Department of Computing and Software

Faculty of Engineering — McMaster University

Automatic Disablement of $tick$ Event in Sampled-Data Supervisory Control

by

Hina Mahmood and Ryan J. Leduc

CAS Report Series **CAS-23-07-RL**

Department of Computing and Software **July 2023**
Information Technology Building
McMaster University
1280 Main Street West Hamilton, Ontario, Canada L8S 4K1

Automatic Disablement of *tick* Event in Sampled-Data Supervisory Control

Hina Mahmood and Ryan J. Leduc
Department of Computing and Software, Faculty of Engineering,
McMaster University, Hamilton, Ontario, Canada

Technical Report CAS-23-07-RL
Department of Computing and Software
McMaster University

July 31, 2023

Abstract

Sampled-data (SD) supervisory control theory [42, 43, 29] focuses on implementing timed discrete-event system (TDES) supervisors as SD controllers, and addresses the implementation and concurrency issues involved. The SD approach requires that as soon as a prohibitable event is eligible in the closed-loop system, it must be forced by explicitly disabling the *tick* event.

As multiple modular supervisors are usually in control of the same prohibitable event, designers typically do not find it easy and straightforward to satisfy this property at every state of the closed-loop system while designing their TDES supervisors by hand. This is believed to be one of the primary reasons for limited adoption of the SD supervisory control methodology.

In this report, we present an approach to automate the *tick* disablement/event forcing mechanism in the SD supervisory control framework. Specifically, we introduce a new synchronization operator, called the *SD synchronous product*. Our SD synchronous product operator is capable of automatically disabling a *tick* event in the closed-loop system, if both *tick* and a prohibitable event is possible in the plant model and enabled by all supervisors.

Based on this synchronization operator, we formulate and describe our SD synchronous product setting. We redefine the existing TDES and SD definitions to utilize our synchronization operator and setting. We establish and prove equivalence between our SD synchronous product setting and the existing SD supervisory control setting.

Then, we present controllability and nonblocking verification results for our SD synchronous product setting. Specifically, we formally prove that if a theoretical TDES designed in our setting is controllable, nonblocking and satisfies our specified properties, then the implemented system also exhibits correct behaviour and abides by the control laws. After that, we give our tweaked predicate-based algorithms for

verifying various TDES and SD properties in our setting. We have implemented these algorithms in a DES research tool, DESpot [14].

Finally, we present the example of a Flexible Manufacturing System to show that our approach simplifies the design logic of modular supervisors and reduces their state size. This results in improving the ease of manually designing SD controllable supervisors and reducing the verification time of the overall TDES system. This should increase the adoption of the SD supervisory control methodology in particular, and formal methods in general, in the industry by simplifying the formal design and verification process of control systems.

Keywords: discrete-event systems, timed discrete-event systems, sampled-data supervisory control, *tick* disablement, event forcing

Contents

1 Introduction

Discrete Event Systems (DES) are dynamic systems that encompass processes that are discrete in time and state space, often asynchronous, and typically non-deterministic. These systems evolve by changing state in accordance with the instantaneous occurrence of physical *events*. DES are quite common in industry and include a variety of man-made systems namely manufacturing systems, computer and communication networks, transport and logistic systems, and traffic control systems. These applications generally require some degree of control and coordination to ensure the orderly flow of events according to the given specifications and/or to prevent the occurrence of undesired chains of events.

To solve the control problem of DES and extend the control theory concepts of continuous systems to DES, [46, 34] introduced *Supervisory Control Theory (SCT)*. This theoretical approach is based on automata theory and formal language models [22]. SCT provides algorithms and methods for the analysis and control of DES.

In SCT, a *supervisor* is introduced to alter the unrestricted behaviour of the *plant* DES using feedback control as per the desired specifications. SCT partitions the set of events into *controllable* and *uncontrollable*, the former being amenable to disablement by a supervisor. In SCT, a system is desired to have two properties, *controllability* (undesired actions do not occur) and *nonblocking* (no deadlock or livelock).

A *timed DES (TDES)* model adds timing information to an untimed DES in order to deal with temporal specifications. The TDES modelling framework, proposed in [5, 7], extends the untimed DES by introducing a new *tick* event. The *tick* event represents the passage of one time unit and corresponds to the tick of a global clock to which the system is assumed to be synchronized.

In TDES, non-*tick* controllable events are referred to as *prohibitable events*. It also introduces a new class of non-*tick* events called *forcible events* that can preempt the occurrence of a *tick* event. In order to force an event, the standard method used in TDES framework is to "explicitly disable" the *tick* event while designing TDES supervisors by hand.

The ultimate goal of designing a theoretical TDES supervisor is to generate its corresponding controller implementation. To the best of our knowledge, the first generic formal implementation approach applicable to a wide variety of TDES is the *sampled-data (SD) supervisory control* theory [42, 43, 29], although several adhoc approaches (discussed in Section 1.1) existed before. We use the SD methodology as the basis of our work.

The goal of the work presented in this report is to reduce the design complexity and size of TDES supervisors that are manually designed in the SD supervisory control framework, and ease the process of modelling and verification of these SD controllable supervisors. These goals are achieved by bridging the gap between theoretical TDES supervisors and physical controllers by making them behave in a similar way with respect to forcing of events. Our strategy to do this is to adopt the controller's way of event forcing and apply it to the theoretical TDES setting of SD supervisory control.

We propose an approach to automate the *tick* disablement/event forcing mechanism in TDES by devising a new synchronization operator, called the *SD synchronous product*. The SD synchronous product operator, represented as "$||_{SD}$", is intended to be used to combine the plant and supervisor models to construct closed-loop system in the SD framework. Our synchronization operator is smart enough to automatically disable the *tick* event, if a forcible event is eligible in the plant and enabled by all supervisors. As a result, software designers no longer need to explicitly incorporate this logic while designing their SD controllable TDES

supervisors by hand. This should increase the adoption of SD supervisory control theory in particular, and formal methods in general, in the industry by simplifying the formal design and verification process of control systems.

We formally verify our approach with respect to the desired properties of controllability and nonblocking. In order to do this, we first establish equivalence between the existing SD supervisory control setting and our proposed SD synchronous product setting. We then utilize this equivalence to reprove all existing results of the SD supervisory control setting for our SD synchronous product setting. Specifically, we show that if a theoretical TDES system designed in our SD synchronous product setting is controllable, nonblocking and abide by the specified control laws, then the physically implemented system retains these properties, and the generated SD controller behaves as expected with respect to control action, event forcing and nonblocking.

In this report, we first introduce our SD synchronous product operator, discuss its relevant properties, and describe the proposed SD synchronous product setting. Then, we adapt the existing TDES and SD properties to make them compatible with the proposed synchronization operator and setting. After that, we establish and formally prove equivalence between our SD synchronous product setting and the existing SD supervisory control setting. This is followed by formally proving the controllability, nonblocking and desired SD supervisory control results for the proposed setting. Finally, we demonstrate the design complexity and supervisor state size reduction as well as the ease of modelling and verifying SD controllable TDES supervisors in our SD synchronous product setting with the help of an example.

In the remainder of this section, we report several adhoc approaches for implementing DES and TDES supervisors, followed by a discussion of why we have chosen the SD supervisory control methodology as the basis of our work. We then explain the research problem that we have addressed in this report, along with our proposed solution by presenting a small independent chunk from our complete illustrative example. We close this section by giving an outline of the rest of the sections that are present in this report.

1.1 Why SD Supervisory Control?

Although theoretical aspects of SCT have received substantial attention in academia, the implementation of DES supervisors is still an open issue [40, 48]. This is because of the discrepancy between the abstract SCT supervisors and resulting control realization [15]. Due to the clear interpretation gap between the roles a supervisor is assumed to play within the SCT modelling framework and the roles a controller has to play in the real-world control systems, the implementation of DES supervisors is not a straightforward task [47].

In order to facilitate the implementation of asynchronous, event-driven, theoretical supervisor on a synchronous, signal-based programmable logic controller (PLC) [4], several untimed approaches and algorithms have been developed. These include [1, 27, 25, 13, 18, 11, 41, 20, 26, 24, 16, 40, 33, 23, 35].

Unlike untimed DES, only a few studies have focused on the supervisor's implementation for time-sensitive systems due to the added complexity of incorporating time in the DES modelling and control. Some significant timed implementation approaches presented in the literature include [6, 39, 17, 38, 37].

A detailed analysis of the existing untimed and timed approaches reveals that these implementation solutions are application-specific, and cannot be generalized and applied to other DES. Most of them are limited to PLC implementation, which further restricts the applicabil-

ity of the presented methodologies. Moreover, these approaches do not guarantee a controllable and nonblocking implementation, even if designers make sure that theoretical models satisfy these properties. Also, they address implementation issues on an adhoc basis while paying little or no attention to concurrency issues.

To the best of our knowledge, SD supervisory control methodology, presented in [42, 43, 29], is the first generic formal implementation approach that addresses the implementation, concurrency and timing issues in a well-defined way. The SD approach proposes to implement TDES supervisors as *SD controllers*, a good example of which is a Moore synchronous Finite State Machine (FSM) [8].

This approach provides sufficient conditions to guarantee that if a theoretical TDES is controllable, nonblocking, and satisfies these properties, then the physical system will also exhibit correct behaviour, i.e. the implementation will be controllable, nonblocking and abide by the specified control laws. It ensures this by: 1) identifying a set of existing TDES properties, and introducing the new property of *SD controllability* (Definition 3.7) that deals with concurrency and timing issues, 2) establishing a formal representation of TDES supervisors as SD controllers, and 3) providing a formal translation method to convert TDES supervisors into Moore FSM, which can either be implemented on a PLC, in hardware using digital logic, or as a computer program.

Due to these distinctive characteristics of the SD supervisory control theory and its applicability to a variety of TDES applications, we are using this approach as the foundation of our work.

1.2 Motivating Problem

In the TDES framework, the standard method used by software designers to force an event at a given state of the system is to "explicitly disable" the *tick* event at the corresponding state of the TDES supervisor. This has the effect of removing the now impossible behaviour that *tick* could occur before the forcible event, as the forced event is guaranteed to occur before the *tick*.

In the SD supervisory control theory, all prohibitable events are treated as forcible events (Section 3.3). The SD controllability property (Definition 3.7) requires that a prohibitable event be forced in the same clock period (before *tick*) in which it is enabled, and must remain disabled otherwise. Specifically, the forward implication ($\Rightarrow$) of Point ii enforces this check to make sure that at a given state, if *tick* and a prohibitable event is possible in the plant model, and prohibitable event is enabled by the TDES supervisor, then the supervisor must explicitly disable *tick* in order to force the prohibitable event.

Failure to satisfy this property correctly can have serious consequences. For instance, consider the following two scenarios:

1. At a given state, both *tick* and a prohibitable event is possible in the plant and enabled by all modular TDES supervisors. This is highly undesirable as the SD controller would not know whether to let *tick* occur or force a prohibitable event at this state. This would make the translation of TDES supervisors into SD controllers ambiguous.
2. A prohibitable event is enabled by a modular supervisor and supervisor has also disabled the *tick* event which was possible in the plant model. However, if one of the other modular supervisors has disabled this prohibitable event or the event is not currently possible in the plant, our system will become uncontrollable (Definition 2.22).

In order to avoid these unwanted outcomes, it is important that all plant components and modular supervisors must coordinate and agree on the enablement/disablement of *tick* and prohibitable events. For this purpose, the designer has to manually keep track of two things: 1) when is the *tick* event possible in the plant, and 2) when any prohibitable event is possible in the plant and is not disabled by any of the modular supervisors. In this case, the designer must explicitly disable *tick* event in the supervisor model to force the prohibitable event.

Clearly, keeping track of this information manually, and at the same time guarantee that Point ii ($\Rightarrow$) of SD controllability property is always satisfied at every state of the closed-loop system is not a trivial and trouble-free task. This is further complicated by the fact that multiple modular supervisors are usually in control of the same prohibitable event. This is believed to be one of the primary reasons for limited adoption of SD supervisory control methodology, as typically designers do not find it convenient to manually satisfy Point ii ($\Rightarrow$) of SD controllability property, especially while developing modular solutions.

Moreover, this logic of disabling *tick* in the presence of an enabled prohibitable event also needs to be explicitly specified in the design of various TDES supervisors so that modular supervisors have sufficient knowledge of the plant's behaviour as well as each other's behaviour, in order to work together appropriately. In order to make the TDES models aware of each other's behaviour with respect to the *tick* event and common prohibitable events, designers usually rely on two methods: 1) duplicate the logic of one TDES model in the other model(s), or 2) add *expansion events*, i.e. non-physical/virtual events that are added solely to aid in communication between modular supervisors. Five such events, prefixed by "no", are discussed in Section 10.1.

Besides making the TDES modelling process demanding and laborious for designers, these methods also increase the design complexity and size of TDES supervisors, hence the overall SD system. We briefly present an example for the first technique of duplicating logic and its associated complexities below. Please see Section 10.3 for a comprehensive discussion on both techniques.

1.2.1 Illustrative Example

In Section 10, we will discuss the TDES example of a Flexible Manufacturing System (FMS) from [42, 43] to apply our approach and discuss our results. Here, we briefly present one part of this example to concretely illustrate the afore-mentioned issues, and then demonstrate how our proposed approach addresses them (Section 1.3.1).

The FMS, shown in Figure 1, consists of two conveyor belts (Con2 and Con3)[1], four machines (Robot, Lathe, PM and AM), and five buffers (B2, B4, B6, B7 and B8)[1]. Each buffer has the capacity to hold a single part and it is desired that buffers never overflow (try to put a part in the buffer when it already has one) or underflow (try to remove a part from the buffer when it is empty). Please note that the behaviour of buffers is treated as specifications, and need to be implemented as TDES supervisors. In Figure 1, event names preceded by '!' represent uncontrollable events, and those without '!' are prohibitable events.

In the FMS, once a new part enters the system via Con2 (*pt_ent_sys*), it goes to buffer B2 (*pt_ent_B2*). The Robot is responsible for taking parts from buffer B2 (*R_from_B2*) and pass them on to buffer B4 (*R_to_B4*) for further processing by Lathe. After processing, the part returns to buffer B4, from which Robot moves it either to buffer B6 (*R_to_B6*) or B7 (*R_to_B7*). Please see Section 10.1 for a more in-depth explanation of this system.

[1]This example is taken from a larger example, which is why the part labels are not contiguous.

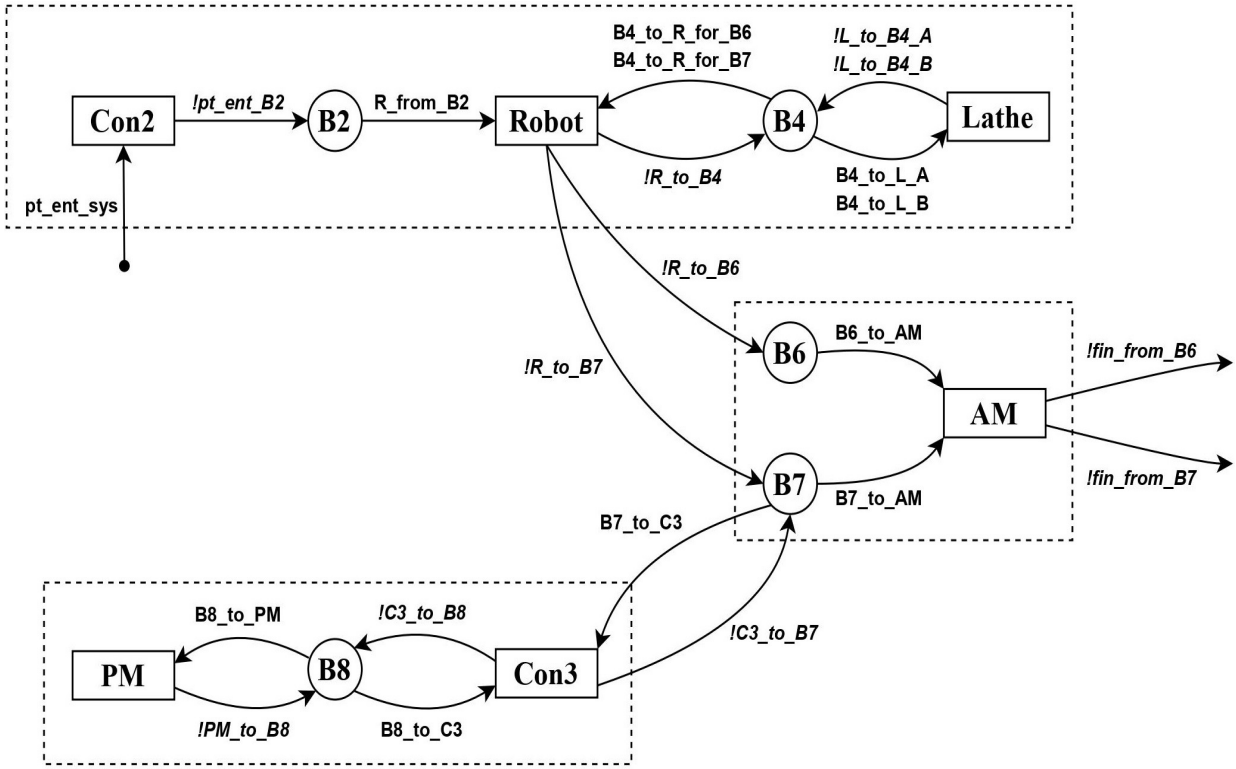


Figure 1: An Overview of Flexible Manufacturing System

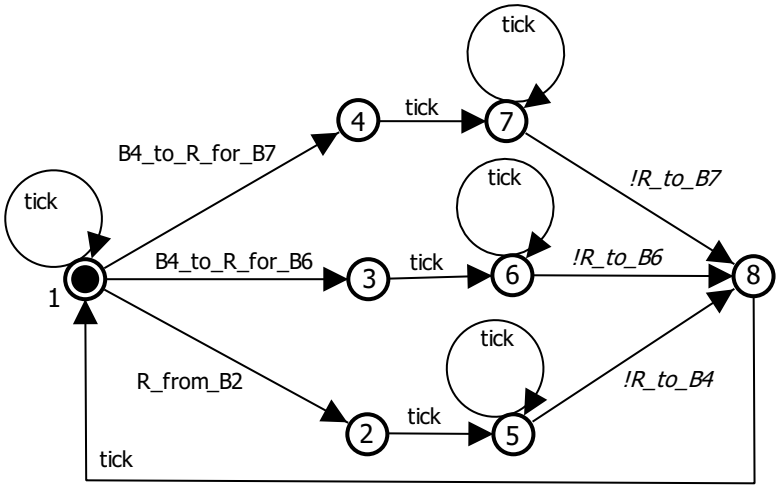


Figure 2: TDES Plant **Robot**

In order to manage and force the prohibitable event *R_from_B2* as per the given specifications, TDES plant model **Robot** (Figure 2), and modular TDES supervisors, ***B2*** (Figure 3) and ***TakeB2*** (Figure 4), are designed in the existing SD supervisory control setting [43]. Please note that in the complete FMS example (Section 10), *R_from_B2* is under the control of four modular supervisors. To keep our example simple, here we are discussing only two of them that are tightly coupled to one another. To understand the graphical representation and notation of a TDES automaton, please see Section 2.3.

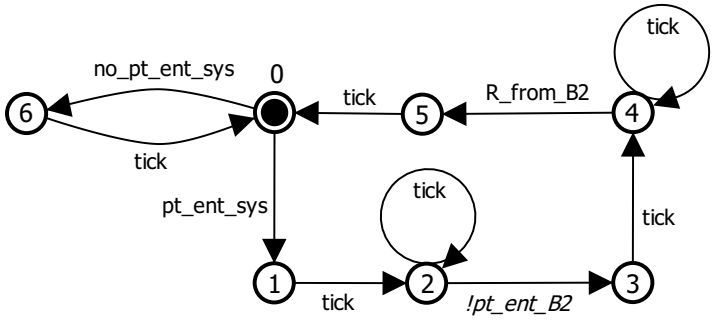


Figure 3: TDES Supervisor ***B2***

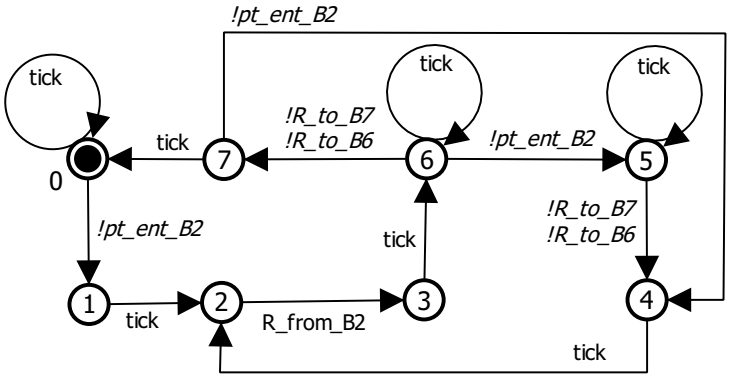


Figure 4: TDES Supervisor ***TakeB2***

Supervisor ***TakeB2*** is primarily responsible for forcing *R_from_B2*. In order to keep the system timed controllable (Definition 2.22), it is important to make sure that ***TakeB2*** does not disable *tick* and try to force *R_from_B2* when it is not possible in **Robot** or disabled by supervisor ***B2***. Therefore, ***TakeB2*** must take into account the behaviour of these models before it attempts to force *R_from_B2*.

By looking at supervisor ***B2***, we note that it enables *R_from_B2* after the part has reached buffer B2 (*pt_ent_B2*). This means that ***TakeB2*** needs to keep track of the part's progress. Once the part has reached buffer B2, only then ***TakeB2*** should disable *tick* and force *R_from_B2*, as this is the time when the prohibitable event will be enabled by supervisor ***B2*** and possible in **Robot**.

Now that we have manually figured out the "right time" for forcing *R_from_B2*, this logic needs to be incorporated in the design of supervisor ***TakeB2***. This is done by adding the event *pt_ent_B2* to its event set, and duplicating the related behaviour of ***B2*** in ***TakeB2***, i.e. by replicating the event sequence "*pt_ent_B2* − *tick*" from supervisor ***B2*** to ***TakeB2***. Only after knowing about the occurrence of this sequence of events, ***TakeB2*** forces *R_from_B2* by disabling *tick* at state 2, thus making sure that system does not become uncontrollable while it is trying to force *R_from_B2*.

It is notable that uncontrollable event *pt_ent_B2* gets added to ***TakeB2*** as part of explicitly specifying this forcing logic, and now ***TakeB2*** is also in charge of allowing/disallowing this event to occur in the system. In this case, designers need to effectively handle two more things. First, they must make sure that ***TakeB2*** should always allow *pt_ent_B2* to happen, when needed. In other words, ***TakeB2*** must never block *pt_ent_B2* when it is possible in

the plant model, otherwise the system will become uncontrollable. Second, designers must design ***TakeB2*** in such a way that *pt_ent_B2* interleaves properly with the other events of ***TakeB2***, (i.e. *R_to_B6* and *R_to_B7*), in order to prevent the violation of other desired SD properties and overall system specifications.

