dead state. This addition augments the original M to now have the traces $\mathbf{traces}(M)\,\Sigma^*_{uc}$.

Finally, in the third step, intersection is carried out between the traces of the newly generated specification, i.e. $\mathbf{traces}(M)\,\Sigma^*_{uc}$, with the traces of the uncontrolled plant $\mathbf{traces}(P)$, i.e. $\mathbf{traces}(M)\,\Sigma^*_{uc} \cap \mathbf{traces}(P)$. This intersection of traces is the infimial controllable superlanguage of P and M, $\mathbf{inf\,C}(P||M)$.

Definition 15. *Given a plant P, a specification M, and uncontrollable events Σ_{uc}, the infimal controllable superlanguage* ***inf C***$(P||M)$ *is given as:*

$$\boldsymbol{inf\,C}(P||M) = \boldsymbol{traces}(M)\,\Sigma_{uc}^{*} \cap \ \boldsymbol{traces}(P).$$

Though the description of the $\mathbf{inf\,C}(P||M)$ computation is based on languages, the efficient way to do it is to work on FSM. Thus, in practice an FSM M' is generated such that $\mathbf{traces}(M') = \mathbf{inf\,C}(P||M)$.

From Def. 15 we see that the traces of M' are the traces of M extended by uncontrollable events that exist within the plant P. Thus, M' is larger than M in terms of behavior, but controllably contained within P.

Now, to amend a faulty implementation G with additional nominal behavior i.e. $\Sigma_{K_x} \subseteq \Sigma_G$, the computation of infimal controllable superlanguage can be computed with respect to the safety specification K_x such that $\Sigma_x \subseteq \Sigma_{uc}$. However, this time, it is the traces of the implementation G that will be modified with respect to K_x. Thus, we will compute the *infimal safety extension* of G with respect to K_x, $\mathbf{inf\,SE}(G||K_x)$ as:

$$\mathbf{inf\,SE}(G||K_x) = \mathbf{traces}(G)\,\Sigma_x^{*} \cap \mathbf{traces}(K_x)$$

An important thing to remember while computing $\mathbf{inf\,SE}(G||K_x)$ is to preserve the segments of the given implementation that do not require amendment to ensure safety. Thus, the infimal computation must be restricted only to the faulty segments of the implementation denoted G_{faulty}. This modified procedure can, as will be shown, help in amending the faulty segments, while keeping the non-faulty segments of the given implementation in its original form.

8.1. Computation of the infimal safety extension

To selectively amend a given implementation using the infimal controllable superlanguage such that the non-faulty segments of the implementation are not disturbed, the computation needs to be restricted to a particular set of states denoted $Q_{aug} \subseteq Q_G$. As these states belong to the faulty segment G_{faulty} and require amendment. The procedure for computing infimal safety extension is as follows.

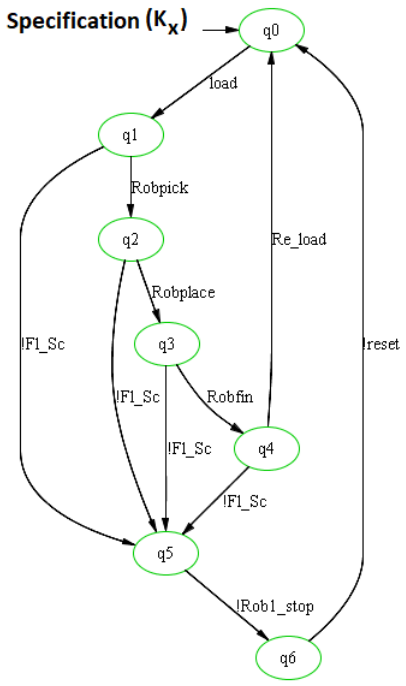

Figure 3: Safety specification K_x.

1. In the first step, the states i.e. $Q_{aug} \subseteq Q_G$ that requires augmentation are partitioned after creating K'_x by setting $\Sigma_{K_x} = \Sigma_G$ and computing $G||K'_x$. Setting $\Sigma_{K_x} = \Sigma_G$ restricts the events that are outside K_x from participating in the computation of $G||K'_x$. Thus, states associated with the $\Sigma_G \setminus \Sigma_{K_x}$ events do not show up in the final result.
2. In the second step, G_{aug} is created by *augmenting* the states of Q_{aug}. This is done by adding a DEAD state in G with Σ_x self-looped. Then, from each $p_i \in Q_{aug}$ transitions on the undefined safety events i.e. Σ_x are added from p_i to the DEAD state on the events $\sigma \in \Sigma_x$ state.
3. The third step computes a new model of the implementation $G' = G_{aug}||K_x$.
4. In the fourth step, a model G'' is generated from G' after *merging* each state labelled $(DEAD, q_j) \in Q_{G'}$ with any state $(p_i, q_j) \in Q_{G'}$.