It is obvious that duplicating the behaviour of supervisor ***B2*** and specifying all this additional logic in relation to *pt_ent_B2* not only make the design of ***TakeB2*** more complicated, but also add more states to ***TakeB2***, thus increasing its state size. It is easy to see that this trend will continue to grow rapidly when system has several prohibitable events that are under the control of multiple modular supervisors (as evident in Section 10). Not to mention the extra effort and time that designers need to invest to manually figure out the logic and right time for forcing every single prohibitable event of the system, deal with the related complications, and then explicitly specify all this logic in the design of various modular TDES supervisors.

Clearly, an easier way is needed to determine and specify the logic of forcing prohibitable events. This method should simplify the TDES modelling process as well as the actual design of modular TDES supervisors in the SD supervisory control theory.

1.3 Proposed Solution

In the example discussed above, additional design logic and extra design effort is required because we need to make sure that system remains timed controllable, while we are trying to manually satisfy Point ii ($\Rightarrow$) of SD controllability by explicitly disabling *tick* and forcing the enabled prohibitable event. This makes us think that if "manual" disablement of *tick* event somehow gets eliminated, then we should be able to "automatically" satisfy Point ii ($\Rightarrow$) without having to worry about potentially disabling *tick* at the wrong time and making the system uncontrollable. Keeping this in view, we propose an approach to "automate" the *tick* disablement/event forcing mechanism to "automatically" satisfy the property checked by Point ii ($\Rightarrow$) of SD controllability in the SD supervisory control framework.

Our approach is inspired by the event forcing mechanism of physical controllers. In fact, we adopt the controllers' way of event forcing and apply it to the theoretical TDES setting. Controllers indicate the forcing of an event not by disabling the *tick*, but by enabling the event. If a controller wants an event to occur, it simply enables it. As soon as all the controllers that control this event enable it, the event occurs. In this case, none of the controllers is explicitly responsible for forcing the event.

This is exactly what our approach does in the SD supervisory control framework. Using our approach, if designers want to force a prohibitable event, they simply enable it in the modular supervisor without explicitly disabling the *tick*. As soon as the prohibitable event is enabled by all concerned supervisors and possible in the plant, *tick* "automatically" gets disabled to force the prohibitable event, thus automatically satisfying the property enforced by Point ii ($\Rightarrow$) of SD controllability. In this way, our approach essentially liberates the designers from manually keeping track of the enablement/disablement of *tick* and prohibitable events in various plant and supervisor models, and instead makes the forcing decision implicit.

In order to "automatically" disable *tick* in the presence of an eligible prohibitable event, we change the way we construct closed-loop system in the SD supervisory control framework. Specifically, we devise a new synchronization operator, named the *SD synchronous product* ($||_{SD}$), to form the closed-loop system. While synchronizing plant and supervisor models, our SD synchronous product operator checks to see that at a given state, whether *tick* and a

prohibitable event are both possible in the plant and enabled by all modular supervisors. If so, our synchronization operator disables *tick* event at the corresponding state of the closed-loop system without relying on any of the supervisor models to explicitly do this action.

This means that in the presence of our SD synchronous product operator, event forcing logic does not need to be explicitly specified in any of the supervisor models. As none of the modular supervisors is responsible for deciding when to force a prohibitable event, they no longer need to keep track of each others' behaviour. In other words, just like controllers, TDES supervisors are only concerned about their own behaviour in our approach. They simply enable the prohibitable event when they want it to occur.

In this way, by automating the *tick* disablement/event forcing mechanism in theoretical TDES setting, our approach aims at simplifying the design logic and modelling process of TDES supervisors, hence the overall system in the SD supervisory control framework. Also, it bridges the gap between theoretical TDES supervisors and physical controllers by making the event forcing mechanism of supervisors match with that of controllers. This makes the SD supervisory control methodology more accessible to software and hardware designers and practitioners.

1.3.1 Illustrative Example

Now we will redesign the TDES supervisor ***TakeB2*** (Figure 4) from the FMS example (introduced in Section 1.2.1) by taking into consideration the automatic *tick* disablement mechanism of our SD synchronous product operator. Figure 5 shows the TDES supervisor **TakeB2** that we have designed by applying our approach. Please note that although we are using the same name for our redesigned TDES supervisor, we have written it in a different text style (**bold**, instead of ***bold italics***) to clearly distinguish it from the original supervisor of the SD supervisory control framework.

In our approach, none of the modular supervisors has to be responsible for explicitly forcing the prohibitable event. This implies that **TakeB2** can simply enable the prohibitable event *R_from_B2* without explicitly deciding when to force it. That is why, we have enabled both *tick* and prohibitable event *R_from_B2* at state 0 of **TakeB2**, leaving it up to the SD synchronous product operator to automatically disable *tick* and make the forcing decision for us when *R_from_B2* is possible in TDES plant model **Robot** (Figure 2) and enabled by TDES supervisors ***B2*** (Figure 3) and **TakeB2**.

Since **TakeB2** has not explicitly disabled the *tick* event, there is no concern of making

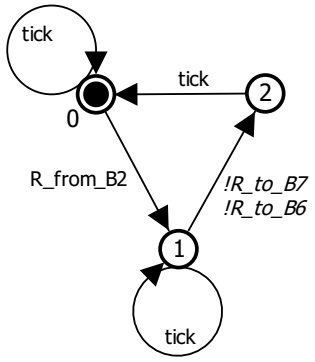


Figure 5: TDES Supervisor **TakeB2**

the system potentially uncontrollable by disabling *tick* at the wrong time. Also, the synchronization logic of our SD synchronous product operator guarantees that the property checked by Point ii ($\Rightarrow$) of SD controllability will always be satisfied at every state of the closed-loop system.

As **TakeB2** is not explicitly disabling *tick* while enabling *R_from_B2*, this implies that it neither needs to have knowledge about the behaviour of other TDES models, nor does it have to keep track of the part's progress. Therefore, supervisor **TakeB2** does not need to duplicate the design logic of supervisor ***B2***. In fact, there is no need to include the uncontrollable event *pt_ent_B2* in supervisor **TakeB2** at all. As a result, all concerns and issues that designers had to deal with after including *pt_ent_B2* in supervisor ***TakeB2*** automatically vanish by using our approach.

In this way, the 8-state supervisor ***TakeB2*** designed in the original SD supervisory control framework gets reduced to the 3-state supervisor **TakeB2** in the presence of our SD synchronous product operator. Also, it is evident that our approach has greatly simplified the design logic of **TakeB2** as compared to its corresponding supervisor ***TakeB2***, thus improving the ease of designing SD controllable TDES supervisors by hand.

1.4 Structure of the Report

The rest of this report is organized as follows.

Section 2 introduces the area of DES and TDES by describing the basic concepts and terminology used throughout this report. **Section 3** provides an overview of the sampled-data (SD) supervisory control theory [42, 43, 29] on which our work is based upon. This section covers all the essential aspects of the SD methodology needed in the following sections.

In **Section 4**, we present a novel mechanism for constructing closed-loop systems in the SD supervisory control framework. This section introduces our *SD synchronous product* ($||_{SD}$) synchronization operator, provides the adapted TDES and SD definitions, and describes our SD synchronous product setting ("$||_{SD}$ *setting*," from now on) in detail.

Our next task is to establish equivalence between the SD supervisory control setting (or "*SD setting*," for short) and our $||_{SD}$ setting. We discuss and formally prove this equivalence between the two settings in **Sections 5**$-$**7**.

In **Section 8**, we present the controllability and nonblocking verification results for our $||_{SD}$ setting. We formally prove that if a theoretical system designed in our $||_{SD}$ setting is controllable, nonblocking and satisfies the required properties, then the physically implemented system will also have these properties, and the system abides by the specified control laws. **Section 9** presents the predicate-based algorithms that we have adapted from [42] to verify various TDES and SD properties in our $||_{SD}$ setting. We have implemented these algorithms as part of a DES research tool, DESpot [14].

In **Section 10**, we discuss the example of a Flexible Manufacturing System (FMS) to demonstrate the application and utilization of our SD synchronous product operator and our $||_{SD}$ setting. This section also presents the verification results of the FMS example in our $||_{SD}$ setting. **Section 11** finishes off this report by stating our conclusions and discussing the future work.

In the appendices, we have included some content for the sake of completeness of this report. **Appendix A** gives miscellaneous definitions that are used in the fundamental DES concepts described in Section 2. **Appendix B** primarily presents some predicate-based algorithms from [42]. We are reusing these unmodified algorithms of the SD setting to verify some

properties in our $||_{SD}$ setting.

2 Preliminaries

This section presents a summary of the fundamental Discrete-Event System (DES) and Timed DES (TDES) terminology and concepts that we will use in this report. Details can be found in [45].

2.1 Linguistic Preliminaries

This section introduces key language concepts that are required to understand the terminology given in the following sections.

2.1.1 Strings

Let Σ be a finite set of distinct symbols (*events*). We refer to Σ as an *alphabet* e.g. $\Sigma = \{\alpha, \beta, \gamma, \sigma\}$. A *string* s over Σ is a finite sequence of events of the form $s = \sigma_1\sigma_2 \dots \sigma_n$, where $\sigma_i \in \Sigma$ and $0 \leq i \leq n$. A string with no events is called an *empty string*, denoted as ϵ, where $\epsilon \notin \Sigma$.

Let Σ^+ be the set of non-empty, finite sequences of events over Σ. We define Σ^* to be the set of all finite sequences of events over Σ, including the empty string ϵ. Thus, we have $\Sigma^* := \Sigma^+ \cup \{\epsilon\}$. Given a string $s = \sigma_1\sigma_2 \dots \sigma_n, |s| = n$ is the *length* of s. The empty string ϵ has a length of zero, i.e. $|\epsilon| = 0$.

Definition 2.1. Let $s, t \in \Sigma^*$, where $s = \alpha_1\alpha_2 \dots \alpha_m$ and $t = \beta_1\beta_2 \dots \beta_n$. The operation of *catenation* of strings s and t, $\text{cat} : \Sigma^* \times \Sigma^* \to \Sigma^*$, is defined as:

$$\begin{aligned} \text{cat}(\epsilon, s) = \text{cat}(s, \epsilon) &= s && s \in \Sigma^* \\ \text{cat}(s, t) = st &= \alpha_1\alpha_2 \dots \alpha_m\beta_1\beta_2 \dots \beta_n && s, t \in \Sigma^+ \end{aligned}$$

As $|s| = m$ and $|t| = n$, the length of catenated string is $|\text{cat}(s, t)| = |s| + |t| = m + n$.

Definition 2.2. For some $s, t \in \Sigma^*$, we say that t is a *prefix* of s, written as $t \leq s$, if $(\exists u \in \Sigma^*)\, s = tu$.

By definition, a string $s \in \Sigma^*$ is a prefix of itself, since $s \leq s$. Also, we have that ϵ is a prefix of all strings, since $(\forall s \in \Sigma^*)\, \epsilon \leq s$.

2.1.2 Languages

Languages are used to represent system behaviour. A language is defined as a set of strings. Formally, a *language* L over Σ is any subset of Σ^*, i.e. $L \subseteq \Sigma^*$.

Definition 2.3. The *prefix closure* of language $L \subseteq \Sigma^*$ is the language $\overline{L}$, defined as $\overline{L} := \{t \in \Sigma^* \mid t \leq s \text{ for some } s \in L\}$.

This definition says that $\overline{L}$ consists of all prefixes of strings of L. By definition, a language L is a subset of the prefix closure of itself, i.e. $L \subseteq \overline{L}$. A language L is said to be *prefix-closed* if $L = \overline{L}$.

Let $\text{Pwr}(\Sigma)$ denote the set of all possible subsets of Σ. For $\sigma \in \Sigma$, we will use the notation $\Sigma^*.\sigma$ to represent the set of all strings $s\sigma$ for some $s \in \Sigma^*$.

Definition 2.4. For language $L \subseteq \Sigma^*$ and string $s \in \Sigma^*$, the *eligibility operator* $Elig_L : \Sigma^* \to \text{Pwr}(\Sigma)$ is defined as $Elig_L(s) := \{\sigma \in \Sigma \mid s\sigma \in L\}$.

In simple words, the eligibility operator returns a set of events $\sigma \in \Sigma$ that can follow string s to create a string $s\sigma \in L$.

2.1.3 Nerode Equivalence Relation

Definition 2.5. The *nerode equivalence relation*[2] on Σ^* with respect to L, i.e. Σ^* mod L, is defined as $(\forall s, t \in \Sigma^*)\, s \equiv_L t$ or $s \equiv t \,(\text{mod } L)$ iff $(\forall u \in \Sigma^*)\, su \in L$ iff $tu \in L$.

This definition states that two strings s and t are nerode equivalent with respect to L if and only if they can be extended by any string $u \in \Sigma^*$ such that either both strings are in L or neither string is in L.

2.2 Discrete Event Systems

Supervisory control theory (SCT) [46, 34] provides a formal framework for the analysis and control of discrete-event systems (DES). SCT is automaton-based and models DES as the generator of a formal language. The uncontrolled behaviour of the system of interest, modelled by an automaton, is referred to as the *plant* DES. The desired behaviour of the controlled plant is that its generated language be contained in a specification language. To achieve this desired behaviour as per the given specifications, a *supervisor* DES, modelled by an automaton, is introduced. Supervisor DES alters unrestricted behaviour of the plant DES within prescribed limits by operating synchronously with it and using a feedback control mechanism.

This section presents the formal DES representation and fundamental concepts related to DES.

2.2.1 Generator

Definition 2.6. A DES is formally represented as a *generator* which is defined as a 5-tuple:

$$\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$$

where Q is the *state set*, Σ is the *event set*, $\delta : Q \times \Sigma \to Q$ is the partial *transition function*, $q_o \in Q$ is the *initial state*, and $Q_m \subseteq Q$ is the *set of marked states*.

The event set Σ of DES $\mathbf{G}$ can be partitioned into the set of *controllable events* (Σ_c) and *uncontrollable events* (Σ_u), i.e. $\Sigma = \Sigma_c \,\dot\cup\, \Sigma_u$, where $\dot\cup$ represents *disjoint union* of the two sets, Σ_c and Σ_u. Controllable events can be enabled or disabled by a supervisor, and can occur only when a supervisor enables them. On the other hand, uncontrollable events are not under the control of the supervisor. These events are assumed to be always enabled. Once the plant DES reaches a state where an uncontrollable event is possible, this event cannot be prevented from occurrence.

Each transition in δ is a 3-tuple (or *triple*) of the form (q, σ, q'), where $\delta(q, \sigma) = q'$ such that $q, q' \in Q$ and $\sigma \in \Sigma$. We refer to q as the *exit (source) state* and q' as the *entrance (destination) state*.

[2] See Definition *A*.1 of *equivalence relation* in Appendix A.

The notation $\delta(q,\sigma)!$ means the transition is defined at state $q \in Q$ for event $\sigma \in \Sigma$. We extend δ to $\delta : Q \times \Sigma^* \to Q$ in the natural way as:

$$\begin{aligned} \delta(q,\epsilon) &= q && \text{for } q \in Q \\ \delta(q,s\sigma) &= \delta(\delta(q,s),\sigma) && \text{for } q \in Q, s \in \Sigma^* \text{ and } \sigma \in \Sigma, \text{ as long as } q' := \delta(q,s)! \text{ and } \delta(q',\sigma)! \end{aligned}$$

For the following definitions, let DES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$.

Definition 2.7. A state $q \in Q$ is *reachable* in $\mathbf{G}$ if $(\exists s \in \Sigma^*)\, \delta(q_o, s)!$ & $\delta(q_o, s) = q$.

This definition states that a state q is reachable if, starting from the initial state q_o, there exists a string $s \in \Sigma^*$ that can take us to state q.

Definition 2.8. The *reachable state subset* Q_r of $\mathbf{G}$ is defined as:

$$Q_r := \{q \in Q \mid (\exists s \in \Sigma^*)\ \delta(q_o, s) = q\}$$

Definition 2.9. A DES $\mathbf{G}$ is *reachable* if all of its states are reachable, i.e. $Q_r = Q$.

Definition 2.10. A DES $\mathbf{G}$ is said to be *deterministic* if it has a single initial state, and for each $q \in Q$, and each $\sigma \in \Sigma$, there is at most one σ transition leaving q.

Note: In this report, we always assume that a DES is reachable, deterministic and has a finite state space and a finite event set.

Definition 2.11. The *closed behaviour* of $\mathbf{G}$ is defined as $L(\mathbf{G}) := \{s \in \Sigma^* \mid \delta(q_o, s)!\}$.

In simple words, we say that $L(\mathbf{G})$ represents all possible sequences of events that could occur in the system. Clearly, $\epsilon \in L(\mathbf{G})$ as long as $Q \neq \emptyset$.

Definition 2.12. The *marked behaviour* of $\mathbf{G}$ is defined as:

$$L_m(\mathbf{G}) := \{s \in \Sigma^* \mid \delta(q_o, s)!\ \ \&\ \ \delta(q_o, s) \in Q_m\}$$

The marked behaviour of $\mathbf{G}$ is interpreted as representing the set of all strings in Σ^* that start at q_o and end at a state in Q_m. Marked behaviour represents "completed" tasks carried out by the system that $\mathbf{G}$ is intended to model. Clearly, $L_m(\mathbf{G}) \subseteq L(\mathbf{G})$.

Definition 2.13. A DES $\mathbf{G}$ is said to be *nonblocking* if $\overline{L_m(\mathbf{G})} = L(\mathbf{G})$.

This definition says that any string that can be generated by $\mathbf{G}$ is a prefix of (i.e. can always be extended to) a marked string of $\mathbf{G}$. In other words, every string in $L(\mathbf{G})$ can be extended to a completed task in $L_m(\mathbf{G})$.

Definition 2.14. For DES $\mathbf{G}$, let λ be an equivalence relation[3] on Q such that $(\forall q, q' \in Q)$ $q \equiv q' \pmod{\lambda}$ if and only if:

1. $(\forall s \in \Sigma^*)\, \delta(q, s)! \Leftrightarrow \delta(q', s)!$
2. $(\forall s \in \Sigma^*)\, \delta(q, s)!\ \ \&\ \ \delta(q, s) \in Q_m \Leftrightarrow \delta(q', s)!\ \ \&\ \ \delta(q', s) \in Q_m$

This definition means that for states q and q' such that $q \equiv q' \pmod{\lambda}$, they have the same future with respect to the closed behaviour $L(\mathbf{G})$ and marked behaviour $L_m(\mathbf{G})$. Based on this, for string $s \in L(\mathbf{G})$, a state $q = \delta(q_o, s)$ represents all strings in Σ^* that are nerode equivalent to s mod $L(\mathbf{G})$ and mod $L_m(\mathbf{G})$.

The λ-equivalence relation allows us to reduce a reachable generator to a minimal state version that represents the same closed and marked behaviour.

Definition 2.15. A DES $\mathbf{G}$ is said to be *minimal* if $(\forall q, q' \in Q)\, q \equiv q' \pmod{\lambda} \Leftrightarrow q = q'$.

[3] See Definition *A*.1 of *equivalence relation* in Appendix A.

This definition states that for all states $q, q' \in Q$, q is equivalent to q' (mod λ) if and only if q and q' are the same state. In other words, $\mathbf{G}$ is minimal if it does not have two distinct states q and q' in Q that are λ-equivalent.

2.2.2 DES Synchronization

From the designer's point of view, it is often easier to model the system as several smaller DES components, rather than designing the whole system as a single, more complex DES all at once. These multiple DES components are synchronized together using a *synchronization operator* to construct the complete system. The commonly used synchronization operators include the synchronous product, the product[4], and the meet[5] operator. Before defining the synchronous product operator formally, first we will introduce the *natural projection* operator and its *inverse*.

Natural Projection Let $L_i \subseteq \Sigma_i^*$, for $i = 1, 2$. Let $\Sigma = \Sigma_1 \cup \Sigma_2$.

Definition 2.16. The *natural projection* P_i of Σ^* onto Σ_i^*, i.e. $P_i : \Sigma^* \to \Sigma_i^*$, is defined as:

$$P_i(\epsilon) = \epsilon$$
$$P_i(\sigma) = \begin{cases} \epsilon & \text{if } \sigma \notin \Sigma_i \\ \sigma & \text{if } \sigma \in \Sigma_i \end{cases}$$
$$P_i(s\sigma) = P_i(s)P_i(\sigma) \quad \text{for } s \in \Sigma^*, \sigma \in \Sigma$$

This definition says that the action of P_i on a string s is to erase all occurrences of $\sigma \notin \Sigma_i$, that are in s.

Definition 2.17. Let $P_i^{-1} : \text{Pwr}(\Sigma_i^*) \to \text{Pwr}(\Sigma^*)$ be the *inverse image function* of P_i, namely for $L \subseteq \Sigma_i^*$, we have $P_i^{-1}(L) := \{s \in \Sigma^* \mid P_i(s) \in L\}$.

Synchronous Product First, we will define the synchronous product of two languages L_1 and L_2 in terms of natural projection.

Definition 2.18. Let $L_i \subseteq \Sigma_i^*$, for $i = 1, 2$. The *synchronous product* $L_1 \,||\, L_2 \subseteq \Sigma^*$ is defined as $L_1 \,||\, L_2 := P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$.

Thus, $s \in L_1 \,||\, L_2$ if and only if $P_1(s) \in L_1$ and $P_2(s) \in L_2$.

Now, we will define the synchronous product of two DES $\mathbf{G}_1$ and $\mathbf{G}_2$.

Definition 2.19. Let $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$, for $i = 1, 2$. The *synchronous product* of the two DES, represented as $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2$, is defined as:

$$\mathbf{G} := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$$

where $\delta((q_1, q_2), \sigma)$ is only defined and equals:

$$(q_1', q_2') \text{ if } \sigma \in (\Sigma_1 \cap \Sigma_2), \delta_1(q_1, \sigma) = q_1', \delta_2(q_2, \sigma) = q_2' \text{ or}$$
$$(q_1', q_2) \text{ if } \sigma \in \Sigma_1 - \Sigma_2, \delta_1(q_1, \sigma) = q_1' \text{ or}$$
$$(q_1, q_2') \text{ if } \sigma \in \Sigma_2 - \Sigma_1, \delta_2(q_2, \sigma) = q_2'$$

Let $L(\mathbf{G}_1)$ and $L(\mathbf{G}_2)$ be the closed behaviour, and $L_m(\mathbf{G}_1)$ and $L_m(\mathbf{G}_2)$ be the marked behaviour of $\mathbf{G}_1$ and $\mathbf{G}_2$ respectively. Synchronizing $\mathbf{G}_1$ and $\mathbf{G}_2$ using the synchronous product

[4]See Definition *A*.2 of *product* operator in Appendix A.
[5]See Definition *A*.3 of *meet* operator in Appendix A.

operator will generate the closed and marked behaviour of the resultant DES $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2$ as $L(\mathbf{G}) = L(\mathbf{G}_1) \,||\, L(\mathbf{G}_2)$ and $L_m(\mathbf{G}) = L_m(\mathbf{G}_1) \,||\, L_m(\mathbf{G}_2)$ respectively.

It follows from Definition 2.18 that:

$$L(\mathbf{G}) = P_1^{-1}(L(\mathbf{G}_1)) \cap P_2^{-1}(L(\mathbf{G}_2)) \quad \text{and} \quad L_m(\mathbf{G}) = P_1^{-1}(L_m(\mathbf{G}_1)) \cap P_2^{-1}(L_m(\mathbf{G}_2))$$

If both $\mathbf{G}_1$ and $\mathbf{G}_2$ are defined over the same alphabet Σ, i.e. $\Sigma = \Sigma_1 = \Sigma_2$, then:

$$L(\mathbf{G}) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2) \quad \text{and} \quad L_m(\mathbf{G}) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$$

However, if $\mathbf{G}_1$ and $\mathbf{G}_2$ are not defined over the same alphabet Σ, we can simply add *selfloops*[6] to each DES for the missing events at every state to extend them over the same event set Σ, without any loss of generality.

It is important to note here that if $\Sigma = \Sigma_1 = \Sigma_2$, then synchronizing $\mathbf{G}_1$ and $\mathbf{G}_2$ using the synchronous product, product and meet operator will generate the same closed and marked language of the resultant DES $\mathbf{G}$. In this case, we have:

$$L(\mathbf{G}) = L(\mathbf{G}_1 \,||\, \mathbf{G}_2) = L(\mathbf{G}_1 \times \mathbf{G}_2) = L(\mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2)$$
$$L_m(\mathbf{G}) = L_m(\mathbf{G}_1 \,||\, \mathbf{G}_2) = L_m(\mathbf{G}_1 \times \mathbf{G}_2) = L_m(\mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$$

Note: In this report, we assume that we have $m > 1$ plant DES components, $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_m$, and that they are always combined using the synchronous product operator to obtain the composite plant DES $\mathbf{G}$, i.e. $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2 \,||\, \ldots \,||\, \mathbf{G}_m$. Likewise, we have $n > 1$ modular supervisor DES, $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n$, and they are always assumed to be combined using the synchronous product to construct the supervisor DES $\mathbf{S}$, i.e. $\mathbf{S} = \mathbf{S}_1 \,||\, \mathbf{S}_2 \,||\, \ldots \,||\, \mathbf{S}_n$. We also assume that both $\mathbf{G}$ and $\mathbf{S}$ are always defined over the same event set Σ, either by modelling the system in this way or by explicitly adding selfloops later on, unless stated otherwise.

2.2.3 Controllability

Let DES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and DES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. As per Definition 2.6, $\Sigma = \Sigma_c \,\dot\cup\, \Sigma_u$.

In order to construct the *closed-loop system*, we synchronize plant $\mathbf{G}$ and supervisor $\mathbf{S}$ using a synchronization operator. The behaviour of $\mathbf{G}$ under the control of $\mathbf{S}$ is referred to as the *closed-loop behaviour* of the system.

Definition 2.20. Supervisor $\mathbf{S}$ is *controllable* with respect to plant $\mathbf{G}$ if:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))\,(\forall \sigma \in \Sigma_u)\, s\sigma \in L(\mathbf{G}) \Rightarrow s\sigma \in L(\mathbf{S})$$

This definition can be restated in terms of the eligibility operator as follows:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))\, Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\mathbf{S})}(s)$$

This definition states that for all legal strings s that are possible in the closed-loop system, an uncontrollable event must be allowed by $\mathbf{S}$ if it is possible in $\mathbf{G}$ after s.

Note: In this report, as we will be focusing on timed DES models (introduced in the next section), we will refer to this definition explicitly as the "untimed controllability" definition.

2.3 Timed DES

Timed DES (TDES), introduced by [5, 7], is a discrete-time model that extends untimed DES theory by adding a new event called the *tick* (τ) event. The *tick* event represents the passage

[6] See Definition *A*.4 of *selfloop* operation in Appendix A.

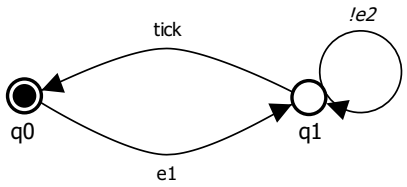


Figure 6: An Example TDES Automaton

of one time unit, and corresponds to the tick of a global clock that the system is assumed to be synchronized with. Thus, the event set of a TDES contains the *tick* event as well as other non-*tick* events called *activity events* (Σ_{act}).

Definition 2.21. A TDES automaton **G** is formally represented as a 5-tuple:

$$\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$$

where Q is the *state set*, $\Sigma = \Sigma_{act} \dot\cup \{\tau\}$ is the *event set*, the partial function $\delta : Q \times \Sigma \to Q$ is the *transition function*, $q_o \in Q$ is the *initial state*, and $Q_m \subseteq Q$ is the *set of marked states*. We extend δ to $\delta : Q \times \Sigma^* \to Q$ in the natural way.

TDES contain *forcible events* (Σ_{for}) and *prohibitable events* (Σ_{hib}). Forcible events represent a class of non-*tick* events which are guaranteed to occur before the next clock *tick*, when required. Hence, they can be relied upon to preempt the *tick* event, when needed. The method used by a TDES supervisor to indicate that an event $\sigma \in \Sigma_{for}$ should be forced at a given state, is to disable *tick* at this state. This has the effect of removing the now impossible behaviour that *tick* could occur before σ. Prohibitable events are non-*tick* events that can be enabled or disabled by a supervisor.

Like a DES generator (Definition 2.6), the event set Σ of a TDES automaton can be partitioned into the set of *controllable events* (Σ_c) and *uncontrollable events* (Σ_u), i.e. $\Sigma = \Sigma_c \dot\cup \Sigma_u$. The set of controllable events in TDES theory is $\Sigma_c = \Sigma_{hib} \dot\cup \{\tau\}$, where $\Sigma_{hib} \subseteq \Sigma_{act}$. The set of uncontrollable events is $\Sigma_u = \Sigma - \Sigma_c = \Sigma_{act} - \Sigma_{hib}$.

Let us consider a TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ with the following tuple information:

State set: $Q = \{\text{q0, q1}\}$

Event set: $\Sigma = \{\text{e1, e2, tick}\}$, where $\Sigma_c = \{\text{e1, tick}\}, \Sigma_u = \{\text{e2}\}, \Sigma_{act} = \{\text{e1, e2}\}$ and $\Sigma_{hib} = \{\text{e1}\}$

Transition function: $\delta = \{(\text{q0, e1, q1}), (\text{q1, e2, q1}), (\text{q1, tick, q0})\}$

Initial state: $q_o = \text{q0}$

Set of marked states: $Q_m = \{\text{q0}\}$

This TDES **G** is represented graphically in Figure 6. The states of **G**, q0 and q1, are equated with the nodes (circles) of the graph. Transitions are represented by arrows. Arrows are labelled by events, e1, e2 and *tick*, in Σ. The event name e2 in italics and preceded by "!", indicates that the event is uncontrollable. The initial state q0 is represented by a double circle, whereas a filled circle shows that q0 is also a marked state.

Note: In this report, we will use the above-mentioned graphical notation to represent our TDES models.

Since TDES framework is an extension of the DES theory, therefore all DES concepts and properties presented in the previous sections remain valid and applicable to TDES theory. In the following sections, we introduce/restate only those definitions that are specific to TDES framework.

2.3.1 Controllability and Supervision

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor.

Definition 2.22. TDES supervisor $\mathbf{S}$ is *timed controllable* with respect to TDES plant $\mathbf{G}$ if $(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))$,

$$Elig_{L(\mathbf{S})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{for} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

This definition states that supervisor $\mathbf{S}$ must accept an uncontrollable event if it is possible in the plant $\mathbf{G}$ after a legal string s. In addition, $\mathbf{S}$ must enable a *tick* event if it is possible in $\mathbf{G}$, unless there exists an eligible forcible event in the system to preempt the *tick*.

Note: In this report, as we will only be dealing with TDES models, therefore we will drop the word "timed", and will refer to this property as "$\mathbf{S}$ is controllable with respect to $\mathbf{G}$" for simplicity.

Definition 2.23. A *TDES supervisory control* for $\mathbf{G}$ is any map $V : L(\mathbf{G}) \to \text{Pwr}(\Sigma)$ such that $(\forall s \in L(\mathbf{G}))$,

$$V(s) \supseteq \begin{cases} \Sigma_u \cup (\{\tau\} \cap Elig_{L(\mathbf{G})}(s)) & \text{if } V(s) \cap Elig_{L(\mathbf{G})}(s) \cap \Sigma_{for} = \emptyset \\ \Sigma_u & \text{if } V(s) \cap Elig_{L(\mathbf{G})}(s) \cap \Sigma_{for} \neq \emptyset \end{cases}$$

In the following definitions, we write $V/\mathbf{G}$ to denote the pair $(\mathbf{G}, V)$, i.e. to represent $\mathbf{G}$ under the supervision of V.

Definition 2.24. The *closed behaviour* of $V/\mathbf{G}$ is the language $L(V/\mathbf{G}) \subseteq L(\mathbf{G})$ defined inductively as follows:

i. $\epsilon \in L(V/\mathbf{G})$

ii. If $s \in L(V/\mathbf{G}), \sigma \in V(s)$, and $s\sigma \in L(\mathbf{G})$ then $s\sigma \in L(V/\mathbf{G})$

iii. No other strings belong to $L(V/\mathbf{G})$

$L(V/\mathbf{G})$ is prefix-closed, nonempty, and in the range $\{\epsilon\} \subseteq L(V/\mathbf{G}) \subseteq L(\mathbf{G})$.

Definition 2.25. The *marked behaviour* of $V/\mathbf{G}$, $L_m(V/\mathbf{G})$, is defined as:

$$L_m(V/\mathbf{G}) := L(V/\mathbf{G}) \cap L_m(\mathbf{G})$$

Definition 2.26. V is said to be *nonblocking for* $\mathbf{G}$ if $\overline{L_m(V/\mathbf{G})} = L(V/\mathbf{G})$.

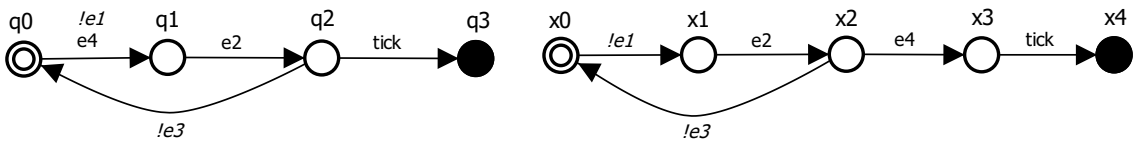
2.3.2 Control Equivalent Supervisors

Let $\mathbf{G} = (Q, \Sigma_\mathbf{G}, \delta, q_o, Q_m)$ be a TDES plant. Let $\mathbf{S}_1 = (X_1, \Sigma_1, \xi_1, x_{o,1}, X_{m,1})$ and $\mathbf{S}_2 = (X_2, \Sigma_2, \xi_2, x_{o,2}, X_{m,2})$ be two TDES supervisors.

Definition 2.27. Supervisors $\mathbf{S}_1$ and $\mathbf{S}_2$ are considered to be *control equivalent* for a given plant $\mathbf{G}$, if they produce the same closed-loop behaviour.

As this definition specifically focuses on the "closed-loop behaviour", two points are notable and worth elaborating.

1. This definition does not make any assumptions about how the two closed-loop systems are constructed, i.e. it is independent of the synchronization operators that are used to form the two closed-loop systems. The two supervisors $\mathbf{S}_1$ and $\mathbf{S}_2$ may be combined with $\mathbf{G}$ using the same synchronization operator, e.g. synchronous product, or two different synchronization operators, e.g. $\mathbf{S}_1$ is combined with $\mathbf{G}$ using synchronous product, and



(a) TDES Plant **G** (b) TDES Supervisor **S**

Figure 7: An Example to Illustrate Various TDES Properties

$\mathbf{S}_2$ is combined with **G** using the sampled-data synchronous product operator (introduced in Section 4.1). As long as the closed-loop behaviour of the two systems is the same, the definition remains applicable and valid, and the choice of synchronization operator(s) is not important. This will allow us to compare the action of two supervisors that are combined with the same plant, but using different operators to construct the closed-loop systems.

2. The definition is given with respect to the closed-loop behaviour of the two systems, i.e. the closed and marked languages, and not in terms of the actual closed-loop system automata. This is because a TDES representation of the two closed-loop systems having the same closed-loop behaviour might not be exactly the same due to different state labels. They might not even be identical up to state relabelling as one TDES could be in its minimal form and the other one could be a non-minimal version. However, irrespective of their minimal or non-minimal representation, their closed and marked languages will still be same.

Based on the above discussion, we can restate the definition of two supervisors being control equivalent for a given plant model (Definition 2.27).

Definition 2.28. Let $\mathbf{G}_{cl,1}$ be the closed-loop system that is constructed by synchronizing $\mathbf{S}_1$ and **G**, and let $\mathbf{G}_{cl,2}$ be the closed-loop system that is formed by combining $\mathbf{S}_2$ and **G**. Then $\mathbf{S}_1$ and $\mathbf{S}_2$ are said to be *control equivalent* for **G** if $L(\mathbf{G}_{cl,1}) = L(\mathbf{G}_{cl,2})$ and $L_m(\mathbf{G}_{cl,1}) = L_m(\mathbf{G}_{cl,2})$.

2.3.3 TDES Properties

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor.

We will use an example TDES plant **G** (Figure 7a) and TDES supervisor **S** (Figure 7b) shown in Figure 7 to illustrate various TDES properties.

First, we want to impose a technical condition on our TDES to exclude the physically unrealistic possibility that a *tick* transition might be preempted indefinitely by repeated execution of an activity loop within a fixed unit time interval.

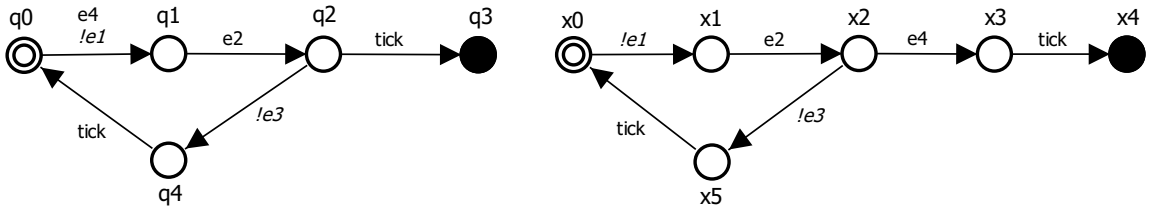
Definition 2.29. TDES **G** is said to have an *activity-loop* if $(\exists q \in Q)\,(\exists s \in \Sigma^+_{act})\ \delta(q, s) = q$.

In Figure 7a, **G** has activity loops of "e1-e2-e3-e1" and "e4-e2-e3-e4" that could preempt the *tick* event from occurring for an indefinite amount of time. Likewise, *tick* event in **S** can be preempted indefinitely by repeated execution of "e1-e2-e3-e1" activity loop, as shown in Figure 7b. To rule this out, we require that a TDES must be activity-loop-free.

Definition 2.30. TDES **G** is *activity-loop-free (ALF)* if $(\forall q \in Q_r)\,(\forall s \in \Sigma^+_{act})\ \delta(q, s) \neq q$.

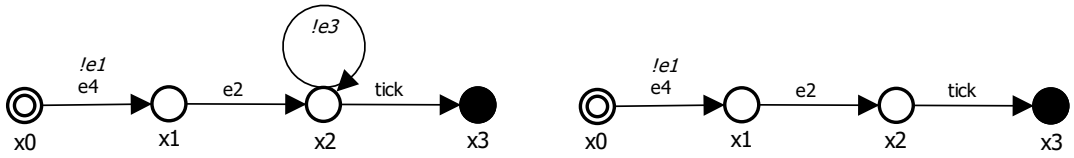
Please note that this definition is given in terms of only the reachable states, since unreachable states do not contribute to the closed and marked behaviour of a TDES.

One simple way to make our **G** and **S** of Figure 7 ALF is by adding a *tick* transition after transition 'e3'. This ALF version of **G** and **S** is shown in Figure 8.



(a) ALF TDES Plant **G** (b) ALF TDES Supervisor **S**

Figure 8: An Example Satisfying ALF Property



(a) Non-Selfloop ALF TDES **S** (b) ALF TDES **S**′

Figure 9: An Example Illustrating Non-Selfloop ALF Property

Practically, it is not always possible to make supervisors ALF, as they typically have selfloops of activity events. However, these selflooped events are sometimes not possible in the plant model, thus making the closed-loop system ALF. Therefore, in the sampled-data supervisory control theory [42, 29], the authors desire that their supervisors should preferably satisfy a less restrictive condition of being non-selfloop ALF.

Definition 2.31. Let **S** be a TDES, and let **S**′ be **S** with all activity event selfloops removed. **S** is *non-selfloop ALF* if **S**′ is ALF.

This definition states that if we remove all activity event selfloops from a non-selfloop ALF TDES **S**, then it must become ALF.

A non-selfloop ALF TDES **S** is shown in Figure 9a. If we remove the selfloop of activity event e3 at state x2, the TDES becomes ALF, as shown in Figure 9b.

The following defintion is taken from [44]. Only plant TDES are required to satisfy this property.

Definition 2.32. TDES **G** has *proper time behaviour* if $(\forall q \in Q_r)\,(\exists \sigma \in \Sigma_u \cup \{\tau\})\ \delta(q, \sigma)!$.

It says that at each reachable state, either an uncontrollable event or a *tick* event must be possible. This ensures that a TDES can never express that a prohibitable event must occur before the next *tick*, since a supervisor could disable that prohibitable event, thus "stopping the clock". This is neither desirable nor realistic.

TDES plant **G** shown in Figure 7a does not have proper time behaviour. The reason is that at state q1, neither an uncontrollable event nor *tick* event is possible. The only event possible at state q1 is the prohibitable event e2.

Usually, controllable events are often part of the supervisor's implementation. This means that supervisors can make these events to occur at any time, even when the plant model says they can't. In order to prevent the violation of the plant model, the property of plant completeness was defined with respect to controllable events by [3]. It has been adapted to use only prohibitable events for the sampled-data supervisory control theory [29], which is the basis of our work.

Definition 2.33. A TDES plant **G** is *complete* for TDES supervisor **S** if:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}))\,(\forall \sigma \in \Sigma_{hib})\ s\sigma \in L(\mathbf{S}) \Rightarrow s\sigma \in L(\mathbf{G})$$

This definition states that for every state in $\mathbf{G}$, if a prohibitable event σ is enabled by $\mathbf{S}$, it must be possible in $\mathbf{G}$. This condition can be seen as dual to the definition of $\mathbf{S}$ being controllable with respect to $\mathbf{G}$ (Definition 2.22).

In Figure 7, $\mathbf{S}$ enables prohibitable event e4 at state x2. However, event e4 is not possible in $\mathbf{G}$ at state q2, thus violating the property of plant completeness.

3 Sampled-Data Supervisory Control

Sampled-Data (SD) supervisory control theory [42, 43, 29] focuses on the implementation of TDES supervisors as SD controllers. It establishes sufficient conditions to ensure that if a theoretical TDES is controllable, nonblocking, and satisfies these properties, then the physical implementation will also have these properties and exhibit correct behaviour as specified by the control laws.

In this section, we will only focus on those aspects of the SD methodology that are required to follow our work presented in the following sections. To gain a thorough understanding of the SD supervisory control theory, please refer to [42, 43, 29].

It is worth clarifying here that in the SD supervisory control setting (or "*SD setting,*" for short) described in [42], the closed-loop system is constructed by combining the TDES plant $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ and the TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ using the **meet** operator, i.e. $\mathbf{meet}(\mathbf{G}, \mathbf{S})$, and all theoretical proofs and results are given in terms of the **meet**. However, in [43, 29], the product operator is used to form the closed-loop system, expressed as $\mathbf{G} \times \mathbf{S}$, and discuss all verification results.

As noted in Section 2.2.2, if $\mathbf{G}$ and $\mathbf{S}$ are both defined over the same event set, then $\mathbf{meet}(\mathbf{G}, \mathbf{S})$, $\mathbf{G} \times \mathbf{S}$, and $\mathbf{G} \,||\, \mathbf{S}$ will produce the same closed and marked behaviours, and can thus be used interchangeably. To keep things simple and consistent throughout this report, we will use the synchronous product operator to discuss the SD supervisory control framework. In this case, we assume that $\mathbf{S}$ and $\mathbf{G}$ are defined over the same event set. We will thus define the closed-loop system to be $\mathbf{S} \,||\, \mathbf{G}$. The system's closed behaviour is thus defined as $L(\mathbf{S} \,||\, \mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$, and its marked behaviour as $L_m(\mathbf{S} \,||\, \mathbf{G}) = L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

3.1 SD Controllers

A *sampled-data (SD) controller* is driven by a global periodic clock whose clock edge is associated with the *tick* (τ) event of the TDES. It views the system as a series of inputs and outputs that can take the values of *True* and *False* only. On the rising edge of the clock, it samples its inputs, changes its state based on the inputs and current state, and updates its outputs based on the new state it has transitioned to.

To use an SD controller to manage a given system, an input is associated with each non-*tick* event, called an *activity event*, and an output with each non-*tick* controllable event, called a *prohibitable event*. The occurrence of an event is indicated by its input going true during a given clock period. A prohibitable event is considered enabled when its corresponding output has been set true by the controller, disabled otherwise. If a prohibitable event is enabled at a given state, the controller will always make sure it happens before the next clock edge. For example, in a digital logic implementation, the output set to true is usually taken to mean that the event has occurred.

An SD controller samples inputs, changes state and updates outputs only on a clock edge. This has the following implications: 1) An SD controller knows nothing about the occurrence

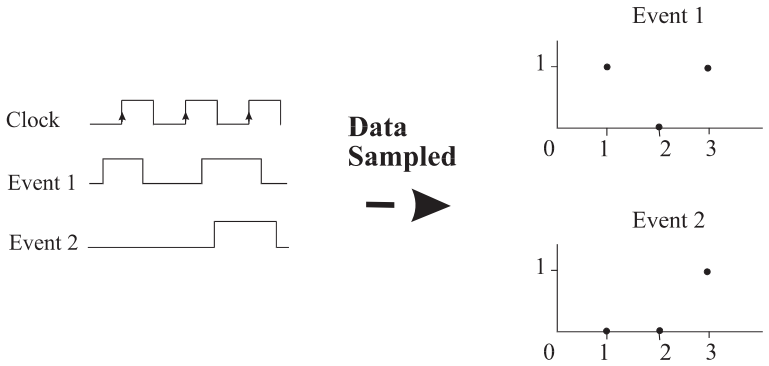


Figure 10: An Example for Event Sampling *(Reprinted from [42])*

of events in a given sampling period (clock period) until the next clock edge. 2) On the next clock edge, the only information it receives is which events have occurred in a given sampling period. 3) Neither does it know anything about the order the events occurred in, nor the number of times an event has occurred in a given sampling period. 4) An SD controller updates the enablement and forcing information on the clock edge and then keeps it unchanged for the entire clock period.