The process of merging basically re-directs the in-going and out-going transitions of $\langle \mathrm{DEAD}, q_j \rangle$

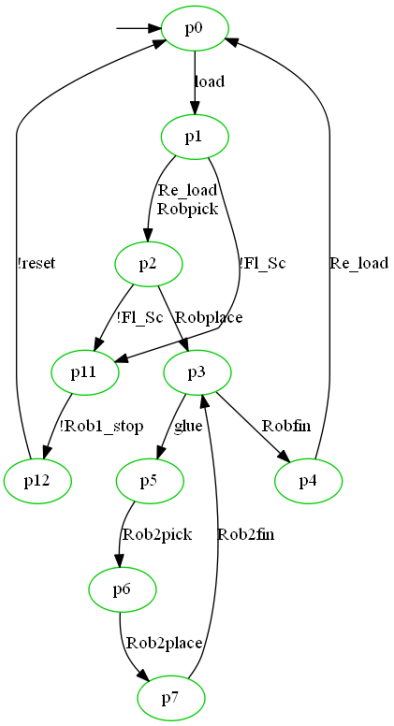

Figure 4: Implementation G

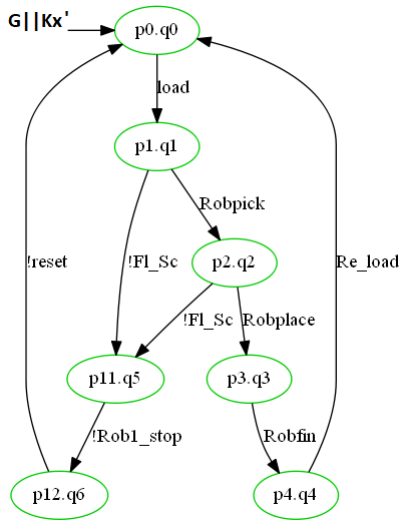

Figure 5: Faulty segment G_{faulty} of the implementation.

to $\langle p_i, q_j \rangle$, which makes G'' non-blocking. Now, this G'' fulfils Def. 10, and is controllable with respect to K'_x.

8.2. Computational complexity

The overall computation complexity of the proposed procedure is dominated by $O(n^2)$ and the breakdown of each step is as follows. In the first step synchronous composition is computed between two machines. Thus the complexity of step 1 is $O(n^2)$, where n is the number of states in the system. Step 2 has an overall complexity of $O(n)$. In step 3 we again compute synchronous composition of two machines hence the complexity is $O(n^2)$. For step 4, the worst case complexity is $O(n^2)$ i.e. if all states are required to be merged and for each state, all the other $n-1$ states have to be searched.

8.3. Example

Let us illustrate the infimal safety extension procedure by applying it to the implementation G given in Fig. 4. This implementation is not safe-IOCOS with respect to K_x (see Fig. 3) as the safety event *!Fl_Sc* prescribed by K_x in states q_3 and q_4 are missing. In addition there is a spurious event *Re_load* in state p_1. To distinguish the implementation from specification, implementation states are labelled as p_i, and specification states are labelled as q_i.

1. The first step is to partition the states $Q_{aug} \subseteq Q_G$ by computing computing $G||K'_x$. This is done by creating K'_x after adding the events, *glue, Rob2pick, Rob2place, Rob2fin* to K_x. These events also includes the renamed semi-nominal events in G. Then $G||K'_x$ is computed to identify all $p_i \in Q_{aug}$, which can be seen in Fig. 5. The identified states that belong to Q_{aug} are p_0, p_1, p_2, p_3, p_4, p_{11}, and p_{12}.
2. In the second step, G_{aug} is computed by augmenting the states of Q_{aug}. This is done by adding a DEAD state in G with Σ_x self-looped. Then, from each identified state i.e. p_0, p_1, p_2, p_3, p_4, p_{11}, and p_{12}, the undefined safety events Σ_x are added to the DEAD state. The result of the computation can be seen in Fig. 6.
3. The third step computes the intermediate FSM $G' = G_{aug}||K_x$, given in Fig. 7.
4. In the fourth step, a model G'' is generated from G' after merging the states $\langle \text{DEAD}, q_0 \rangle$, $\langle \text{DEAD}, q_5 \rangle$ and $\langle \text{DEAD}, q_6 \rangle$ to $\langle p_0, q_0 \rangle$, $\langle p_{11}, q_5 \rangle$,

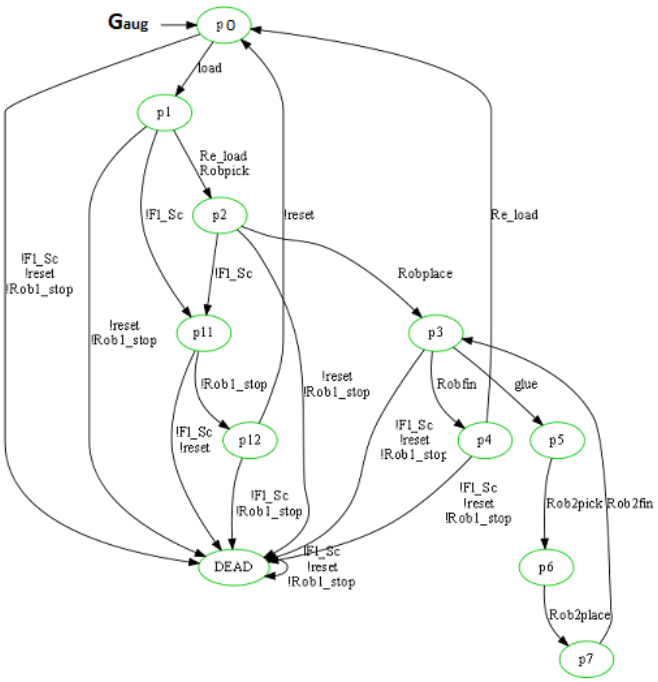

Figure 6: The augmented FSM G_{aug} of Step 2.

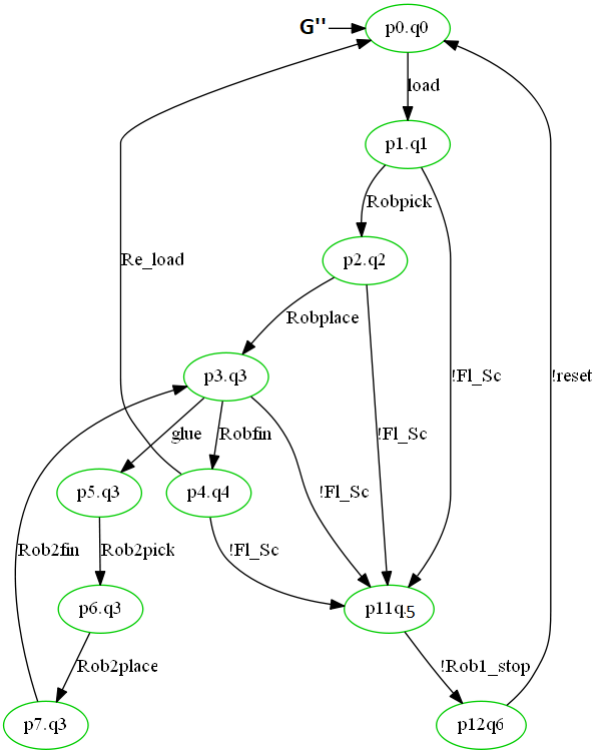

Figure 8: Final result G'' of Step 4, after merging states

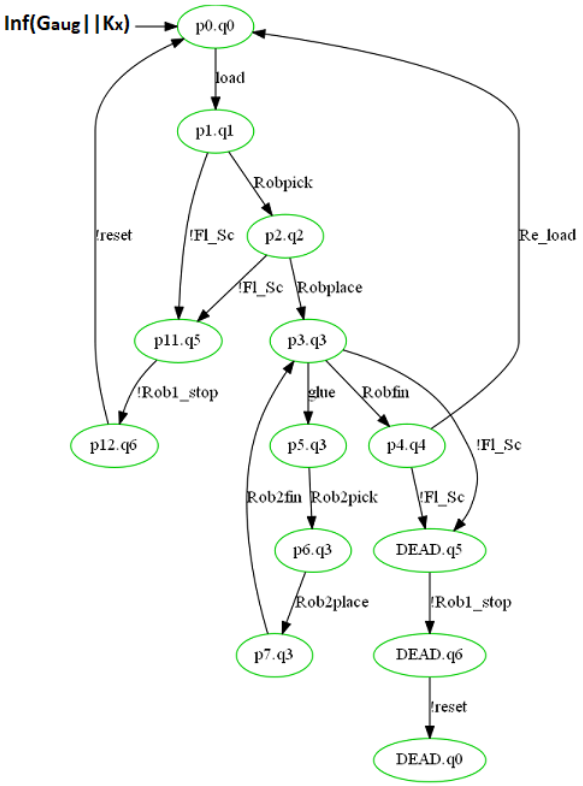

Figure 7: Intermediate FSM G' of Step 3

and $\langle p_{12}, q_6\rangle$ respectively in G'. The final result after merging is given in Fig. 8.