Figure 10 shows an example of event sampling with respect to an SD controller. The left figure shows that $Event1$ and $Event2$ occurred in the 2^{nd} sampling period. However, an SD controller will know nothing about the occurrence of these events until the next clock edge, i.e. 3^{rd} rising edge of the clock. On the next clock edge, the only information it receives is that $Event1$ and $Event2$ occurred in the sampling period that has just ended, without any information about the order or frequency of occurrence of these events (right figure). This means that an SD controller will not know about the exact string that actually happened in the last sampling period, and cannot differentiate between strings such as "Event1-Event2-τ", "Event2-Event1-τ", "Event1-Event2-Event1-τ" or "Event2-Event2-Event1-τ".

3.2 Concurrency and Timing Issues

Timed DES theory assumes that: 1) events occur in an interleaving fashion (we can always determine the event ordering), 2) we know immediately when events occur, and 3) enablement and forcing occur immediately (i.e. no communication delay).

Because these assumptions are not true in general for SD controllers, several concurrency and timing issues arise when representing TDES supervisors as deterministic SD controllers. For example, if multiple forcible events are enabled in a single clock period (i.e. there is a *choice*), how does the controller decide which events to generate/force in the current clock period, and in which order to force the events? Likewise, if an event is enabled for multiple clock periods (say 3 clock periods), how does the controller decide when to force it and in which clock period (force it in the 1^{st}, 2^{nd} or 3^{rd} clock period)? Also, if an SD controller is forcing multiple events (say e1 and e2) in the same clock period, these events may only actually occur in a specific order (say "e1-e2" only) even though the TDES model says they can occur in multiple orderings (say "e2-e1" as well). This could even vary from one implementation to the other.

These issues have ramifications with respect to controllability, plant model correctness, and the SD controller's ability to determine which state the TDES currently is in. They could make the controller implementation block, uncontrollable, or violate the specified control laws, even

though our original TDES is nonblocking and controllable. Also, these issues are important for the unambiguous translation of TDES supervisors into SD controllers and to obtain a deterministic controller.

These issues are primarily addressed in the SD supervisory control framework by introducing the property of SD controllability (Section 3.5).

3.3 SD Assumptions

The SD approach makes the following assumptions that must be met by the system designer while developing the TDES models.

1. The set of prohibitable events is exactly equal to the set of forcible events, i.e. $\Sigma_{for} = \Sigma_{hib}$.
2. A prohibitable event is forced in the same sampling period in which it is enabled. It is only allowed to occur once per clock period.
3. When an event is forced in a given sampling period, no assumptions are made about exactly when the event will occur during that clock period. This is because timing may vary depending upon the controller's implementation.
4. The SD controllers will be implemented centrally with a common clock such that they all are synchronized, i.e. they all sample inputs and update outputs at the same time. Moreover, the controllers generate all prohibitable events, so that there is no issue of communication delay with respect to event enablement/disablement.
5. An event is assumed to have "occurred" when its input goes true. If this happens so close to the clock edge that it shows up in the next sampling period, then it "occurs" immediately after the clock edge. The system designer should reflect this in the plant model.
6. The length of an input pulse should be appropriate to be detected and interpreted correctly by the controller. It should not be so short that it could be missed by the controller (i.e. occurs between two clock edges). It should also not be so long that the controller sees and interprets it as an event occurring multiple times in different clock periods, when the event actually occurred only once in the current clock period.

Assumptions 1, 4, 5 are not very restrictive and essentially represent modelling issues. Assumptions 4, 5 partially deal with timing and communication delay issues.

Note: As we build our work on the SD supervisory control theory, these assumptions apply to our study as well.

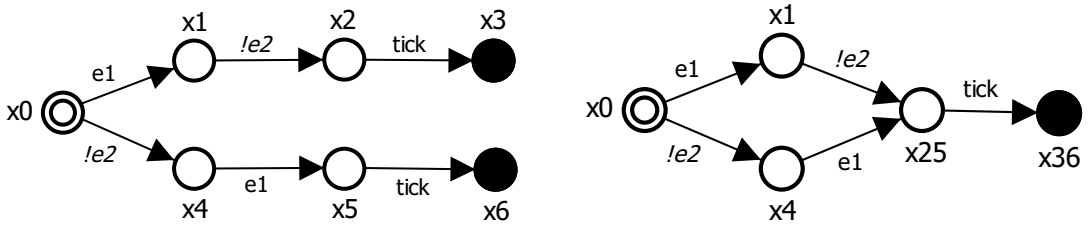
3.4 SD Preliminaries

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor.

An SD controller samples inputs and changes state on the clock edge, which is associated with the *tick* event of the TDES. This means an SD controller can only observe strings ending with a *tick*. Additionally, it can also see the empty string (ϵ) that represents the initial state of the system which is always known. Such strings are referred to as sampled strings.

Definition 3.1. The set of *sampled strings*, L_{samp}, is defined as $L_{samp} = \Sigma^* \,.\, \tau \cup \{\epsilon\}$.

Sampled strings represent observable points in the system. If the controller is implementing TDES supervisor $\mathbf{S}$, states reached from the initial state by sampled strings represent states in $\mathbf{S}$ that are at least partially observable. These states are referred to as sampled states.



(a) Non-CS Deterministic Supervisor **S** (b) CS Deterministic Supervisor **S**′

Figure 11: An Example Illustrating CS Deterministic Supervisor Property

Definition 3.2. For supervisor **S**, the set of *sampled states*, X_{samp}, is defined as:

$$X_{samp} = \{x \in X \,|\, (\exists s \in L(\mathbf{S}) \cap L_{samp})\, x = \xi(x_o, s)\}$$

An SD controller changes state after each clock edge (*tick*). Its next state is determined by all the strings that can occur containing a single *tick* event at the end, since the last *tick* event. Such strings are referred to as concurrent strings.

Definition 3.3. The set of *concurrent strings*, L_{conc}, is defined as $L_{conc} = \Sigma^*_{act} \,.\, \tau \subset L_{samp}$.

Two concurrent strings containing the same events but in different order/number are indistinguishable to an SD controller. An occurrence operator is defined to capture this uncertainity. The occurrence operator takes a string and returns the set of events (*occurrence image*) that make up the string.

Definition 3.4. For $s \in \Sigma^*$, the *occurrence operator*, $Occu : \Sigma^* \to \mathrm{Pwr}(\Sigma)$, is defined as:

$$Occu(s) := \{\sigma \in \Sigma \,|\, s \in \Sigma^*.\sigma.\Sigma^*\}$$

If two concurrent strings with the same occurrence image are possible at a given sampled state and they lead to two different states in **S**, this will make the translation of **S** into an SD controller ambiguous and the translated SD controller non-deterministic. To circumvent this undesirable situation, TDES supervisors are required to be concurrent string deterministic.

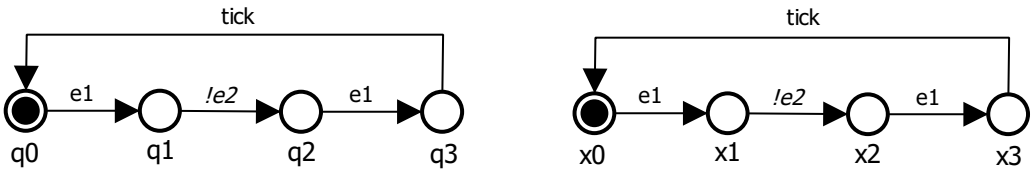
Definition 3.5. A TDES supervisor **S** is *concurrent string (CS) deterministic*, if:

$$(\forall s \in L(\mathbf{S}) \cap L_{samp})\,(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\mathbf{S}) \wedge Occu(s') = Occu(s'')] \Rightarrow$$
$$[ss' \equiv_{L(\mathbf{S})} ss'' \;\wedge\; ss' \equiv_{L_m(\mathbf{S})} ss'' \;\wedge\; \xi(x_o, ss') = \xi(x_o, ss'')]$$

A supervisor **S** failing the CS deterministic property is shown in Figure 11a. In **S**, two concurrent strings, "e1-e2-τ" and "e2-e1-τ", leave the initial state x0. Despite having the same occurrence image of {e1, e2, τ}, they go to two different sampled states, x3 and x6. In this case, we note that **S** fails Definition 3.5 because it is not minimal. For example, states x3 and x6 are λ-equivalent (Definition 2.14) and can be combined together. This is also true for states x2 and x5. After combining these λ-equivalent states, the resulting minimal TDES **S**′ is shown in Figure 11b. Please note that we cannot merge two or more states to obtain a CS deterministic supervisor if they are not λ-equivalent.

One of the assumptions (Point 2 of Section 3.3) says that the controllers allow prohibitable events to occur only once per sampling period. This must be reflected in the TDES plant model and is captured by the following property.

Definition 3.6. For TDES plant **G** and TDES supervisor **S**, **G** is said to have **S**-*singular*



(a) TDES Plant **G** (b) TDES Supervisor **S**

Figure 12: An Example Failing **S**-Singular Prohibitable Behaviour Property

prohibitable behaviour if:

$$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp})\,(\forall s' \in \Sigma^*_{act})\, ss' \in L(\mathbf{S}) \cap L(\mathbf{G}) \Rightarrow$$
$$(\forall \sigma \in Occu(s') \cap \Sigma_{hib})\, \sigma \notin Elig_{L(\mathbf{G})}(ss')$$

An example failing the property of **S**-singular prohibitable behaviour is shown in Figure 12. Plant **G** (Figure 12a) does not have **S**-singular prohibitable behaviour with respect to supervisor **S** (Figure 12b). This is because the prohibitable event e1 is possible twice in the given sampling period in **G**, at state q0 and q2, and this event is also allowed by **S**.

3.5 SD Controllability

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor.

Assume a theoretical system with the following properties: 1) TDES **G** and **S** have finite state spaces and finite event sets, 2) **G** has proper time behaviour and is complete for **S**, 3) **S** is controllable with respect to **G**, 4) **S** is CS deterministic, and 5) $\mathbf{S}\,||\,\mathbf{G}$ is ALF and nonblocking. Even if TDES satisfy the above-mentioned properties, the actual system behaviour under the control of the corresponding SD controller could block, violate the control laws, or exhibit behaviour not contained in **G**. To address these issues and handle the problems discussed in Section 3.2, the property of SD controllability is introduced.

Definition 3.7. TDES supervisor **S** is *SD controllable* with respect to TDES plant **G** if, $\forall s \in L(\mathbf{S}) \cap L(\mathbf{G})$, the following statements are satisfied:

i) $Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\mathbf{S})}(s)$

ii) If $\tau \in Elig_{L(\mathbf{G})}(s)$, then $\tau \in Elig_{L(\mathbf{S})}(s) \Leftrightarrow Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

iii) If $s \in L_{samp}$ then

1) $(\forall s' \in \Sigma^*_{act})\,[ss' \in L(\mathbf{S}) \cap L(\mathbf{G})] \Rightarrow$
$$[Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(ss') \cup Occu(s')] \cap \Sigma_{hib} = Elig_{L(\mathbf{S})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib}$$

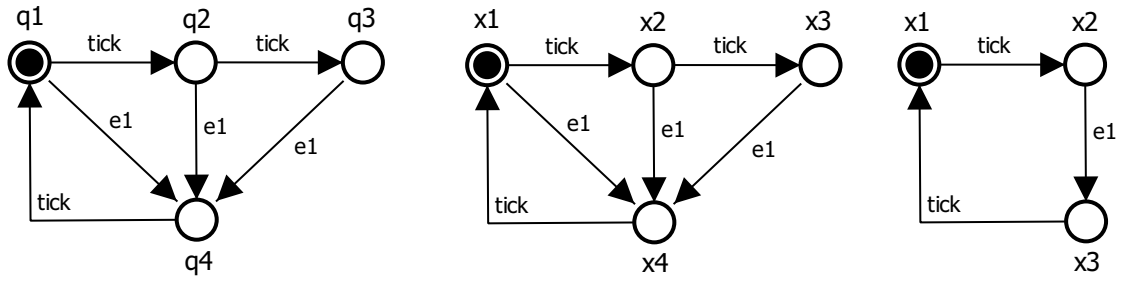
2) $(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\mathbf{S}) \cap L(\mathbf{G}) \wedge Occu(s') = Occu(s'')] \Rightarrow$
$$ss' \equiv_{L(\mathbf{S})\cap L(\mathbf{G})} ss'' \wedge ss' \equiv_{L_m(\mathbf{S})\cap L_m(\mathbf{G})} ss''$$

iv) $L_m(\mathbf{S}) \cap L_m(\mathbf{G}) \subseteq L_{samp}$

We now give a brief explanation for each of these points.

Point i: This is the standard untimed controllability property (Definition 2.20).

Point ii: In the reverse direction ($\Leftarrow$), it says that a *tick* event cannot be disabled unless there exists an eligible prohibitable event to preempt the *tick*. Together with Point i, this implies standard timed controllability (Definition 2.22), since $\Sigma_{for} = \Sigma_{hib}$ in the SD setting.



(a) TDES **G** (b) TDES **Sup1** (c) TDES **Sup2**

Figure 13: An Example Failing SD Controllability Point ii (⇒)

The forward direction (⇒) states that if a prohibitable event is enabled, *tick* must be disabled. This captures the notion that a prohibitable event is enabled only when it needs to be forced, otherwise it must remain disabled. This removes the ambiguity about which sampling period an enabled prohibitable event should be forced in, by making enabling and forcing essentially one and the same. This not only makes the conversion of TDES supervisors into SD controllers simple and straightforward, but also ensures that TDES behaviour is closer to the implementation by removing forcing options that are not actually used in the physical system.

For plant **G** (Figure 13a), supervisor **Sup1** (Figure 13b) does not satisfy Point ii (⇒) as both *tick* and prohibitable event e1 are enabled at states x1 and x2. This creates uncertainty about when event e1 should be forced (at state x1, x2 or x3) and makes the translation of TDES supervisors into SD controllers ambiguous. However, supervisor **Sup2** (Figure 13c) satisfies this property and removes the ambiguous and unused behaviour by allowing *tick* to occur at state x1 and forcing prohibitable event e1 at state x2.

Point iii: For a sampled string s, the following two sub points must be satisfied.

Point iii.1: This point expresses that when a prohibitable event is possible in a clock period, it must be possible immediately after the *tick* and stay possible for the period until it occurs. This captures two ideas: 1) The enablement information of an SD controller is constant for the entire clock period. 2) When a controller forces a prohibitable event, the event must occur before the next *tick*, but we don't know when. So the event must be possible in the plant for the entire clock period till it occurs and must be able to interleave with the other events occurring in the same clock period. Point iii.1 bridges the gap between TDES supervisors and SD controllers by restricting the way the TDES supervisors change their enablement information and makes it consistent with the SD controllers. This property also emphasizes that if two prohibitable events should occur in a specific order, they must be forced in separate clock periods.

Point iii.2: This point states that if two concurrent strings with the same occurrence image are possible after a given sampled string, they must have the same future with respect to the system's closed behaviour (i.e. same control action must be taken now and in the future for both strings), and with respect to its marked behaviour (i.e. the strings are interchangeable with respect to reaching future marked states).

Point iv: All marked strings in the closed-loop system must be sampled strings.

It is worth noting that Point iii and Point iv apply to both **G** and **S**.

3.6 Formal Model of SD Controller

In the SD supervisory control framework, an SD controller is modelled as a Moore synchronous Finite State Machine (FSM) [8]. A *Moore FSM* is a Moore state machine that changes state only on the rising or falling edge of the clock. It chooses its next state based on its current state and inputs. Its outputs are determined by its current state only.

Before giving the formal definition of an SD controller, first we need to introduce some notation.

The inputs and outputs of an SD controller are represented as *boolean vectors*. A boolean vector is a vector whose individual elements can only be assigned the values of *True* (1) or *False* (0). These vectors of information change periodically with respect to some clock.

Let $k \in \{0, 1, 2, \ldots\}$. For any vector $\mathbf{v} = [v_1, v_2, \ldots, v_n] \in V$ or any of its element v_j where $j \in \{1, \ldots, n\}$, "$\mathbf{v}(k)$" and "$v_j(k)$" is used to denote the value of $\mathbf{v}$ and v_j at time k. "At time k'' means that k clock ticks have gone by since the starting reference point, $k = 0$. For $k = 0$, $\mathbf{v}(0)$ represents the initial or starting value of $\mathbf{v}$. $k = 0$ represents the time when an SD controller has just been turned on or reset. As index k takes on new values, vector $\mathbf{v}$ defines a sequence with respect to the clock ticks, which are defined to be $\{\mathbf{v}(k) \mid k = 0, 1, 2 \ldots\}$, and is denoted as $\{\mathbf{v}(k)\}$. A 'clock tick' corresponds to the occurrence of a *tick* event of a TDES.

Definition 3.8. An *SD controller* $\mathbf{C}$ is defined as a 6-tuple, $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$, where:

- I is the set of possible boolean vectors that the *inputs* of the controller can take on. Each vector $\mathbf{i} = [i_0, i_1, \ldots, i_{v-1}] \in I$ has v input variables. Each element of I corresponds to a unique activity event in the system. When an element is set to 1, this means the corresponding event has occurred at least once in the previous clock period, otherwise it is set to 0. Each input vector $\mathbf{i}(k') \in \{\mathbf{i}(k)\}$ is sampled at the occurrence of a *tick* event, except for $k = 0$ which occurs when the controller is turned on.
- Z is the set of possible boolean vectors that the *outputs* of the controller can take on. Each vector $\mathbf{z} = [z_0, z_1, \ldots, z_{r-1}] \in Z$ has r output variables. Each element of Z corresponds to a unique prohibitable event in the system. When an element is set to 1, this means the corresponding event is enabled and the controller should make the event occur before the next clock tick, where 0 means it is disabled. Each output vector $\mathbf{z}(k') \in \{\mathbf{z}(k)\}$ is generated at the occurrence of a *tick* event, except for $k = 0$ which occurs when the controller is turned on.
- Q is the set of possible boolean vectors that the *states* of the controller can take on. Each vector $\mathbf{q} = [q_0, q_1, \ldots, q_{l-1}] \in Q$ has l state variables. Starting at $k = 1$, each state $\mathbf{q}(k') \in \{\mathbf{q}(k)\}$ changes to next state $\mathbf{q}(k' + 1) \in \{\mathbf{q}(k)\}$ at the occurrence of a *tick* event.
- $\Omega : Q \times I \to Q$ is the *next-state* function. It takes the current state $\mathbf{q}(k) \in Q$ and an input vector $\mathbf{i}(k + 1) \in I$, and returns the next state $\mathbf{q}(k + 1) \in Q$ such that $\mathbf{q}(k + 1) = \Omega(\mathbf{q}(k), \mathbf{i}(k + 1))$.
- $\Phi : Q \to Z$ is the *state-to-output* map. For state $\mathbf{q} \in Q$, the output $\mathbf{z} \in Z$ at this state is defined as $\mathbf{z} = \Phi(\mathbf{q})$.
- $\mathbf{q}_{res} \in Q$ is the *initial (reset)* state for when the controller starts operating or is reset. Thus we have $\mathbf{q}(0) = \mathbf{q}_{res}$.

Starting at time $k = 0$, a specific run of the controller would give a specific sequence of inputs $\{\mathbf{i}(k)\}$. This sequence, combined with $\mathbf{q}_{res}$ and Ω, will uniquely define the current sequence of states, $\{\mathbf{q}(k)\}$. In turn, $\{\mathbf{q}(k)\}$ and Φ will uniquely define the current sequence

of outputs, $\{\mathbf{z}(k)\}$. To distinguish between two vector sequences, different variables will be used, e.g. $\{\mathbf{i}(k)\}$ and $\{\mathbf{i}(k')\}$.

3.7 TDES to FSM Translation

In this section, we introduce the TDES to FSM translation method from [43]. We focus on the aspects that are required to comprehend our work presented in the following sections. Please refer to [42, 43] for an in-depth discussion of the complete translation method.

The TDES-FSM translation starts with a CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$. By using the information for $\mathbf{S}$, it constructs the corresponding SD controller $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$.

In order to do the translation, each item in the controller's tuple (i.e. I, Z, Ω, etc.) needs to be defined in terms of TDES $\mathbf{S}$. To do this, the authors have defined several translation functions. These functions capture the next state behaviour and enablement information from $\mathbf{S}$, associate events with elements of input and output vectors, and associate sampled states of $\mathbf{S}$ with states of the controller. They also map event subsets of input or output vectors, as well as define the controller's next state logic (Ω) and state-to-output map (Φ) in terms of supervisor $\mathbf{S}$.

3.7.1 Translation Functions

Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let $\Sigma_{act} \subset \Sigma$ be the set of all activity events, and $\Sigma_{hib} \subseteq \Sigma_{act}$ be the set of all prohibitable events. Let $X_{samp} \subseteq X$ be the set of sampled states of $\mathbf{S}$. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the controller implementation of a CS deterministic supervisor $\mathbf{S}$.

A formal TDES to FSM translation method has been developed by defining several translation functions. These translation functions can be used to take the components of $\mathbf{S}$, and define the components of $\mathbf{C}$. Below, we only list down those functions that we need to prove our equivalence of the SD controllers presented in Section 7.

TDES Mapping Functions The following two functions express the SD behaviour of a TDES.

Definition 3.9. Let $\mathbf{S}$ be a CS deterministic TDES. For $x \in X_{samp}$ and $\Sigma' \subseteq \Sigma_{act}$, the partial function of *next sampling state function*, $\Delta : X_{samp} \times \mathrm{Pwr}(\Sigma_{act}) \to X_{samp}$, is defined as:

$$\Delta(x, \Sigma') := \begin{cases} \xi(x, s) & \text{if } (\exists s \in L_{conc})\, \xi(x, s)! \;\&\; Occu(s) \cap \Sigma_{act} = \Sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

The next sampling state function represents how a TDES will move from one sampled state to the next via concurrent strings.

Definition 3.10. Let TDES supervisor $\mathbf{S}$ be SD controllable with respect to TDES plant $\mathbf{G}$. For $x \in X_{samp}$, the *prohibited action function*, $\zeta : X_{samp} \to \mathrm{Pwr}(\Sigma_{hib})$, is defined as $\zeta(x) := \{\sigma \in \Sigma_{hib} \mid \xi(x, \sigma)!\}$.

This function defines the control action that will take place at a given sampled state x, i.e. it captures the prohibitable events that are enabled at x.

Event Mapping Functions For the following event mapping functions, let $\Sigma_{\mathbf{S}} \subseteq \Sigma$ be the event set of a CS deterministic supervisor $\mathbf{S}$.

Definition 3.11. Let bijective map[7] $\gamma_g : \Sigma_{act} \to \{0, \ldots, |\Sigma_{act}| - 1\}$ be the *canonical event mapping function* such that $(\forall \sigma_1, \sigma_2 \in \Sigma_{act})\, \sigma_1 = \sigma_2 \Leftrightarrow \gamma_g(\sigma_1) = \gamma_g(\sigma_2)$.

Definition 3.12. The *input event mapping function* for $\mathbf{C}$ is a bijective map $\gamma : \Sigma_{\mathbf{S}} \cap \Sigma_{act} \to \{0, 1, \ldots, v-1\}$, where $v = |\Sigma_{\mathbf{S}} \cap \Sigma_{act}|$. It is defined such that:

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{\mathbf{S}} \cap \Sigma_{act})\, \gamma_g(\sigma_1) < \gamma_g(\sigma_2) \Rightarrow \gamma(\sigma_1) < \gamma(\sigma_2)$$

Definition 3.13. The *output event mapping function* for $\mathbf{C}$ is a bijective map $\eta : \Sigma_{\mathbf{S}} \cap \Sigma_{hib} \to \{0, 1, \ldots, r-1\}$, where $r = |\Sigma_{\mathbf{S}} \cap \Sigma_{hib}|$. It is defined such that:

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{\mathbf{S}} \cap \Sigma_{hib})\, \gamma_g(\sigma_1) < \gamma_g(\sigma_2) \Rightarrow \eta(\sigma_1) < \eta(\sigma_2)$$

Controller Functions For the following definitions, let $\mathbf{C}$ be the corresponding controller for CS deterministic supervisor $\mathbf{S}$.

Definition 3.14. Let $\Sigma_{act} \subset \Sigma$ be the set of global activity events. Let $\mathbf{i}_g$ be a single input vector that the system sees, i.e. it is globally available. $\mathbf{i}_g = [i_{g,0}, i_{g,1}, \ldots, i_{g,v_g-1}]$ is required to be defined over Σ_{act}, where $v_g = |\Sigma_{act}|$. That is, for any event $\sigma \in \Sigma_{act}$, there is an element in $\mathbf{i}_g$ that corresponds to σ and only σ. We call $\{\mathbf{i}_g(k)\}$ a *canonical input sequence*, and $\mathbf{i}_g \in \{\mathbf{i}_g(k)\}$ a *canonical input vector*[8].

Definition 3.15. For CS deterministic supervisor $\mathbf{S}$, let $\Lambda : X_{samp} \to Q$ be an arbitrary injective map, where $X_{samp} \subseteq X$. Λ is a *state mapping function* for $\mathbf{C}$ if, for all $x \in X_{samp}$, $\Lambda(x)$ returns a vector of state variables $\mathbf{q} = [q_0, q_1, \ldots, q_{l-1}]$ such that:

$$(\forall x_1, x_2 \in X_{samp})\, \Lambda(x_1) = \Lambda(x_2) \Leftrightarrow x_1 = x_2$$

The initial state is also a sampled state, and is mapped to be $\Lambda(x_o) = \mathbf{q}_{res} = \mathbf{q}(0)$.

Definition 3.16. Let γ be the input event mapping function for $\mathbf{C}$. A bijective map of *input set mapping function* for $\mathbf{C}$, $\Gamma_I : \mathrm{Pwr}(\Sigma_{act}) \to I$, is defined as follows. For arbitrary $\Sigma_I \subseteq \Sigma_{act}$, we have $\Gamma_I(\Sigma_I) = [i_0, i_1, \ldots, i_{v-1}]$ such that for $j = 0, 1, \ldots, v-1$,

$$i_j := \begin{cases} 1 & \text{if } (\exists \sigma \in \Sigma_I)\, \gamma(\sigma) = j \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.17. Let η be the output event mapping function for $\mathbf{C}$. A bijective map of *output set mapping function* for $\mathbf{C}$, $\Gamma_Z : \mathrm{Pwr}(\Sigma_{hib}) \to Z$, is defined as follows. For arbitrary $\Sigma_Z \subseteq \Sigma_{hib}$, we have $\Gamma_Z(\Sigma_Z) = [z_0, z_1, \ldots, z_{r-1}]$ such that for $j = 0, 1, \ldots, r-1$,

$$z_j := \begin{cases} 1 & \text{if } (\exists \sigma \in \Sigma_Z)\, \eta(\sigma) = j \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.18. Let Δ be the next sampling state function for $\mathbf{S}$, and let $X_{samp} \subseteq X$. For state $\mathbf{q} \in Q$ and arbitrary input $\mathbf{i} \in I$, the *next state function* Ω is defined as:

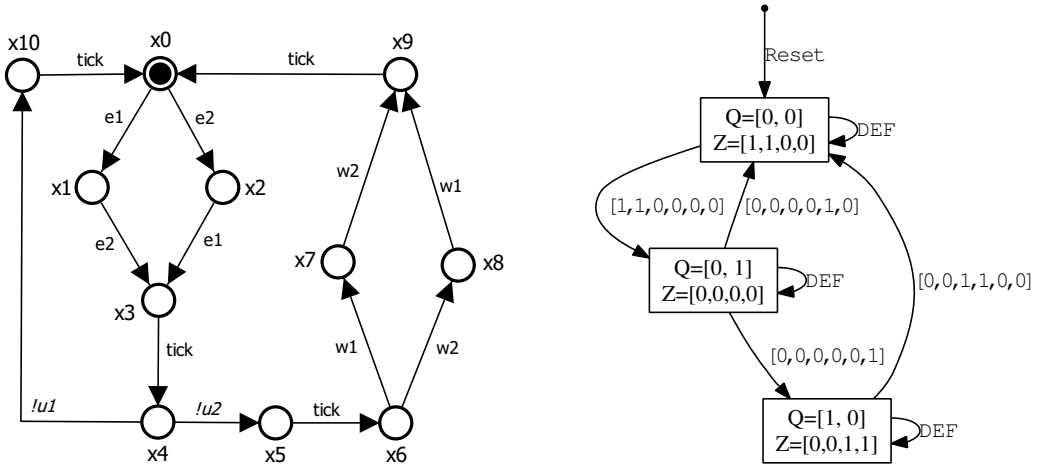
$$\Omega(\mathbf{q}, \mathbf{i}) := \begin{cases} \Lambda(\Delta(x, \Gamma_I^{-1}(\mathbf{i}))) & \text{if } (\exists x \in X_{samp})\, \mathbf{q} = \Lambda(x)\ \&\ \Delta(x, \Gamma_I^{-1}(\mathbf{i}))! \\ \text{arbitrary} & \text{otherwise} \end{cases}$$

Definition 3.19. Let ζ be the prohibited action function for $\mathbf{S}$. For state $\mathbf{q} \in Q$, the *state-to-output map* Φ is defined as:

$$\Phi(\mathbf{q}) := \begin{cases} \Gamma_Z(\zeta(x)) & \text{if } (\exists x \in X_{samp})\, \mathbf{q} = \Lambda(x) \\ \Gamma_Z(\emptyset) & \text{otherwise} \end{cases}$$

[7] See Definition *A*.5 of *bijective function* in Appendix A.

[8] The use of "canonical" here refers to the size and ordering of the inputs, not to the actual values of the input sequence or a given vector.



(a) TDES Supervisor **S** (b) Translated FSM

Figure 14: An Example of TDES to FSM Translation Method *(Reprinted from [42])*

3.7.2 Translation Method

This section defines the TDES to FSM translation method from [42, 43], and provides a simple example.

To translate a TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ into an SD controller $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$, **S** must be CS deterministic to ensure that the resulting SD controller is deterministic. This translation method will not work otherwise. In practice, **S** should preferably be non-selfloop ALF as well. However, this is just a design aid, and not a hard requirement.

In order to construct an SD controller **C**, the values for each member of its tuple need to be defined. To define I, Z and Q, the authors define the size of each vector, as each element will represent a distinct $\sigma \in \Sigma_{act}$, $\sigma \in \Sigma_{hib}$, or state $x \in X_{samp}$ respectively. For each $i \in I$, its size is defined as $v = |\Sigma_{act}|$. For each $z \in Z$, its size is defined as $r = |\Sigma_{hib}|$.

To define the size of Q, the size of each $q \in Q$ needs to be large enough to encode a unique value for each $x \in X_{samp} \subseteq X$. If each state contains l elements, 2 unique values can be expressed. Thus, l is selected such that $2^{l-1} < |X_{samp}| \leq 2^l$.

The mapping functions (given in the previous section) are then used to associate event subsets and sampled states to specific values in I (map Γ_I), Z (map Γ_Z) and Q (map Λ). The initial/reset state is immediately set to $\mathbf{q}_{res} = \Lambda(x_o)$.

Next, Definition 3.18 is used to define the controller's next state function, Ω. It is notable that if the input vector does not represent a concurrent string accepted by **S**, the next state (and thus the resulting logic) is defined arbitrary.

Finally, Definition 3.19 is used to define the controller's state-to-output map, Φ. Please note that if state q does not represent a sampled state (i.e. $|X_{samp}| < 2^l$, and thus have unused states), then all of its outputs are set to $False$ (0).

Informally, the translation process begins by taking the sampled states of **S** as the states of **C**. The initial state of **S** would be the initial (*reset*) state of **C**. Next step is to determine which concurrent strings are possible from a given sampled state. The occurrence image of these concurrent strings would then define the next-state conditions, and the state will be changed accordingly.

As an example, consider the CS deterministic supervisor **S** and its corresponding translated FSM shown in Figure 14. The sampled states of **S** (Figure 14a), x0 (initial state), x4 and x6,

are equated to three states in the FSM (Figure 14b), $\mathbf{q}_{res}$ = x0 = [0, 0], x4 = [0, 1] and x6 = [1, 0]. We assume the ordering I = [e1, e2, w1, w2, u1, u2], and Z = [e1, e2, w1, w2]. As only two prohibitable events, e1 and e2, are possible at state x0 in **S**, only these outputs are set to 1 at state [0, 0] in the FSM. Similarly, all outputs are set to 0 at state [0, 1], and only w1 and w2 outputs are set to 1 at state [1, 0].

Examining state x0, we see that the only concurrent strings leaving it are "e1-e2-τ" and "e2-e1-τ". They have the same occurrence image and both strings take us to the same next state x4 in **S**. Thus, our next-state condition is that only when e1 and e2 have occurred, we go to state [0, 1] in the FSM. Next-state conditions for other sampled states are determined in the similar fashion.

As ξ of **S** is a partial function and Ω of **C** is a total function, a **DEF** (default) transition usually needs to be added to the translated FSM. **DEF** is a shorthand notation to cover input combinations that are not explicitly specified, i.e. it matches all the remaining unspecified input combinations.

3.8 Supervisory Control

The concept of supervisory control V (Definition 2.23) is originally defined in terms of the set of forcible events, Σ_{for}. Since $\Sigma_{for} = \Sigma_{hib}$ in the SD setting, this definition has been expressed with respect to the set of prohibitable events as follows.

Definition 3.20. A *TDES supervisory control* for $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ is a map $V : L(\mathbf{G}) \to \mathrm{Pwr}(\Sigma)$, such that $(\forall s \in L(\mathbf{G}))$,

$$V(s) \supseteq \begin{cases} \Sigma_u \cup (\{\tau\} \cap Elig_{L(\mathbf{G})}(s)) & \text{if } V(s) \cap Elig_{L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ \Sigma_u & \text{if } V(s) \cap Elig_{L(\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$$

Note: In this report, as we will only be dealing with TDES models, therefore we will drop the word "TDES", and will often refer to this property as "V is a supervisory control for **G**".

Definition 3.21. For TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, the *concurrent behaviour* of **G** is defined to be a map $CB_{\mathbf{G}} : L(\mathbf{G}) \cap L_{samp} \to \mathrm{Pwr}(L_{conc})$[9], such that for $s \in L(\mathbf{G}) \cap L_{samp}$,

$$CB_{\mathbf{G}}(s) := \{s' \in L_{conc} \mid ss' \in L(\mathbf{G})\}$$

It says that the possible concurrent behaviour for **G** after sampled string s, is the set of concurrent strings that can extend s to a string in the closed behaviour of **G**.

Since an SD controller only changes state when a *tick* occurs, it is difficult to relate its control action directly to strings. Therefore, a corresponding supervisory control V is constructed to express the enablement information that controller **C** would provide to plant **G**. Precisely, it captures two ideas: 1) Enablement information changes immediately after a *tick* event and then stays constant till the next *tick*. 2) As soon as a prohibitable event is enabled, the controller will force the event to occur before the next *tick*.

Algorithm 1 constructs supervisory control V (Proposition 3.3 (page 33) shows that V is indeed a TDES supervisory control) by keeping track of how controller $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ changes state in response to strings generated by plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$. A brief description of the algorithm, and variables used in the algorithm follows.

- $Pend \subseteq L_{samp} \times Q$: Set of pending string-state pairs, $(s, \mathbf{q})$, to be analyzed, where s is a sampled string in $L(\mathbf{G})$, and $\mathbf{q} \in Q$ is the corresponding state in **C** reached by input

[9] This map is different from the map of Definition 4.1 given in [29] due to error in the original definition.

Algorithm 1 Obtaining V from Controller $\mathbf{C}$, Acting on Plant $\mathbf{G}$

1: **for all** $s \in L(\mathbf{G})$ **do**
2: $\quad V(s) \leftarrow \Sigma_u \cup \{\tau\}$
3: **end for**
4: $Pend \leftarrow \{(\epsilon, \mathbf{q}_{res})\}$
5: **while** $Pend \neq \emptyset$ **do**
6: $\quad (s, \mathbf{q}) \leftarrow$ a member from $Pend$
7: $\quad Pend \leftarrow Pend - \{(s, \mathbf{q})\}$
8: $\quad \mathbf{z} \leftarrow \Phi(\mathbf{q})$
9: $\quad \Sigma_V \leftarrow \Gamma_Z^{-1}(\mathbf{z})$
10: $\quad$ **if** $\Sigma_V \neq \emptyset$ **then**
11: $\quad\quad V(s) \leftarrow (V(s) \cup \Sigma_V) - \{\tau\}$
12: $\quad$ **end if**
13: $\quad$ **for all** $s' \leftarrow \sigma_1\sigma_2 \ldots \sigma_j \in CB_{\mathbf{G}}(s)$ **do** $\quad$ // $\sigma_j = \tau$ by definition
14: $\quad\quad$ **if** $(Occu(s') \cap \Sigma_{hib} \subseteq \Sigma_V) \wedge (ss' \in L(\mathbf{S}))$ **then**
15: $\quad\quad\quad \Sigma_{temp} \leftarrow \Sigma_V$
16: $\quad\quad\quad \mathbf{i} \leftarrow \Gamma_I(Occu(s') - \{\tau\})$
17: $\quad\quad\quad \mathbf{q}' \leftarrow \Omega(\mathbf{q}, \mathbf{i})$
18: $\quad\quad\quad Pend \leftarrow Pend \cup \{(ss', \mathbf{q}')\}$
19: $\quad\quad\quad$ **if** $j > 1$ **then**
20: $\quad\quad\quad\quad$ **for** $i \leftarrow 1$ to $j - 1$ **do**
21: $\quad\quad\quad\quad\quad \Sigma_{temp} \leftarrow \Sigma_{temp} - \sigma_i$
22: $\quad\quad\quad\quad\quad$ **if** $\Sigma_{temp} \neq \emptyset$ **then**
23: $\quad\quad\quad\quad\quad\quad V(s\sigma_1\sigma_2 \ldots \sigma_i) \leftarrow (V(s\sigma_1\sigma_2 \ldots \sigma_i) \cup \Sigma_V) - \{\tau\}$
24: $\quad\quad\quad\quad\quad$ **else**
25: $\quad\quad\quad\quad\quad\quad V(s\sigma_1\sigma_2 \ldots \sigma_i) \leftarrow (V(s\sigma_1\sigma_2 \ldots \sigma_i) \cup \Sigma_V)$
26: $\quad\quad\quad\quad\quad$ **end if**
27: $\quad\quad\quad\quad$ **end for**
28: $\quad\quad\quad$ **end if**
29: $\quad\quad$ **end if**
30: $\quad$ **end for**
31: **end while**
32: **return** V

sequences that would match the concurrent strings that make up s. If $s = \epsilon$, then $\mathbf{q} = \mathbf{q}_{res}$.

- Σ_V: Set of prohibitable events enabled by $V(s)$ for current sampled string s that is being processed.
- Σ_{temp}: Copy of Σ_V that is made while processing a concurrent string that extends sampled string s that is currently being processed. It keeps track of the prohibitable events in Σ_V that have not yet occurred in substrings of the concurrent strings that extend s in $L(\mathbf{G})$.

For all strings $s \in L(\mathbf{G})$, the algorithm starts by adding all uncontrollable events (Σ_u) and *tick* (τ) event to $V(s)$ from **lines 1-3**. This is done to satisfy Definition 3.20 of supervisory control V.

As controller always starts operating at its reset state, $(\epsilon, \mathbf{q}_{res})$ is the 1st string-state pair that is added to $Pend$ at **line 4**. All string-state pairs that get added to $Pend$ during the

execution of the algorithm are extracted and analyzed one by one in the **while-loop** running from **lines 5-31**.

At **lines 6-7**, the next string-state pair to be analyzed, $(s, \mathbf{q})$, is extracted and removed from *Pend*. At **line 8**, for current state $\mathbf{q}$ of $\mathbf{C}$, the output vector $\mathbf{z}$ is obtained by applying the state-to-output map Φ (Definition 3.19). At **line 9**, $\mathbf{z}$ is used to construct Σ_V using the inverse of output set mapping function Γ_Z (Definition 3.17). Σ_V now contains the set of all prohibitable events enabled by $\mathbf{C}$ at state $\mathbf{q}$.

Lines 10-12 process $V(s)$. If any prohibitable event is enabled at state $\mathbf{q}$ (**line 10**), the enablement information Σ_V is added to $V(s)$ for current sampled string s (**line 11**). Also, since a prohibitable event is enabled and needs to be forced, *tick* (added at **line 2**) gets removed from $V(s)$ to satisfy Point ii ($\Rightarrow$) of the SD controllability definition (Definition 3.7).

Lines 13-30 loops through all possible concurrent strings s' that extend s in $L(\mathbf{G})$ ($s' = \sigma_1\sigma_2\ldots\sigma_j \in CB_{\mathbf{G}}(s)$). However, at **line 14**, those concurrent strings whose occurrence images contain prohibitable events that have been disabled by $\mathbf{C}$ at state $\mathbf{q}$ (not in Σ_V) are ignored. **Line 14** also disregards concurrent strings that do not represent a valid behaviour by extending s in $L(\mathbf{S})$, thus restricting the valid strings to $L(\mathbf{S}) \cap L(\mathbf{G})$. As these illegal strings represent behaviour that will not actually happen in the closed-loop system, they are left at their default enablement information (**line 2**).

Line 15 copies Σ_V to Σ_{temp}. Using the occurrence image of concurrent string s', **line 16** computes input vector $\mathbf{i}$ by applying input set mapping function Γ_I (Definition 3.16). At **line 17**, the next-state function Ω (Definition 3.18) is used to compute the next state $\mathbf{q}'$ of $\mathbf{C}$ that is reached from $\mathbf{q}$ by $\mathbf{i}$. This new string-state pair $(ss', \mathbf{q}')$ also needs to be analyzed, so it is added to *Pend* at **line 18**.

Line 19 checks to see if s' contains any activity events (for $j = |s'|$, if $j > 1$). If so, each substring $\sigma_1\sigma_2\ldots\sigma_i$, where $i < j$, is analyzed from **lines 20-27**.

Line 21 potentially removes one prohibitable event from Σ_{temp}. If Σ_{temp} contains more prohibitable events that have not yet occurred (**line 22**), then *tick* is removed from $V(s\sigma_1\sigma_2\ldots\sigma_i)$ to force the remaining enabled prohibitable events in the current sampling period (**line 23**). Otherwise, *tick* event is not removed from $V(s\sigma_1\sigma_2\ldots\sigma_i)$ (**line 25**). Moreover, in both cases, Σ_V is added to $V(s\sigma_1\sigma_2\ldots\sigma_i)$, since the enablement information of $\mathbf{C}$ remains constant until the next *tick*.

It is worth clarifying that this algorithm abstractly describes how map V is related to $\mathbf{C}$. As $L(\mathbf{G})$ may not be finite, there might be infinite number of string-state pairs to analyze, and the algorithm may never terminate. In [42], the authors have proven that map V constructed from $\mathbf{C}$ using this algorithm is well defined.

Definition 3.22. For plant $\mathbf{G}$, and CS deterministic supervisor $\mathbf{S}$ that is SD controllable for $\mathbf{G}$, let $\mathbf{C}$ be the SD controller that is constructed from $\mathbf{S}$ using the translation method described in Section 3.7, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 1. The *marked behaviour* of $V/\mathbf{G}$ is defined as $L_m(V/\mathbf{G}) := L(V/\mathbf{G}) \cap L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

Definition 3.23. V is said to be *nonblocking for* $\mathbf{G}$ if $\overline{L_m(V/\mathbf{G})} = L(V/\mathbf{G})$.

3.9 Verification Results

Comprehensive theoretical proofs and results for verifying the control action of an SD controller and comparing it to that of the TDES supervisor from which it was converted are presented in [42, 29]. In this section, we only outline some significant conclusions, and restate those

theorems/propositions that we will refer to in our work presented in the subsequent sections.

For TDES plant $\mathbf{G}$, TDES supervisor $\mathbf{S}$ and an SD controller $\mathbf{C}$, the system is required to satisfy the following properties: 1) $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces and finite event sets, 2) $\mathbf{G}$ has proper time behaviour, 3) $\mathbf{G}$ is complete for $\mathbf{S}$, 4) $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour, 5) $\mathbf{S}\,||\,\mathbf{G}$ is ALF, 6) $\mathbf{S}$ is SD controllable with respect to $\mathbf{G}$, 7) $\mathbf{S}$ is CS deterministic, and 8) $\mathbf{C}$ is an SD controller translated from $\mathbf{S}$ as described in Section 3.7. Given that these conditions are met, the following results have been proven for the SD supervisory control methodology.

3.9.1 SD Controller as a Supervisory Control

To compare the control action of $\mathbf{S}$ and $\mathbf{C}$, a supervisory control V is constructed using Algorithm 1. It is demonstrated in [42] that V is indeed a map that expresses the enablement and forcing behaviour of $\mathbf{C}$.

Proposition 3.1 given below is taken from [29]. Although the control action of $\mathbf{C}$ could be quite different than that of $\mathbf{S}$, this proposition proves that if any string is not accepted by $\mathbf{S}$, it will also be rejected by $\mathbf{C}$, i.e. if a certain path is not possible in the theoretical model, it can never occur in the implemented system, thus preventing the physical system to behave in an undesirable and unexpected way.