The process of merging is required to remove the blocking caused by the state $\langle DEAD, q_0\rangle$ in G'. This means that $\overline{\mathbf{traces}_m(G')} \neq \mathbf{traces}(G')$. To solve this issue, a states labeled $\langle DEAD, q_j\rangle$ is merged with any state $\langle p_i, q_j\rangle$.

This is done by assigning semantics to the state-labels in a similar way to *Kripke-structures* [20]. The second element of the tuple $\langle$DEAD, $q_j\rangle$ is matched to the second element of states $\langle p_i, q_j\rangle$, and then these states are merged by re-directing the incoming and outgoing transitions of $\langle$DEAD, $q_j\rangle$ to $\langle p_i, q_j\rangle$. And this is done for all states $\langle$DEAD, $q_j\rangle$ and any $\langle p_i, q_j\rangle$.

For example, the state $\langle$DEAD, $q_0\rangle$ at the bottom of Fig. 7 is merged with the state (p_0, q_0). Similarly, states $\langle$DEAD, $q_5\rangle$ and $\langle$DEAD, $q_6\rangle$ are merged with states $\langle p_{11}, q_5\rangle$ and (p_{12}, q_6), respectively. The final implementation model G'' after merging all related states is given in Fig. 8.

Now, the generated G'' in Fig. 8 is safe-IOCOS, and controllable with respect to K'_x, because the

missing safety events *!Fl_Sc* is added to the states p_3 and p_4. In addition, the spurious event *!Re_load* in state p_1 is removed. This means that $\forall t \in \mathbf{traces}(G''||K'_x) : \mathbf{act}(\mathbf{after}(G'', t)) \cap \Sigma_{K'_x} = \mathbf{act}(\mathbf{after}(K'_x, t)) \cap \Sigma_{K'_x}$.

In the next section it is shown that the approach will always amend a given implementation G with respect to a safety specification K_x such that the computed model G'' is safe-IOCOS, non-blocking, and controllable with respect to K'_x.

9. Correctness

For G'' computed by the above described infimal safety extension procedure to be correct by construction, it is required that the procedure adds the missing safety events and removes spurious events, in such a way that G'' is controllable with respect to the safety specification, non-blocking, and safe-IOCOS. The following theorem proves that G'' is correct by construction.

Theorem 1. *Given an implementation G with event set Σ_G, and a safety specification K_x with event set $\Sigma_{K_x} = \Sigma_x \cup \Sigma_{sn} \subseteq \Sigma_G$, where the safety events $\Sigma_x \subseteq \Sigma_{uc}$ and the semi-nominal events $\Sigma_{sn} \subseteq \Sigma_n$, with Σ_n the nominal events, and $\Sigma_x \cap \Sigma_n = \emptyset$. Computation of the* infimal safety extension *G'' of G with respect to K_x will make G'' controllable wrt K'_x, non-blocking, and $\forall t \in \boldsymbol{traces}(G''||K'_x), \boldsymbol{act}(\boldsymbol{after}(G'', t)) \cap \Sigma_{K'_x} = \boldsymbol{act}(\boldsymbol{after}(K'_x, t)) \cap \Sigma_{K'_x}$.*

Proof. We show this by showing that $\mathbf{traces}(G''||K'_x) = \mathbf{traces}(K'_x)$. Since $\Sigma_{G''} = \Sigma_{K'_x} (= \Sigma_G)$, we have that $\mathbf{traces}(G''||K'_x) = \mathbf{traces}(G'') \cap \mathbf{traces}(K'_x)$.

Take $t \in \mathbf{traces}(G''||K'_x) = \mathbf{traces}(G'') \cap \mathbf{traces}(K'_x)$. Thus, $t \in \mathbf{traces}(K'_x)$.