Proposition 3.1. [29] For plant $\mathbf{G}$ and supervisor $\mathbf{S}$, let $\mathbf{S}$ be CS deterministic and SD controllable for $\mathbf{G}$, and let $\mathbf{G}$ be complete for $\mathbf{S}$, and have $\mathbf{S}$-singular prohibitable behaviour. Let $\mathbf{C}$ be the SD controller that is constructed from $\mathbf{S}$.
$(\forall s \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp})\,(\forall s' \in CB_{\mathbf{G}}(s))$
If s takes $\mathbf{C}$ to state $\mathbf{q}$ and $ss' \notin L(\mathbf{S})$ then $\mathbf{C}$ will reject s'.

3.9.2 Controllability

Using Proposition 3.2, Theorem 3.1 given below proves that the closed-loop behaviour of $\mathbf{G}$ under the control of $\mathbf{C}$ (represented as $L(V/\mathbf{G})$) is same as the closed-loop behaviour of $\mathbf{S}$ and $\mathbf{G}$. This is despite the fact that $\mathbf{S}$ can change its enablement and forcing information at any time, as opposed to $\mathbf{C}$ that is restricted to do so only on the clock edge and then it must keep it constant during the entire clock period. This shows that SD controllers can be used to implement TDES supervisors and obtain the expected closed-loop behaviour, at least with respect to the required enablement and forcing actions of the controller.

Proposition 3.2. [29] For CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$.
$(\forall s \in L(\mathbf{S}) \cap L_{samp})$
String s will take $\mathbf{C}$ to state $\mathbf{q} = \Lambda(\xi(x_o, s))$ with outputs $\sigma \in \Sigma_{\mathbf{q}} = Elig_{L(\mathbf{S})} \cap \Sigma_{hib}$ set to *true*.

Theorem 3.1. [29] For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behaviour, let $\mathbf{S}\,||\,\mathbf{G}$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 1. Then, $L(V/\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$.

By proving the following proposition, it has been demonstrated that map V is indeed a TDES supervisory control for $\mathbf{G}$.

Proposition 3.3. [29] For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behaviour, let $\mathbf{S}||\mathbf{G}$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 1. Then map V is a TDES supervisory control for $\mathbf{G}$.

3.9.3 Event Generation

Theorem 3.2 has been proven in [29] to show that $\mathbf{C}$ cannot generate a prohibitable event when $\mathbf{G}$ won't accept it. This result guarantees that illegal transitions won't occur, thus preventing the system from violating control laws. It also means that $\mathbf{G}$ will accurately reflect the system's behaviour when controlled by $\mathbf{C}$.

Theorem 3.2. [29] For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behaviour, let $\mathbf{S}||\mathbf{G}$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$, and let V be the map that is constructed from $\mathbf{C}$ using Algorithm 1.
$(\forall s \in L(V/\mathbf{G}) \cap L_{samp})\,(\forall s' \in \Sigma^*_{act})\,(\forall \sigma \in \Sigma_{hib})$
If $ss' \in L(V/\mathbf{G})$ and σ then physically occurs after ss' and before any other events can occur, then $ss'\sigma \in L(\mathbf{G})$.

3.9.4 Nonblocking

Before discussing the nonblocking verification results, the following concept has been introduced in the SD setting.

Definition 3.24. Let $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a TDES plant, and let V and V' be supervisory controls for $\mathbf{G}$. V' is said to be *concurrent supervisory control equivalent (CSCE)* to V if:

1. $(\forall s \in L(\mathbf{G}))\, V'(s) \subseteq V(s)$
2. $(\forall s \in L(V'/\mathbf{G}) \cap L_{samp})\,(\forall s' \in L_{conc})\, ss' \in L(V/\mathbf{G}) \Rightarrow$
$(\exists s'' \in L_{conc})\, ss'' \in L(V'/\mathbf{G}) \wedge Occu(s') = Occu(s'')$

Point 1 requires that each event allowed by $V'(s)$ is also allowed by $V(s)$. This is to ensure that $L(V'/\mathbf{G})$ does not include any unwanted behaviour. Point 2 requires that if $V'/\mathbf{G}$ accepts a sampled string s, and $V/\mathbf{G}$ accepts a concurrent string s' after s, then $V'/\mathbf{G}$ must accept a concurrent string s'' that has the same occurrence image as s'.

In the SD setting, the following theorem has been proven to show that $\mathbf{G}$ under the control of $\mathbf{C}$ is nonblocking if and only if $\mathbf{S}\,||\,\mathbf{G}$ is nonblocking. This is true even if only a single concurrent string, out of multiple possible concurrent strings with the same occurrence image possible in the TDES model at a given sampled state, is actually possible in the physical system. The SD approach has been proven to be robust with respect to such variations and nonblocking.

Theorem 3.3. [29] For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete for $\mathbf{S}$, have proper time and $\mathbf{S}$-singular prohibitable behaviour, let $\mathbf{S}||\mathbf{G}$ be ALF, let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller that is constructed from $\mathbf{S}$, and let V be the

map that is constructed from $\mathbf{C}$ using Algorithm 1. Let V' be a supervisory control for $\mathbf{G}$. If V is nonblocking for $\mathbf{G}$ and V' is CSCE to V, then V' is also nonblocking for $\mathbf{G}$.

4 Sampled-Data Synchronous Product

In this section, we present a novel mechanism for constructing closed-loop system in the SD supervisory control framework. Specifically, we devise a new synchronization operator, called the *sampled-data (SD) synchronous product*, to combine TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ to form the closed-loop system. After defining our SD synchronous product operator, we discuss and prove the relevant fundamental properties of this synchronization operator. This is followed by a description of our SD synchronous product setting.

As we are proposing a new way of constructing the closed-loop system, existing properties of the SD supervisory control theory need to be adapted to work with our new synchronization operator. The rest of this section focuses on adapting these properties to make them compatible with our SD synchronous product setting. Finally, this section finishes off with some useful results about the activity-loop-free property (Definition 2.30) with respect to our SD synchronous product setting.

4.1 SD Synchronous Product Operator

In this section, we define our new synchronization operator, called the *sampled-data (SD) synchronous product*, represented as $||_{SD}$, to combine two TDES models. This operator is specifically designed to synchronize TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ in order to construct a closed-loop system in the SD supervisory control framework, and to address the issues discussed in Section 1.2.

The SD synchronous product operator is basically an intelligent and powerful version of the standard synchronous product operator. It is smart enough to automatically disable a *tick* event in the closed-loop system, if both *tick* and prohibitable events are possible in $\mathbf{G}$ and enabled by $\mathbf{S}$.

This implies that in the presence of the SD synchronous product operator, while designing the system, designers no longer need to keep track of the enablement/disable-ment of *tick* event and prohibitable events, and incorporate this logic of explicit *tick* disablement manually in various modular TDES supervisors. This also means that while verifying the system model, the property of SD controllability Point ii ($\Rightarrow$) no longer needs to be explicitly checked, as the SD synchronous product operator guarantees that this property will always be satisfied at every state of the closed-loop system (we elaborate this point later in Section 4.5).

Definition 4.1. Let TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$, for $i = 1, 2$. The *sampled-data (SD) synchronous product* of two TDES, represented as $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$, is defined as:

$$\mathbf{G} := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$$

where $\delta((q_1, q_2), \sigma)$, for $(q_1, q_2) \in Q_1 \times Q_2$ and $\sigma \in \Sigma_1 \cup \Sigma_2$, is only defined and equals:

i) (q_1', q_2') if $\sigma \in (\Sigma_1 \cap \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1' \ \wedge \ \delta_2(q_2, \sigma) = q_2' \ \wedge$
$[(\sigma \neq \tau) \ \vee \ ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib}) \neg \delta((q_1, q_2), \sigma')!)]$

ii) (q_1', q_2) if $\sigma \in (\Sigma_1 - \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1'$

iii) (q_1, q_2') if $\sigma \in (\Sigma_2 - \Sigma_1) \ \wedge \ \delta_2(q_2, \sigma) = q_2'$

Note: From now on, we will refer to this synchronization operator by interchangeably using its name "SD synchronous product" and its symbol "$||_{SD}$" (to be concise).

We will now explain the logic used by the SD synchronous product operator to construct the transition function δ, as this is the only element where the logic of the SD synchronous product differs from the standard synchronous product operator.

The $||_{SD}$ operator constructs the transition function δ of $\mathbf{G}$ based on the component transition functions, δ_1 of $\mathbf{G}_1$ and δ_2 of $\mathbf{G}_2$. As δ_1 and δ_2 are partial functions, the transition function δ constructed by $||_{SD}$ is a partial function as well.

The $||_{SD}$ operator states three rules to define δ. For every state (q_1, q_2) of $\mathbf{G}$ and each $\sigma \in \Sigma_1 \cup \Sigma_2$, these rules are used to determine: (I) If σ transition would be defined at state (q_1, q_2) in $\mathbf{G}$? (II) If so, what would be the destination state that σ would take $\mathbf{G}$ to? These three rules defined by $||_{SD}$ to construct δ are elaborated next.

i) **Point i** applies to events that $\mathbf{G}_1$ and $\mathbf{G}_2$ have in common. This point makes a distinction between the *tick* and non-*tick* (activity) events, and specifies two different rules for defining the *tick* and activity event transitions in $\mathbf{G}$.

a) $\sigma \neq \tau$

For an activity event σ, a transition will be defined at a state in $\mathbf{G}$ if it is defined at the corresponding states in both $\mathbf{G}_1$ and $\mathbf{G}_2$. This means that $\mathbf{G}_1$ and $\mathbf{G}_2$ act together to cooperatively determine and agree on the definition of σ transition, and its corresponding destination state in $\mathbf{G}$. This is essentially the same logic that synchronous product uses to determine its transitions.

It is important to clarify here that the $||_{SD}$ operator is not capable of adding any non-*tick* transition to δ if it does not exist in either δ_1 or δ_2 or both. Likewise, $||_{SD}$ cannot remove a non-*tick* transition from δ if it is defined in both δ_1 and δ_2.

b) $\sigma = \tau$

This is the case where the logic of $||_{SD}$ operator differs from the standard synchronous product, i.e. the case of figuring out the definition of *tick* transitions in $\mathbf{G}$. This point says that a *tick* transition will be defined at a state in $\mathbf{G}$ if the following conditions are satisfied:

I) *tick* transition is defined at the corresponding states in both $\mathbf{G}_1$ and $\mathbf{G}_2$.

II) No prohibitable event is possible at the current state in $\mathbf{G}$.

Point I signifies that $||_{SD}$ will not define a *tick* transition in $\mathbf{G}$ if it is blocked by either $\mathbf{G}_1$ or $\mathbf{G}_2$ or both. This means that our synchronization operator is not capable of adding any *tick* transition to δ on its own. It will 'potentially' add a *tick* transition to δ only if it exists in both δ_1 and δ_2.

However, **Point II** imposes an important condition, which if not satisfied, then $||_{SD}$ operator is capable of deciding "not" to add a *tick* transition to δ, even if is defined in both δ_1 and δ_2. In this case, $||_{SD}$ is smart enough to automatically "disable" a *tick* event if a prohibitable event is currently possible in $\mathbf{G}$.

This means that if *tick* is defined in $\mathbf{G}_1$ and $\mathbf{G}_2$, $||_{SD}$ will not immediately add this *tick* transition to $\mathbf{G}$. First, it will figure out whether or not any prohibitable event σ' is currently possible in $\mathbf{G}$. To determine this, $||_{SD}$ evaluates the transitions for all prohibitable events one by one at the current state in $\mathbf{G}$. Depending upon whether σ' is in $(\Sigma_1 \cap \Sigma_2)$, $(\Sigma_1 - \Sigma_2)$, or $(\Sigma_2 - \Sigma_1)$, the $||_{SD}$ operator will recursively make use of Points i ($\sigma \neq \tau$), ii or iii respectively to figure out if σ' is defined at the corresponding states in both $\mathbf{G}_1$ and

$\mathbf{G}_2$.

If any prohibitable event transition is possible at the current state in $\mathbf{G}$, **Point II** fails, and $||_{SD}$ will "not" add *tick* transition to $\mathbf{G}$. In this way, the $||_{SD}$ operator disables the *tick* event to automatically satisfy Point ii ($\Rightarrow$) of the SD controllability definition (Definition 3.7) at every state of $\mathbf{G}$. On the other hand, if none of the prohibitable events is currently possible in $\mathbf{G}$, **Point II** is satisfied, and $||_{SD}$ will define a *tick* transition in $\mathbf{G}$, given that *tick* is currently possible in both $\mathbf{G}_1$ and $\mathbf{G}_2$.

Please note that since the $||_{SD}$ operator only deals with TDES models, *tick* event will certainly be present in both $\mathbf{G}_1$ and $\mathbf{G}_2$. Thus, $||_{SD}$ will always use this point (and never Point i ($\sigma \neq \tau$), Point ii or Point iii) to determine the definition of *tick* transition and its corresponding next state in $\mathbf{G}$.

Points ii and **iii** are applicable to events that are present in the event set of only one TDES, $\mathbf{G}_1$ or $\mathbf{G}_2$, respectively. These points of the SD synchronous product's definition are identical to the synchronous product's definition.

ii) If an event σ is only in the event set of $\mathbf{G}_1$, then $||_{SD}$ will use **Point ii** to determine the definition and next state of σ transition in $\mathbf{G}$. At a given state, σ will be allowed to occur in $\mathbf{G}$ if it is possible at the corresponding state in $\mathbf{G}_1$. As $\mathbf{G}_2$ does not care about σ, it can neither prevent σ from occurring in $\mathbf{G}$, nor it will change its state as a result of this σ transition.

iii) **Point iii** applies to an event σ that is present only in the event set of $\mathbf{G}_2$. This point says that σ transition will be defined at a state in $\mathbf{G}$ if it is possible at the corresponding state in $\mathbf{G}_2$. $\mathbf{G}_1$ is not related to σ in any way, therefore it cannot block σ transition in $\mathbf{G}$. Also, the occurrence of σ transition will not have any affect on $\mathbf{G}_1$'s current state.

Example Figure 15 shows an example of the $||_{SD}$ operator, and compares its synchronization mechanism to that of the synchronous product operator. In the example, we have two TDES, $\mathbf{G}_1$ (Figure 15a) and $\mathbf{G}_2$ (Figure 15b), that are defined over the same event set Σ, such that $\Sigma = \{e1, e2, \tau\}$, $\Sigma_{hib} = \{e1\}$ and $\Sigma_u = \{e2\}$. At the initial state, q0 of $\mathbf{G}_1$ and x0 of $\mathbf{G}_2$, both *tick* and prohibitable event e1 are defined.

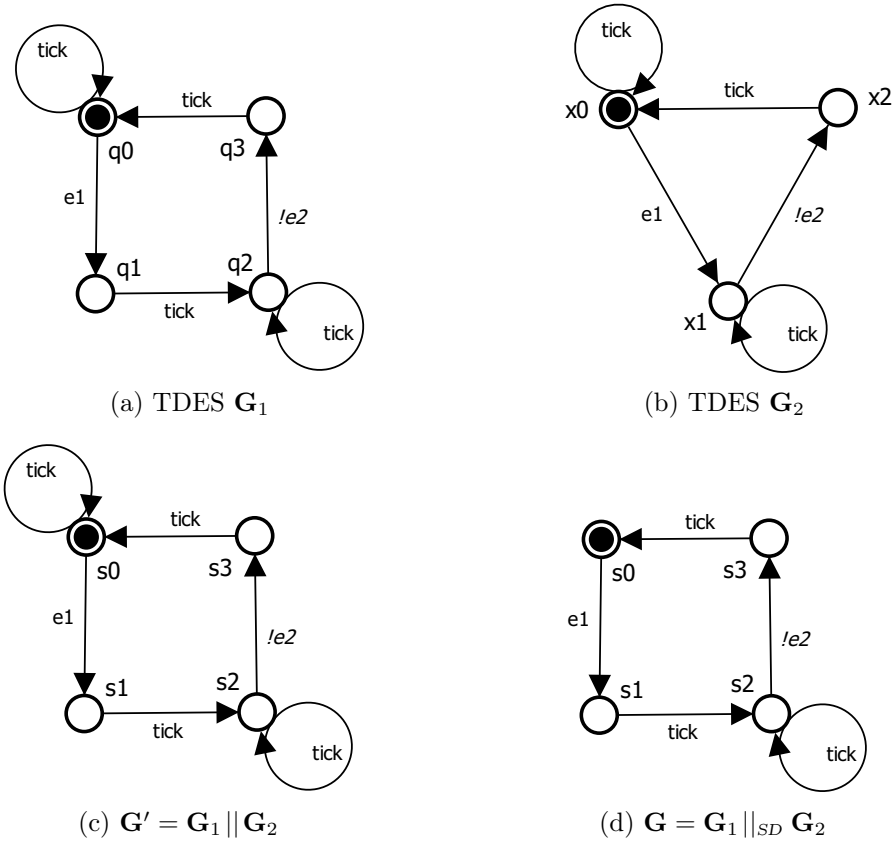
If we construct $\mathbf{G}' = \mathbf{G}_1 \,||\, \mathbf{G}_2$, the synchronous product operator enables both *tick* and prohibitable event e1 at the initial state of $\mathbf{G}'$, as shown in Figure 15c.

Figure 15d illustrates the result of synchronizing $\mathbf{G}_1$ and $\mathbf{G}_2$ using the $||_{SD}$ operator. For $\mathbf{G} = \mathbf{G}_1 ||_{SD} \mathbf{G}_2$, we note that *tick* transition is not defined at the initial state s0, although it is defined at the initial states of $\mathbf{G}_1$ and $\mathbf{G}_2$. The reason is that both $\mathbf{G}_1$ and $\mathbf{G}_2$ have enabled prohibitable event e1 at their initial states. Therefore, the $||_{SD}$ operator enables prohibitable event e1 at the initial state of $\mathbf{G}$ and disables the *tick* event, as desired to satisfy Point ii ($\Rightarrow$) of the SD controllability property.

We also note that this is the only difference between $\mathbf{G}'$ and $\mathbf{G}$, indicating that the rest of the synchronization mechanism of the $||_{SD}$ operator is essentially the same as the synchronous product.

4.2 Properties of SD Synchronous Product Operator

In this section, we discuss and prove some fundamental properties of our SD synchronous product operator. We will start by showing that when we synchronize two TDES automata using the $||_{SD}$ operator, this will result in the generation of a model that is also a TDES automaton.



(a) TDES $\mathbf{G}_1$ (b) TDES $\mathbf{G}_2$

(c) $\mathbf{G}' = \mathbf{G}_1 \,||\, \mathbf{G}_2$ (d) $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$

Figure 15: An Example of SD Synchronous Product Operator

For any synchronization operator, the two key properties of interest are *commutativity* and *associativity*. We will also examine our $||_{SD}$ operator with respect to these properties. Precisely, we demonstrate that the $||_{SD}$ operator is commutative, but not associative. These results will later help us in defining our strategy of constructing the closed-loop system using the $||_{SD}$ operator in our setting, described in Section 4.3.

4.2.1 SD Synchronous Product Defines a TDES

As we have defined a new synchronization operator to combine two TDES automata, it is important to show that the resultant model is also a TDES automaton with all of its elements being well defined. We formally prove this in the following proposition. As the SD synchronous product operator is an adapted version of the standard synchronous product, we will base our argument on the fact that the model generated by the synchronous product operator has these properties.

Proposition 4.1. Let $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{o,2}, Q_{m,2})$ be two TDES. The SD synchronous product of $\mathbf{G}_1$ and $\mathbf{G}_2$, represented as $\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$, defines a TDES automaton.

Proof. The SD synchronous product defines $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$ as a quintuple:

$$\mathbf{G} := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$$

By Definition 2.21, a TDES automaton is formally represented as a quintuple $(Q, \Sigma, \delta, q_o, Q_m)$.

In order to prove that **G** is a TDES automaton, it is sufficient to show that **G**'s tuple is comprised of the five standard elements of a TDES automaton's tuple.

By looking at Definition 4.1 of the $||_{SD}$ operator, it is obvious that the tuple elements of Q, Σ, q_o and Q_m are defined by the $||_{SD}$ operator in exactly the same way as the synchronous product. Clearly these elements of **G** are well defined, as we know that the synchronous product operator is well defined.

Below, we analyze the transition function δ to show that δ defined by $||_{SD}$ is well defined. We will base our argument on the fact that the transition function defined by the synchronous product is well defined.

In order to show that δ is well defined, we need to show that δ unambiguously determines: **(I)** if $\sigma \in (\Sigma_1 \cup \Sigma_2)$ transition would be defined at state $(q_1, q_2) \in Q_1 \times Q_2$ in **G**? **(II)** what would be the destination state for each σ transition that would be defined in **G**?

I) By looking at the definition of δ in the SD synchronous product's definition, we note that **Point ii** and **Point iii** are identical to the synchronous product's transition function. As the synchronous product's transition function is well defined, we deduce that Point ii and Point iii construct δ in a well defined way.

The only rule that makes δ different from the synchronous product's transition function is **Point i**. Therefore, it is sufficient to show that Point i constructs δ in a well defined way.

Depending upon whether an event σ is a *tick* or a non-*tick* (activity) event, Point i specifies two different rules for determining whether or not σ transition would be defined at state (q_1, q_2) in **G**. Thus, we have two cases: **(a)** $\sigma \neq \tau$, and **(b)** $\sigma = \tau$.

In order to show that Point i constructs δ in a well defined, we need to show that δ is constructed in a well defined way in both cases.

Case a) $\sigma \neq \tau$

For an activity event σ, it is decided whether or not $\delta((q_1, q_2), \sigma)!$ in **G**, by evaluating whether or not $\delta_1(q_1, \sigma)!$ and $\delta_2(q_2, \sigma)!$.

This is the same logic that is used by the synchronous product's transition function to figure out its transitions for shared events while synchronizing two TDES models.

As the transition function of synchronous product is well defined, we conclude that for each $\sigma \neq \tau$, δ is well defined in the way it decides whether or not $\delta((q_1, q_2), \sigma)!$ in **G**.

Case (a) complete.

Case b) $\sigma = \tau$

In order to decide whether or not $\delta((q_1, q_2), \tau)!$ in **G**, it is evaluated whether or not: **(1)** $\delta_1(q_1, \tau)!$, **(2)** $\delta_2(q_2, \tau)!$, and **(3)** $(\forall \sigma' \in \Sigma_{hib})\, \delta((q_1, q_2), \sigma')!$.

The process of determining if $\delta_1(q_1, \tau)!$ in $\mathbf{G}_1$ and $\delta_2(q_2, \tau)!$ in $\mathbf{G}_2$ is straightforward and will always give a unique result without any ambiguity, since δ_1 and δ_2 are individually well defined.