Take $t \in \mathbf{traces}(K'_x) = \mathbf{traces}(K_x)$. If $t \notin \mathbf{traces}(G'') \cap \mathbf{traces}(K'_x)$, then $t \notin \mathbf{traces}(G'')$, which means that $t \notin \mathbf{traces}(G')$, which means that $t \notin \mathbf{traces}(G_{aug})$. Let $t = s\sigma$, such that $s \in \mathbf{traces}(G'')$. Obviously, $s \in \mathbf{traces}(K'_x)$. Now, σ is either a semi-nominal event, or an uncontrollable safety event. If $\sigma \in \Sigma_{sn}$ then by the assumption of no missing semi-nominal events Def. 14, since $s\sigma \in \mathbf{traces}(K_x)$ also $s\sigma \in \mathbf{traces}(G)$, and hence $s\sigma \in \mathbf{traces}(G'')$. If $\sigma \in \Sigma_x$, then $s\sigma \in \mathbf{traces}(G_{aug})$ and hence $s\sigma \in \mathbf{traces}(G'')$, since $s\sigma \in \mathbf{traces}(K_x)$. Thus, $t \in \mathbf{traces}(G''||K'_x)$.

Now, since $\mathbf{traces}(G''||K'_x) = \mathbf{traces}(K'_x)$ the claim is shown; $\mathbf{traces}(K'_x)$ is controllable wrt itself, K'_x is by assumption non-blocking, and after each trace of $t \in \mathbf{traces}(G''||K'_x)$ the continuation $t\sigma$ is in both $\mathbf{traces}(G'')$ and $\mathbf{traces}(K'_x)$. □

In Def. 10 it is assumed that the two FSM have the same sets of events. This is why K'_x is used in Theorem 1, instead of K_x. What happens in the composition $G''||K'_x$ is that the segment that should not be touched is disregarded. In the example this is the "appendix" that begins with the *glue* event. This is nominal behavior that should go untreated. K'_x has these events in its alphabet, but has no transitions on them, and so these events are disabled in every state.

Based on the proof, for any given implementation model G and safety specification K_x, the new modified implementation model G'' computed via the proposed infimal safety extension procedure will be correct by construction related to spurious events and missing safety events.

In addition, the implementation model G'' is controllable with respect to the safety specification K_x. This means that in a physical setting the amendment prescribed by G'' can be applied by changing the existing controller S (and not the physical plant P); since $G = P||S$, a new controller S'' can be generated from G'' so that $P||S'' = G''$, precisely because G'' is controllable.

Furthermore, the computed G'' is free of spurious semi-nominal events, that is:

$$\forall t \in \mathbf{traces}(G''||K'_x) : \quad (9)$$
$$\mathbf{act}(\mathbf{after}(G'', t)) \cap \Sigma_{sn} = \mathbf{act}(\mathbf{after}(K'_x, t)) \cap \Sigma_{sn}$$

Expression (9) means that if there is a semi-nominal event prescribed by K'_x in a state then there should exist the same semi-nominal event in the corresponding state of G''. This is required to ensure safety, because if there is a spurious semi-nominal event present in G'', which is not prescribed by K'_x, then that spurious semi-nominal event can lead to an unsafe state. Since by the

proof of Theorem 1 $\mathbf{traces}(G''||K'_x) = \mathbf{traces}(K'_x)$, G'' obviously fulfills (9).

10. Conclusion

In this paper a new approach to automatically amend an implementation with respect to a safety specification is presented. The proposed procedure computes the *infimal safety extension*, which selectively amends the given implementation to ensure safety. The amendment is carried out in such a manner that the other segment of the implementation, which does not require any amendment is preserved in its original form. The infimal safety extension represents a new model of an implementation G'', eliminating faults associated with missing safety events and spurious events. The computed model G'' is safe-IOCOS, controllable with respect to the safety specification, and non-blocking. Furthermore, the proposed approach is illustrated with an example to show the infimal safety extension working dynamics, and is substantiated with a formal proof.

In practical settings, the presented approach can be used to amend the faulty segment of a safety PLC code, because physically, it is the controller that is amended, while the physical plant remains untouched.

In terms of future work, the problem of scalability needs to be handled. In this context, a *compositional* approach would seem to be possible, where both the implementation and the safety specification are given by several components that interact (through synchronous composition) to make up the global system, and advantage is taken of the fact that these are distinct interacting components, similar to [21].

Furthermore, the presented approach deals only with discrete event systems. However, when it comes to continuous safety behavior and timed safety events, the presented approach has limitations, which is planned to be captured in future work.