In order to figure out whether or not, for all $\sigma' \in \Sigma_{hib}, \delta((q_1, q_2), \sigma')!$, it is examined whether or not for each individual σ', $\delta((q_1, q_2), \sigma')!$.

For individual σ', depending upon whether $\sigma' \in (\Sigma_1 \cap \Sigma_2)$, $\sigma' \in (\Sigma_1 - \Sigma_2)$ or $\sigma' \in (\Sigma_2 - \Sigma_1)$, Point i ($\sigma \neq \tau$), Point ii or Point iii will respectively be used to determine whether or not $\delta_1(q_1, \sigma')!$ and/or $\delta_2(q_2, \sigma')!$. Since we have already shown that Point i ($\sigma \neq \tau$), Point ii and Point iii construct δ in a well defined way, we infer that for each individual $\sigma' \in \Sigma_{hib}$,

the process of determining whether or not $\delta((q_1, q_2), \sigma')!$ is well defined.

This implies that the overall process of determining whether or not, for all $\sigma' \in \Sigma_{hib}$, $\delta((q_1, q_2), \sigma')!$ will always give a unique result without any ambiguity.

Hence, we conclude that the overall decision process of δ to determine whether or not $\delta((q_1, q_2), \tau)!$ in $\mathbf{G}$ is well defined.

Case (b) complete.

By Cases (a) and (b), we conclude that **Point i** constructs δ in a well defined way.

As Points (i-iii) of $||_{SD}$ construct δ in a well defined way, hence we conclude that δ is well defined in the way it determines if σ transition would be defined at state (q_1, q_2) in $\mathbf{G}$.

Part (I) complete.

II) By looking at the definition of δ in $||_{SD}$, we note that δ uses the same strategy as the synchronous product's transition function to determine the destination state of each σ transition that would be defined in $\mathbf{G}$. Since the transition function of synchronous product is well defined, we deduce that δ is also well defined in this perspective.

Part (II) complete.

By Parts (I) and (II), we conclude that the transition function δ, defined by $||_{SD}$, is well defined.

We have thus shown that $\mathbf{G}$'s quintuple defined by $||_{SD}$ comprises of five standard elements of a TDES automaton's tuple and all these elements are well defined.

Hence, we conclude that $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$ defines a TDES automaton. □

4.2.2 Commutative Property

The SD synchronous product operator is commutative up to isomorphism, i.e. $\mathbf{G}_1 ||_{SD} \mathbf{G}_2$ and $\mathbf{G}_2 ||_{SD} \mathbf{G}_1$ will give us the same resultant TDES automaton up to relabelling of state components in the composed states. More formally, TDES $\mathbf{G}$ and $\mathbf{G}'$ are isomorphic up to state relabelling if we can define a bijective function[10] that maps $\mathbf{G}$ to $\mathbf{G}'$. Our next proposition formally proves this concept and property.

Proposition 4.2. Let $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{o,2}, Q_{m,2})$ be two TDES. The SD synchronous product of $\mathbf{G}_1$ and $\mathbf{G}_2$ is commutative up to isomorphism.

Proof. Let $\mathbf{G}$ be a TDES constructed as $\mathbf{G} = \mathbf{G}_1 ||_{SD} \mathbf{G}_2$, and let $\mathbf{G}'$ be a TDES constructed as $\mathbf{G}' = \mathbf{G}_2 ||_{SD} \mathbf{G}_1$.

The SD synchronous product operator defines $\mathbf{G}$ and $\mathbf{G}'$ as follows:

$\mathbf{G} := (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$

$\mathbf{G}' := (Q_2 \times Q_1, \Sigma_2 \cup \Sigma_1, \delta', (q_{o,2}, q_{o,1}), Q_{m,2} \times Q_{m,1})$

First, we note that $||_{SD}$ defines the event sets of $\mathbf{G}$ and $\mathbf{G}'$ as $\Sigma_1 \cup \Sigma_2$ and $\Sigma_2 \cup \Sigma_1$ respectively.

The *commutative property for set union* says the order of sets in which we do the union operation does not change the result. This means taking the union of sets Σ_1 and Σ_2 in either order will give the same resulting set, i.e. $\Sigma = \Sigma_1 \cup \Sigma_2 = \Sigma_2 \cup \Sigma_1$.

This implies that both $\mathbf{G}$ and $\mathbf{G}'$ are defined over the same event set Σ. Therefore, the quintuples of TDES automata $\mathbf{G}$ and $\mathbf{G}'$ can be restated as follows:

[10] See Definition *A*.5 of *bijective function* in Appendix A.

$\mathbf{G} := (Q_1 \times Q_2, \Sigma, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$
$\mathbf{G}' := (Q_2 \times Q_1, \Sigma, \delta', (q_{o,2}, q_{o,1}), Q_{m,2} \times Q_{m,1})$

In order to show that the SD synchronous product of $\mathbf{G}_1$ and $\mathbf{G}_2$ is commutative up to isomorphism, it is sufficient to show that $\mathbf{G}$ and $\mathbf{G}'$ are isomorphic up to state relabelling.

By definition, $\mathbf{G}$ and $\mathbf{G}'$ are said to be isomorphic by states if there exists an isomorphic function, $\boldsymbol{iso}$, that maps $\mathbf{G}$ to $\mathbf{G}'$ while preserving all automata-theoretic structure of $\mathbf{G}$ and $\mathbf{G}'$, as defined by $||_{SD}$, up to relabelling of states.

We will show this first by defining a function $\boldsymbol{iso}$, and proving that $\boldsymbol{iso}$ is indeed an isomorphic map. Then we will show that $\boldsymbol{iso}$ maps $\mathbf{G}$ to $\mathbf{G}'$ while preserving all automata-theoretic structure of $\mathbf{G}$ and $\mathbf{G}'$ up to state relabelling.

First, we will define and construct our function $\boldsymbol{iso}$.

We define $\boldsymbol{iso}$ as: $\boldsymbol{iso} : \mathbf{G} \to \mathbf{G}'$

Our goal is to define $\boldsymbol{iso}$ so we achieve the following result:

$$\boldsymbol{iso}((Q_1 \times Q_2, \Sigma, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})) = (Q_2 \times Q_1, \Sigma, \delta', (q_{o,2}, q_{o,1}), Q_{m,2} \times Q_{m,1})$$

To construct our function $\boldsymbol{iso}$, we define two functions: **(i)** iso_Q, and **(ii)** id_Σ.

i) $iso_Q : Q_1 \times Q_2 \to Q_2 \times Q_1 : (q_1, q_2) \mapsto (q_2, q_1)$

The function iso_Q is defined to map the state set of $\mathbf{G}$ to the state set of $\mathbf{G}'$. Specifically, it takes a given state of $\mathbf{G}$ and maps it to its corresponding state in $\mathbf{G}'$ by swapping the elements of $\mathbf{G}$'s state tuple.

$$(\forall (q_1, q_2) \in Q_1 \times Q_2)\ iso_Q\left((q_1, q_2)\right) = (q_2, q_1)$$

Clearly, iso_Q is bijective as $Q_1 \times Q_2$ and $Q_2 \times Q_1$ are the same size, and:

$$(\forall (q_2, q_1) \in Q_2 \times Q_1)\, iso_Q^{-1}((q_2, q_1)) = (q_1, q_2)$$

ii) $id_\Sigma : \Sigma \to \Sigma : \sigma \mapsto \sigma$

id_Σ is defined as an identity function on Σ. Since both $\mathbf{G}$ and $\mathbf{G}'$ are defined over the same event set Σ, this function maps the event set of $\mathbf{G}$ to the event set of $\mathbf{G}'$ by mapping event σ to itself.

$$(\forall \sigma \in \Sigma)\ id_\Sigma\left(\sigma\right) = \sigma$$

Clearly, id_Σ is bijective as it is an identity function.

Using these two functions, we can map each element of $\mathbf{G}$'s quintuple to its corresponding element in $\mathbf{G}'$'s quintuple, as elaborated next.

We first note that for function $f : x \to y$, we can define for $A \subseteq X$, $f(A) = \{f(x) \,|\, x \in A\}$.

1) State Set

We will use $iso_Q(Q_1 \times Q_2) = \{iso_Q((q_1, q_2)) \,|\, (q_1, q_2) \in Q_1 \times Q_2\}$.

As iso_Q is bijective, $iso_Q(Q_1 \times Q_2) = Q_2 \times Q_1$.

2) Event Set

We will use $id_\Sigma(\Sigma) = \{id_\Sigma(\sigma) \,|\, \sigma \in \Sigma\}$. Clearly, $id_\Sigma(\Sigma) = \Sigma$.

3) Transition Function

In order to clearly argue about the preservation of transitions of $\mathbf{G}$ and $\mathbf{G}'$ later in the proof, we will express our transitions as a 3-tuple. The transitions are represented as a

triple of the form $(q, \sigma, q') \subseteq Q \times \Sigma \times Q$, where $\delta(q, \sigma) = q'$. As such, $\delta \subseteq (Q_1 \times Q_2) \times \Sigma \times (Q_1 \times Q_2)$ and $\delta' \subseteq (Q_2 \times Q_1) \times \Sigma \times (Q_2 \times Q_1)$.

To convert δ, we will use:

$$iso_Q \times id_\Sigma \times iso_Q(\delta) = \{iso_Q \times id_\Sigma \times iso_Q(((q_1, q_2), \sigma, (q_1', q_2'))) | ((q_1, q_2), \sigma, (q_1', q_2')) \in \delta\}$$

We will still need to show that this produces δ'. As iso_Q and id_Σ are bijective functions, their cross product will also be bijective. As $\mathbf{G}_1$ and $\mathbf{G}_2$ are arbitrary TDES, showing that the above produces δ', is thus sufficient to prove the inverse function applied to δ' will produce δ.

4) Initial State

We will use $iso_Q((q_{o,1}, q_{o,2})) = (q_{o,2}, q_{o,1})$.

Clearly, this is a bijective process as $iso_Q^{-1}((q_{o,2}, q_{o,1})) = (q_{o,1}, q_{o,2})$.

5) Set of Marked States

We will use $iso_Q(Q_{m,1} \times Q_{m,2}) = \{iso_Q((q_1, q_2)) \,|\, (q_1, q_2) \in Q_{m,1} \times Q_{m,2}\}$.

As iso_Q is bijective, $iso_Q(Q_{m,1} \times Q_{m,2}) = Q_{m,2} \times Q_{m,1}$.

To map $\mathbf{G}$ to $\mathbf{G}'$, we combine the above mappings, and we can express our function iso as follows:

$$iso((Q_1 \times Q_2, \Sigma, \delta, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})) =$$
$$(iso_Q(Q_1 \times Q_2), id_\Sigma(\Sigma), iso_Q \times id_\Sigma \times iso_Q(\delta), iso_Q(q_{o,1}, q_{o,2}), iso_Q(Q_{m,1} \times Q_{m,2})) =$$
$$(Q_2 \times Q_1, \Sigma, iso_Q \times id_\Sigma \times iso_Q(\delta), (q_{o,2}, q_{o,1}), Q_{m,2} \times Q_{m,1})$$

From the above discussion, it is clear that except for δ, every part of the conversion correctly maps each remaining component of $\mathbf{G}$ onto the corresponding component of $\mathbf{G}'$, and in a bijective manner, i.e. applying the mapping in reverse will map these components of $\mathbf{G}'$ to the corresponding components of $\mathbf{G}$.

Now all that remains is to show that $iso_Q \times id_\Sigma \times iso_Q(\delta) = \delta'$.

The $||_{SD}$ operator defines the transition function δ of $\mathbf{G}$ and δ' of $\mathbf{G}'$ as follows:

$\delta((q_1, q_2), \sigma)$ is only defined and equals:

i) (q_1', q_2') if $\sigma \in (\Sigma_1 \cap \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1' \ \wedge \ \delta_2(q_2, \sigma) = q_2' \ \wedge$
$[(\sigma \neq \tau) \ \vee \ ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib}) \neg \delta((q_1, q_2), \sigma')!)]$

ii) (q_1', q_2) if $\sigma \in (\Sigma_1 - \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1'$

iii) (q_1, q_2') if $\sigma \in (\Sigma_2 - \Sigma_1) \ \wedge \ \delta_2(q_2, \sigma) = q_2'$

$\delta'((q_2, q_1), \sigma)$ is only defined and equals:

i) (q_2', q_1') if $\sigma \in (\Sigma_2 \cap \Sigma_1) \ \wedge \ \delta_1(q_1, \sigma) = q_1' \ \wedge \ \delta_2(q_2, \sigma) = q_2' \ \wedge$
$[(\sigma \neq \tau) \ \vee \ ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib}) \neg \delta'((q_2, q_1), \sigma')!)]$

ii) (q_2, q_1') if $\sigma \in (\Sigma_1 - \Sigma_2) \ \wedge \ \delta_1(q_1, \sigma) = q_1'$

iii) (q_2', q_1) if $\sigma \in (\Sigma_2 - \Sigma_1) \ \wedge \ \delta_2(q_2, \sigma) = q_2'$

By examining and comparing the definitions of δ and δ', we note that the rules specified by δ and δ' are logically identical, i.e. they specify the same logic, in terms of δ_1 of $\mathbf{G}_1$ and δ_2 of $\mathbf{G}_2$, to make decisions about defining transitions and determining next states.

The only difference is the way δ and δ' label the exit and entrance states of their transitions

while specifying their rules. Precisely, if we swap the elements in exit and entrance states' tuples in the rules defined by δ, we essentially get the corresponding rules defined by δ'. This means the definitions of δ and δ' are identical up to reordering of elements in the tuples of their exit and entrance states respectively.

This implies that $\mathbf{G}$ and $\mathbf{G}'$, constructed by $||_{SD}$, essentially have the same set of defined transitions, up to relabelling of their exit and entrance states respectively.

Our isomorphic function $\boldsymbol{iso}$ uses iso_Q and id_Σ to map the transition triples defined in $\mathbf{G}$ to their corresponding transition triples in $\mathbf{G}'$ as follows.

$$(iso_Q((q_1,q_2)), id_\Sigma(\sigma), iso_Q((q_1',q_2'))) = ((q_2,q_1),\sigma,(q_2',q_1')),$$
$$\text{where } (q_1,q_2),(q_1',q_2') \in Q_1 \times Q_2 \text{ and } \sigma \in \Sigma$$

It is noticeable that iso_Q maps the exit and entrance states of a given transition in $\mathbf{G}$ to the respective exit and entrance states of its corresponding transition in $\mathbf{G}'$ by swapping the elements individually in exit and entrance states' tuples.

Since $\mathbf{G}$ and $\mathbf{G}'$ are defined over the same event set Σ, id_Σ preserves the identity of an event $\sigma \in \Sigma$ by mapping σ of $\mathbf{G}$ to σ of $\mathbf{G}'$.

This makes it evident that $\boldsymbol{iso}$ maps the transition triple of $\mathbf{G}$ to its corresponding transition triple in $\mathbf{G}'$ by relabelling the exit and entrance states, and preserving the identity of the event, i.e. if σ transition takes $\mathbf{G}$ from state (q_1,q_2) to (q_1',q_2'), $\boldsymbol{iso}$ maps it to its corresponding σ transition that takes $\mathbf{G}'$ from state (q_2,q_1) to (q_2',q_1').

As $\mathbf{G}$ starts at $(q_{o,1},q_{o,2})$ and $\mathbf{G}'$ at $iso_Q(q_{o,1},q_{o,2}) = (q_{o,2},q_{o,1})$, then for any $\sigma \in \Sigma$ such that $\delta((q_{o,1},q_{o,2}),\sigma) = (q_1',q_2')$, it will also be true that $\delta'((q_{o,2},q_{o,1}),\sigma) = (q_2',q_1') = iso_Q((q_1',q_2'))$.

This means that all transitions leaving the initial state of $\mathbf{G}$ will have a matching isomorphic transition leaving the initial state of $\mathbf{G}'$. It is easy to see that all states reached from the initial state of $\mathbf{G}$ will have an isomorphic state reached from the initial state of $\mathbf{G}'$.

Following this to the logical conclusion, any state reachable in $\mathbf{G}$ will have an isomorphic state reachable in $\mathbf{G}'$. Also, at each reachable state (q_1,q_2) in $\mathbf{G}$, the set of transitions leaving (q_1,q_2) will be isomorphic to the set of transitions leaving state (q_2,q_1) in $\mathbf{G}'$.

Hence, we conclude that $\boldsymbol{iso}$ preserves the structure of the transition function of $\mathbf{G}$ and $\mathbf{G}'$ up to relabelling of exit and entrance states in the defined transitions. In other words, $iso_Q \times id_\Sigma \times iso_Q(\delta) = \delta'$.

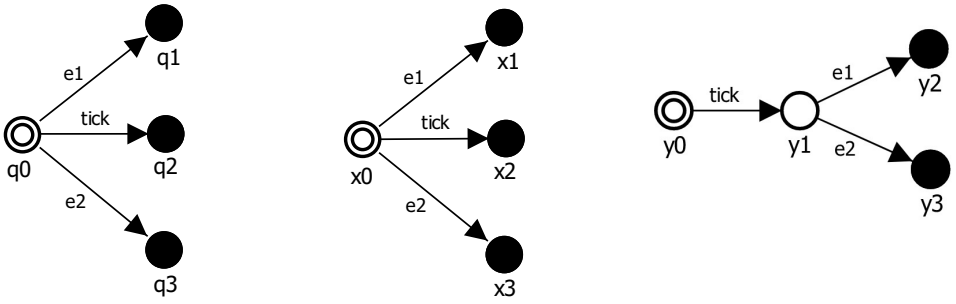
We have thus shown that by using two bijective functions, iso_Q and id_Σ, our isomorphic function $\boldsymbol{iso}$ maps each individual element of $\mathbf{G}$'s quintuple to its corresponding element in $\mathbf{G}'$'s quintuple while preserving its original structure, as defined by $||_{SD}$, up to relabelling of states. Hence, we conclude that $\boldsymbol{iso}$ preserves all automata-theoretic structure of $\mathbf{G}$ and $\mathbf{G}'$ up to state relabelling.

In this way, by constructing our desired isomorphic function, $\boldsymbol{iso}$, we have shown that $\mathbf{G}$ and $\mathbf{G}'$ are isomorphic up to state relabelling.

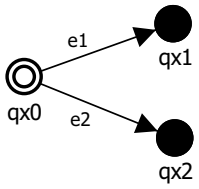
Hence, we conclude that the SD synchronous product of $\mathbf{G}_1$ and $\mathbf{G}_2$ is commutative up to isomorphism. □

4.2.3 Non-Associative Property

The SD synchronous product operator is inherently non-associative, i.e. the order of synchronizing three or more TDES automata using $||_{SD}$ is important and might make a difference in

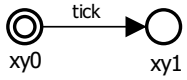


(a) TDES $\mathbf{G}_1$ (b) TDES $\mathbf{G}_2$ (c) TDES $\mathbf{G}_3$

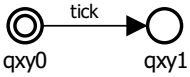


(d) $\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$

(e) $\mathbf{G} = (\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2) \,||_{SD}\, \mathbf{G}_3$



(f) $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$



(g) $\mathbf{G}' = \mathbf{G}_1 \,||_{SD}\, (\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3)$

Figure 16: SD Synchronous Product Operator is Non-Associative

the resultant TDES. In other words, if we have three TDES automata, $\mathbf{G}_1, \mathbf{G}_2$ and $\mathbf{G}_3$, then in general $(\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2) \,||_{SD}\, \mathbf{G}_3 \neq \mathbf{G}_1 \,||_{SD}\, (\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3)$. Below, we demonstrate it with the help of an example.

Figure 16 illustrates the non-associative nature of the $||_{SD}$ operator using three TDES automata, $\mathbf{G}_1$ (Figure 16a), $\mathbf{G}_2$ (Figure 16b) and $\mathbf{G}_3$ (Figure 16c). All TDES are defined over the same event set Σ, such that $\Sigma = \{\text{e1, e2, } \tau\}$ and $\Sigma_{hib} = \{\text{e1, e2}\}$.

First, let us discuss the synchronization mechanism of $||_{SD}$ for constructing TDES $\mathbf{G}$ as $\mathbf{G} = (\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2) \,||_{SD}\, \mathbf{G}_3$. Figure 16d shows the result of synchronizing $\mathbf{G}_1$ and $\mathbf{G}_2$ using $||_{SD}$. As two prohibitable events, e1 and e2, are enabled at the initial states of $\mathbf{G}_1$ and $\mathbf{G}_2$, $||_{SD}$ disables *tick* event at the initial state of $\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$. Thus, the only events possible at the initial state of $\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$ are e1 and e2.

In order to construct $\mathbf{G}$, we synchronize $\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$ with $\mathbf{G}_3$ using $||_{SD}$. We see in Figure 16e that no events are possible at the initial state of $\mathbf{G}$. This is because prohibitable events e1 and e2 that are possible at the initial state of $\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$ have been blocked by $\mathbf{G}_3$ at its initial state. Likewise, *tick* event is possible in $\mathbf{G}_3$ but not in $\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$. This is because the *tick* event, that was originally possible in both $\mathbf{G}_1$ and $\mathbf{G}_2$, has already been disabled by $||_{SD}$ while constructing $\mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2$.

Now we will change the order of synchronizing our three TDES, and construct TDES $\mathbf{G}'$ as $\mathbf{G}' = \mathbf{G}_1 \,||_{SD}\, (\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3)$. At the initial state, prohibitable events e1 and e2 are possible in $\mathbf{G}_2$ but not in $\mathbf{G}_3$. Thus, $||_{SD}$ does not enable these events at the initial state of $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$. As *tick* is possible in both $\mathbf{G}_2$ and $\mathbf{G}_3$, and no prohibitable event is possible in $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$, $||_{SD}$ defines a *tick* transition at the initial state of $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$, as shown in Figure 16f.

To construct $\mathbf{G}'$, we now synchronize $\mathbf{G}_1$ with $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$ using $||_{SD}$. $\mathbf{G}_1$ enables pro-

hibitable events e1 and e2 at its initial state, but since these events are not possible in $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$, $||_{SD}$ does not add their transitions at the initial state of $\mathbf{G}'$. Given that *tick* is possible in both $\mathbf{G}_1$ and $\mathbf{G}_2 \,||_{SD}\, \mathbf{G}_3$, and no prohibitable event is currently possible in $\mathbf{G}'$, $||_{SD}$ enables *tick* at the initial state of $\mathbf{G}'$, as shown in Figure 16g.

By comparing our $\mathbf{G}$ and $\mathbf{G}'$, we note that $\mathbf{G} \neq \mathbf{G}'$. This example clearly demonstrates that the order of synchronizing three TDES using $||_{SD}$ does matter, and we might get different resultant TDES. Hence, we deduce that the $||_{SD}$ operator is inherently non-associative.