Acknowledgements

The authors thank Dr. Sahar Mohajerani for insightful discussions. This work was supported by the Swedish Research Council, VR, under the SyTeC strong research environment, grant 2016-06204, the Swedish Innovation Agency, VINNOVA, under the TESTRON project, grant 2015-04893, and party supported by the Wallenberg AI, Autonomous Systems and Software program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] C. Cassandras, S. Lafortune, Introduction to Discrete Event Systems, SpringerLink Engineering, Springer US, 2009 (2009). doi:10.1007/978-0-387-68612-7.

[2] A. Hellgren, M. Fabian, B. Lennartson, Prioritised synchronous composition of inhibitor arc petri nets, in: Discrete Event Systems, Springer, 2000, pp. 459–466 (2000).

[3] P. J. Ramadge, W. M. Wonham, Supervisory control of a class of discrete event processes, SIAM Journal on Control and Optimization 25 (1) (1987) 206–230 (1987). doi:10.1137/0325013.

[4] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007 (2007). doi:10.1016/B978-0-12-372501-1.X5000-5.

[5] C. Gregorio-Rodríguez, L. Llana, R. Martínez-Torres, Input-output conformance simulation (iocos) for model based testing, in: Formal Techniques for Distributed Systems, Springer, 2013, pp. 114–129 (2013). doi:10.1007/978-3-642-38592-69.

[6] G. Tretmans, Test generation with inputs, outputs and repetitive quiescence, 1996, https://research.utwente.nl/en/publications/test-generation-with-inputs-outputs-and-repetitive-quiescence-2 46 (1996).

[7] A. Khan, D. Thönnessen, M. Fabian, On-the-fly conformance testing of safety PLC code using QuickCheck, in: 2019 IEEE 17th International Conference on Industrial Informatics (INDIN'19), IEEE, 2019 (2019).

[8] A. Khan, M. Fabian, On the Safe IOCOS relation for testing safety PLC code, in: 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2019, pp. 1449–1452 (2019).

[9] G. Frey, L. Litz, Formal methods in PLC programming, in: Systems, Man, and Cybernetics, 2000 IEEE International Conference on, Vol. 4, IEEE, 2000, pp. 2431–2436 (2000). doi:10.1109/ICSMC.2000.884356.

[10] C. G. Lee, S. C. Park, Survey on the virtual commissioning of manufacturing systems, Journal of Computational Design and Engineering 1 (3) (2014) 213–222 (2014).

[11] Z. Liu, C. Diedrich, N. Suchold, Virtual Commissioning of Automated Systems, INTECH Open Access Publisher, 2012 (2012).

[12] A. Khan, P. Falkman, M. Fabian, Testing and validation of safety logic in the virtual environment, CIRP Journal of Manufacturing Science and Technology 26 (2019) 1–9 (2019).

[13] B. Fernández, E. Blanco, A. Merezhin, Testing & verification of PLC code for process control, in: Proc. of the 14th Int. Conf. on Accelerator & Large Experimental Physics Control Systems, 2013, pp. 1258–1261 (2013).

[14] R. Malik, K. Åkesson, H. Flordal, M. Fabian, Supremica-–an efficient tool for large-scale discrete event systems, in: IFAC World Congress, Tolouse, France, 2017, pp. – (2017).

[15] M. Krichen, S. Tripakis, Black-box conformance testing for real-time systems, in: International SPIN Workshop on Model Checking of Software, Springer, 2004, pp. 109–126 (2004).

[16] A. Khan, S. Mohajerani, M. Fabian, On test case reduction for testing safety properties ofmanufacturing systems, CIRP Journal of Manufacturing Systems (Submitted).

[17] F. Reijnen, T. Erens, J. van de Mortel-Fronczak, J. Rooda, Supervisory control synthesis for safety PLCs, in: 15th Workshop on Discrete Event Systems, WODES, 2020, pp. – (2020).

[18] S. Balemi, Control of discrete event systems: theory and application, Ph.D. thesis, Swiss Federal Institute of Technology, Zürich, Switzerland (1992).

[19] K. Bengtsson, Flexible design of operation behavior using modeling and visualization, Chalmers University of Technology,, 2012 (2012).

[20] H. Flordal, R. Malik, M. Fabian, K. Åkesson, Compositional synthesis of maximally permissive supervisors using supervision equivalence, Discrete Event Dynamic Systems 17 (4) (2007) 475–504 (2007).

[21] S. Mohajerani, R. Malik, M. Fabian, Compositional synthesis of supervisors in the form of state machines and state maps, Automatica 76 (2017) 277–281 (Feb. 2017). doi:10.1016/j.automatica.2016.10.012.