4.3 SD Synchronous Product Setting

In our SD synchronous product setting (or "$||_{SD}$ *setting,*" for short), we will use the SD synchronous product operator to combine our TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$. Hence, our closed-loop system is $\mathbf{S} \,||_{SD}\, \mathbf{G}$. Due to the commutative property of the $||_{SD}$ operator, we can synchronize $\mathbf{G}$ and $\mathbf{S}$ in either order, i.e. $\mathbf{G} \,||_{SD}\, \mathbf{S} = \mathbf{S} \,||_{SD}\, \mathbf{G}$.

Note: For consistency, we will always write our closed-loop system as $\mathbf{S} \,||_{SD}\, \mathbf{G}$.

In our $||_{SD}$ setting, we also assume that both $\mathbf{G}$ and $\mathbf{S}$ are defined over the same event set. In case where $\mathbf{G}$ and $\mathbf{S}$ are not defined over the same alphabet, we can simply add selfloops to each TDES for the missing events at every state to extend them over the same event set, without any loss of generality.

In the real world, software designers typically design $\mathbf{G}$ and $\mathbf{S}$ in a modular fashion, rather than as monolithic models. In this case, we assume that these modular plant and supervisor models will be independently synchronized using the standard synchronous product operator to obtain $\mathbf{G}$ and $\mathbf{S}$ respectively. For $m > 1$ plant components, $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_m$, our $\mathbf{G}$ will be obtained as $\mathbf{G} = \mathbf{G}_1 \,||\, \mathbf{G}_2 \,||\, \ldots \,||\, \mathbf{G}_m$. Similarly, for $n > 1$ modular supervisors, $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n$, our $\mathbf{S}$ will be constructed as $\mathbf{S} = \mathbf{S}_1 \,||\, \mathbf{S}_2 \,||\, \ldots \,||\, \mathbf{S}_n$.

There are two reasons for *not* using the $||_{SD}$ operator to combine individual plant and supervisor components to construct $\mathbf{G}$ and $\mathbf{S}$ respectively. The primary reason is the non-associative nature of the $||_{SD}$ operator due to which the order of combining various plant (or supervisor) components becomes important, and different synchronization order will potentially give us a different $\mathbf{G}$ (or $\mathbf{S}$). Moreover, it might also cause our closed-loop system to block, as no events remain possible in TDES $\mathbf{G}$ (Figure 16e) of the example discussed in Section 4.2.3.

Secondly, applying the $||_{SD}$ operator either to plant or supervisor models individually does not look practical and reasonable. The key characteristic of $||_{SD}$ is to automatically disable a *tick* event in the resultant model in cases where source models *agree* on the enablement of one or more prohibitable events. Strictly speaking, there is no concept of enablement/disablement of *tick* event and forcing of prohibitable events solely with respect to either plant or supervisor. Plant model just represents the behaviour of the physical system without any restrictions and constraints. A supervisor is designed to impose control action on the plant model by operating synchronously with it. Hence, it is not justifiable to combine either plant or supervisor components independently using $||_{SD}$, and let only one model decide about the disablement of *tick* event without having any knowledge of the other model's behaviour.

Considering the non-associative property of the $||_{SD}$ operator, it is also evident that once we have formed the closed-loop system using $||_{SD}$, we cannot add any new plant or supervisor component directly to $\mathbf{S} \,||_{SD}\, \mathbf{G}$. This might give us unexpected and problematic results. After constructing $\mathbf{S} \,||_{SD}\, \mathbf{G}$, if we want to add more plant or supervisor models, we need to form

our closed-loop system again. We should first reconstruct our $\mathbf{G}$ and $\mathbf{S}$ separately using the synchronous product, and then combine $\mathbf{G}$ and $\mathbf{S}$ to obtain our closed-loop system $\mathbf{S}\,||_{SD}\,\mathbf{G}$.

However, one possible way to use the $||_{SD}$ operator to combine plant and supervisor components is to synchronize all system models in parallel. In this case, instead of constructing $\mathbf{G}$, $\mathbf{S}$, and $\mathbf{S}\,||_{SD}\,\mathbf{G}$ sequentially, we will synchronize m plant components and n modular supervisors using $||_{SD}$ all at once to construct our closed-loop system. Hence, our closed-loop system will be $\mathbf{G}_1\,||_{SD}\,\mathbf{G}_2\,||_{SD}\,\ldots\,||_{SD}\,\mathbf{G}_m\,||_{SD}\,\mathbf{S}_1\,||_{SD}\,\mathbf{S}_2\,||_{SD}\,\ldots\,||_{SD}\,\mathbf{S}_n$. In this way, the non-associative nature of the $||_{SD}$ operator can be circumvented.

4.4 SD Properties with SD Synchronous Product

TDES and SD properties discussed in the SD setting (Section 3) assume that the closed-loop system is formed by combining TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ using the synchronous product. In our $||_{SD}$ setting, as we have devised a new way of constructing the closed-loop system, these properties need to be adapted with respect to our SD synchronous product operator. In this section, we redefine the TDES and SD properties to match with our $||_{SD}$ setting.

We would like to clarify here that the definitions presented in the following sections are conceptually similar (but not identical) to the ones given in the SD setting. For this reason, we will use the same name followed by "*with SD synchronous product*" to define the adapted version of these properties for our $||_{SD}$ setting. As a shorthand, we will simply write "$\langle$*property name*$\rangle$ *with* $||_{SD}$".

For the following definitions, let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Please note that both $\mathbf{G}$ and $\mathbf{S}$ are defined over the same event set Σ.

4.4.1 Plant Completeness with $||_{SD}$

Definition 4.2. A TDES plant $\mathbf{G}$ is *complete with* $||_{SD}$ for TDES supervisor $\mathbf{S}$ if:

$$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}))\ (\forall \sigma \in \Sigma_{hib})\ s\sigma \in L(\mathbf{S}) \Rightarrow s\sigma \in L(\mathbf{G})$$

In the $||_{SD}$ setting, it says that for all strings s that are possible in $\mathbf{S}\,||_{SD}\,\mathbf{G}$, if a prohibitable event σ is enabled by $\mathbf{S}$ after s, then it must be possible in $\mathbf{G}$ as well.

4.4.2 S-Singular Prohibitable Behaviour with $||_{SD}$

Definition 4.3. For TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$, we say that $\mathbf{G}$ has $\mathbf{S}$-*singular prohibitable behaviour with* $||_{SD}$ if:

$$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp})\ (\forall s' \in \Sigma^*_{act})\ ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \Rightarrow$$
$$(\forall \sigma \in Occu(s') \cap \Sigma_{hib})\ \sigma \notin Elig_{L(\mathbf{G})}(ss')$$

In the $||_{SD}$ setting, this definition states that for a given sampling period, if a prohibitable event σ has already occurred in $\mathbf{S}\,||_{SD}\,\mathbf{G}$, then σ must not be possible in $\mathbf{G}$ again in the same sampling period.

4.4.3 Timed Controllability with $||_{SD}$

As we are building our work on the SD supervisory control, where $\Sigma_{for} = \Sigma_{hib}$, we will adapt and discuss the timed controllability property (Definition 2.22) in terms of prohibitable events

only.

In the SD setting, the closed-loop system is formed by combining plant and supervisor TDES using the synchronous product. In this case, a supervisor is solely in charge of enabling/disabling a *tick* event and forcing prohibitable events in the closed-loop system. The correct behaviour of the supervisor with respect to these decisions is ensured by checking the timed controllability property.

In our $||_{SD}$ setting, we are constructing the closed-loop system using $||_{SD}$. Our $||_{SD}$ operator is also capable of disabling a *tick* event, once a prohibitable event is possible in the plant and enabled by the supervisor. Thus in our setting, in addition to checking that supervisor is enabling/disabling *tick* at the right time, we also need to make sure that the $||_{SD}$ operator does not disable a *tick* event when it is not supposed to, i.e. the $||_{SD}$ operator must not disable a *tick* event when it is possible in the plant and enabled by the supervisor, and no prohibitable events are currently possible in $\mathbf{S}\,||_{SD}\,\mathbf{G}$. Otherwise, our system will become uncontrollable. We capture this notion in the following timed controllability property adapted for our $||_{SD}$ setting.

Definition 4.4. TDES supervisor $\mathbf{S}$ is *timed controllable with* $||_{SD}$ with respect to TDES plant $\mathbf{G}$ if for all $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$,

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$$

It states that all uncontrollable events that are currently possible in $\mathbf{G}$ must be allowed to occur in the closed-loop system, $\mathbf{S}\,||_{SD}\,\mathbf{G}$. In addition, *tick* event must be enabled in $\mathbf{S}\,||_{SD}\,\mathbf{G}$ if it is possible in $\mathbf{G}$, unless there exists an eligible prohibitable event in $\mathbf{S}\,||_{SD}\,\mathbf{G}$ to preempt it. This property makes sure that neither $\mathbf{S}$ nor the $||_{SD}$ operator can disable a *tick* event, if it is possible in $\mathbf{G}$ and no prohibitable events are currently eligible to be forced in the closed-loop system to preempt *tick*.

Note: As our $||_{SD}$ setting is specific to TDES, we will drop the word "timed", and will simply refer to this property as "$\mathbf{S}$ is controllable with $||_{SD}$ with respect to $\mathbf{G}$".

It is notable that the untimed controllability property (Definition 2.20) is part of the standard timed controllability definition (Definition 2.22). Since we have adapted the timed controllability definition for our $||_{SD}$ setting, the untimed controllability property automatically gets redefined as part of it. Below, we explicitly state the untimed controllability with $||_{SD}$ property.

Definition 4.5. TDES supervisor $\mathbf{S}$ is *untimed controllable with* $||_{SD}$ with respect to TDES plant $\mathbf{G}$ if $(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}))\; Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s)$.

4.5 SD Controllability with SD Synchronous Product

This section provides a detailed explanation of how we have adapted the property of SD controllability (Definition 3.7) defined in the SD setting into the property of *SD controllability with SD synchronous product* (*SD controllability with* $||_{SD}$, as a shorthand) for our $||_{SD}$ setting.

In the SD setting, the closed-loop system is constructed by synchronizing $\mathbf{G}$ and $\mathbf{S}$ using the synchronous product, along with the assumption that both $\mathbf{G}$ and $\mathbf{S}$ are defined over the same event set. Therefore, the authors have defined the SD controllability property with respect to the closed language $L(\mathbf{S}) \cap L(\mathbf{G})$, and marked language $L_m(\mathbf{S}) \cap L_m(\mathbf{G})$.

As the synchronization mechanism of the $||_{SD}$ operator is different than the synchronous product, the closed and marked languages generated in the $||_{SD}$ setting will potentially be different than the ones assumed in the SD setting. Keeping this in view, we need to modify the SD controllability definition with respect to the closed and marked languages to make it suitable for our $||_{SD}$ setting. Specifically, we have replaced its $L(\mathbf{S}) \cap L(\mathbf{G})$ with our closed language $L(\mathbf{S} \,||_{SD}\, \mathbf{G})$, and its $L_m(\mathbf{S}) \cap L_m(\mathbf{G})$ with our marked language $L_m(\mathbf{S} \,||_{SD}\, \mathbf{G})$ to make it work for our $||_{SD}$ setting.

Below, we give a formal definition of the SD controllability with $||_{SD}$ property. This is followed by a description of how we logically adapted the individual points of the SD controllability definition to define our SD controllability with $||_{SD}$ property.

Definition 4.6. TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ is *SD controllable with* $||_{SD}$ with respect to TDES plant $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ if, $\forall s \in L(\mathbf{S}||_{SD}\, \mathbf{G})$, the following statements are satisfied:

i) $Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$

ii) If $s \in L_{samp}$ then

1) $(\forall s' \in \Sigma^*_{act})\,[ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})] \Rightarrow$
$$[Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(ss') \cup Occu(s')] \cap \Sigma_{hib} = Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib}$$

2) $(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \wedge Occu(s') = Occu(s'')] \Rightarrow$
$$ss' \equiv_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}\, ss'' \wedge ss' \equiv_{L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})}\, ss''$$

iii) $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \subseteq L_{samp}$

Point i: It says that $\mathbf{S}$ is controllable with $||_{SD}$ with respect to $\mathbf{G}$ (Definition 4.4).

Now we will discuss how this point logically corresponds to Point i and Point ii of the SD controllability definition. Point i of the SD controllability definition is the standard untimed controllability property. Together with Point ii reverse direction ($\Leftarrow$), it becomes the timed controllability property (Definition 2.22) of the SD setting. As we have adapted the timed controllability definition for our $||_{SD}$ setting, we will use our timed controllability with $||_{SD}$ property instead. In this way, Point i of our SD controllability with $||_{SD}$ definition is logically equivalent to Point i and Point ii ($\Leftarrow$) of the SD controllability definition.

In the forward direction ($\Rightarrow$), Point ii of the SD controllability definition states that if a prohibitable event is enabled in the closed-loop system, then *tick* must be disabled. It is noteworthy that this condition is essentially in agreement with the synchronization mechanism of our $||_{SD}$ operator. In simple words, this is exactly what our $||_{SD}$ operator does while synchronizing $\mathbf{G}$ and $\mathbf{S}$, i.e. if a prohibitable event is enabled in the closed-loop system, our $||_{SD}$ operator automatically disables *tick* event in the closed-loop system, even if it is possible in both $\mathbf{G}$ and $\mathbf{S}$.

This implies that our $||_{SD}$ operator guarantees that any closed-loop system constructed as $\mathbf{S}\,||_{SD}\,\mathbf{G}$ will always satisfy the condition imposed by Point ii ($\Rightarrow$) of the SD controllability definition. In our $||_{SD}$ setting, as we construct our closed-loop system as $\mathbf{S}\,||_{SD}\,\mathbf{G}$, this means that we do not need to explicitly check this condition, as it will always be satisfied by the $||_{SD}$ operator while synchronizing $\mathbf{G}$ and $\mathbf{S}$. As a result, we eliminate this explicit condition from our SD controllability with $||_{SD}$ definition. In fact, ensuring the automatic satisfaction of this condition and removing this explicit check is the primary purpose of introducing the

$||_{SD}$ operator and our $||_{SD}$ setting.

In this way, Points i and ii of the SD controllability definition get simplified, and are represented only by Point i in our SD controllability with $||_{SD}$ definition.

Point ii: As a result of the simplification discussed above, Point iii of the SD controllability definition becomes Point ii of the SD controllability with $||_{SD}$ definition. These two points are logically identical except for the way they assume their closed-loop systems to be constructed, which are different for the two settings, SD and $||_{SD}$.

Point iii: Point iii of the SD controllability with $||_{SD}$ definition corresponds to Point iv of the SD controllability definition. These two points essentially represent the same logic. The only difference is their way of representing the marked behaviours, as per the SD and $||_{SD}$ setting.

Since **Point ii** and **Point iii** of the SD controllability with $||_{SD}$ definition are logically identical to Points iii and iv of the SD controllability definition respectively, we have not reexamined these points here. Please refer to Definition 3.7 of SD controllability to see a logical explanation of these points.

4.6 ALF Modularity and SD Synchronous Product

In this section, we present and discuss some important results for the ALF property with respect to our SD synchronous product operator in the $||_{SD}$ setting.

In our $||_{SD}$ setting, we wish our closed-loop system $\mathbf{S}\,||_{SD}\,\mathbf{G}$ to be ALF to rule out the possibility of having the physically unrealistic behaviour that activity events can preempt *tick* for an indefinite amount of time. Instead of first constructing the closed-loop system and then checking its ALF property, it would be much easier and economical if we could find a way to determine whether our closed-loop system is ALF or not before actually constructing it.

One possible way to do this is to apply the ALF check individually on $\mathbf{S}$ and $\mathbf{G}$ before synchronizing them to construct the closed-loop system. The following proposition formally proves that if $\mathbf{S}$ or $\mathbf{G}$ is ALF, then our closed-loop system constructed as $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is guaranteed to be ALF.

Proposition 4.3. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. If either $\mathbf{S}$ or $\mathbf{G}$ is ALF, then the closed-loop system $\boldsymbol{\mathcal{S}} = (Y, \Sigma, \eta, y_o, Y_m)$ constructed as $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF.

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and that either $\mathbf{S}$ or $\mathbf{G}$ is ALF. By Definition 2.30 of the ALF property, this implies:

$$[(\forall s \in \Sigma^+_{act})\,(\forall x \in X_r)\,\xi(x,s) \neq x] \;\vee\; [(\forall s \in \Sigma^+_{act})\,(\forall q \in Q_r)\,\delta(q,s) \neq q] \tag{1}$$

Must show: $\boldsymbol{\mathcal{S}}$ is ALF

By the ALF definition, it is sufficient to show: $(\forall y \in Y_r)\,(\forall s \in \Sigma^+_{act})\,\eta(y,s) \neq y$, where $Y_r \subseteq Y$ is the set of reachable states in $\boldsymbol{\mathcal{S}}$.

We will use proof by contradiction to show that $\boldsymbol{\mathcal{S}}$ is ALF.

Assume $\boldsymbol{\mathcal{S}}$ is not ALF, i.e. there exists an activity loop in $\boldsymbol{\mathcal{S}}$. By Definition 2.29 of activity loop, this implies: $(\exists y \in Y_r)\,(\exists s' \in \Sigma^+_{act})\,\eta(y,s') = y$ (2)

Let $y \in Y_r$, and let $s' \in \Sigma^+_{act}$ such that $\eta(y,s') = y$. (3)

As $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ by (1), by the definition of state set Y in the $||_{SD}$ operator (Definition 4.1), we have: $y = (x,q)$, such that $x \in X$ and $q \in Q$. (4)

Also, by the definition of Y in $||_{SD}$, we know that y is a reachable state in $\boldsymbol{\mathcal{S}}$, only if x is a reachable state in $\mathbf{S}$ and q is a reachable state in $\mathbf{G}$, i.e. $x \in X_r \wedge q \in Q_r$.

By (3), we have: $\eta(y, s') = y$

$\Rightarrow \eta((x, q), s') = (x, q) \quad by\ (4)$

As $\mathbf{S}$ and $\mathbf{G}$ are defined over the same event set Σ, by *Point i* of the $||_{SD}$ definition, we have that a transition will be defined at a state in $\boldsymbol{\mathcal{S}}$, only if it is defined at the corresponding states in both $\mathbf{S}$ and $\mathbf{G}$.

Since we have that transition for string s' is defined at state $y = (x, q)$ in $\boldsymbol{\mathcal{S}}$, this implies that s' transition is defined at state x in $\mathbf{S}$ and state q in $\mathbf{G}$.

$\Rightarrow \xi(x, s') = x \wedge \delta(q, s') = q \quad$ *by Point i of* $||_{SD}$ *definition*

These transitions indicate that both $\mathbf{S}$ and $\mathbf{G}$ are not ALF. This contradicts our assumption of (1) that either $\mathbf{S}$ or $\mathbf{G}$ is ALF.

Thus, we deduce that our assumption of (2) is false, and $\boldsymbol{\mathcal{S}}$ is ALF.

$\Rightarrow (\forall y \in Y_r)\,(\forall s \in \Sigma^{+}_{act})\ \eta(y, s) \neq y$

Hence, we conclude that if either $\mathbf{S}$ or $\mathbf{G}$ is ALF, then $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF. □

As $\mathbf{S}$ and $\mathbf{G}$ are typically designed modularly by designers, our $\mathbf{S}$ and $\mathbf{G}$ will most likely be constructed as $\mathbf{S} = \mathbf{S}_1\,||\,\mathbf{S}_2\,||\,\ldots\,||\,\mathbf{S}_m$ and $\mathbf{G} = \mathbf{G}_1\,||\,\mathbf{G}_2\,||\,\ldots\,||\,\mathbf{G}_n$, where $m, n > 1$. Here, it is worthwhile to mention a proposition from [42] that presents an easy and modular way of obtaining an ALF TDES. The following proposition states that if each individual TDES is ALF, then their synchronous product is ALF. This proposition is useful in our $||_{SD}$ setting as we can make our $\mathbf{S}$ or $\mathbf{G}$ ALF just by making sure that each individual plant or supervisor component is ALF, even if these components are defined over different event sets.

Proposition 4.4. [42] For TDES $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2, \delta_2, q_{o,2}, Q_{m,2})$, if $\mathbf{G}_1$ and $\mathbf{G}_2$ are each ALF, then their synchronous product $\mathbf{G} = \mathbf{G}_1\,||\,\mathbf{G}_2$ is ALF.

In the presence of Proposition 4.3 and Proposition 4.4, it is evident that if we want to construct an ALF closed-loop system in our $||_{SD}$ setting, we simply need to design ALF plant or supervisor components. This is because individual ALF plant or supervisor components ensure that when we synchronize them using synchronous product, our $\mathbf{S}$ or $\mathbf{G}$ will be ALF (Proposition 4.4). This in turn guarantees that our closed-loop system constructed as $\mathbf{S}\,||_{SD}\,\mathbf{G}$ will be ALF (Proposition 4.3). In this way, we can verify the ALF property of our closed-loop system before actually constructing $\mathbf{S}\,||_{SD}\,\mathbf{G}$, or even before constructing composite $\mathbf{S}$ and $\mathbf{G}$.

For our closed-loop system $\mathbf{S}\,||_{SD}\,\mathbf{G}$, we are also interested in making sure that our system does not try to "stop the clock", i.e. it should never reach a state where *tick* events are not possible anymore, as this behaviour is undesirable and physically unrealistic. Therefore, we want to guarantee that after a finite number of activity events, our system should always reach a state where the *tick* event is possible. In the following proposition, we present sufficient conditions to ensure this behaviour. This proposition is inspired by Proposition 6.6 taken from [29].

Proposition 4.5. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, and let $\boldsymbol{\mathcal{S}} = (Y, \Sigma, \eta, y_o, Y_m)$ be the closed-loop system constructed as $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$. If both $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces, $\mathbf{G}$ has proper time behaviour, $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}}$ is ALF, then $(\forall y \in Y_r)\,(\exists s \in \Sigma^{*}_{act})\ \eta(y, s\tau)!$.

Proof. Assume initial conditions, and let $y \in Y_r$.

Must show: $(\exists s \in \Sigma^*_{act})\ \eta(y, s\tau)!$

As both $\mathbf{G}$ and $\mathbf{S}$ have finite, non-empty state spaces (as they both contain an initial state), it follows from the definition of state set Y in the $||_{SD}$ operator (Definition 4.1) that the closed-loop system $\boldsymbol{\mathcal{S}} = \mathbf{S}||_{SD}\ \mathbf{G}$ has a finite, non-empty state space.

Let $n = |Y_r|$.

By our initial assumption, we have that $\boldsymbol{\mathcal{S}}$ is ALF. This implies that starting at state y in $\boldsymbol{\mathcal{S}}$, the system can do at most $n-1$ activity event transitions before it has visited all n reachable states. At this point, there must be no more activity event transitions possible in $\boldsymbol{\mathcal{S}}$. Otherwise, the system would have to visit a state twice, thus creating an activity loop and failing the ALF definition.

This idea can be formally expressed as follows:

$$(\exists s \in \Sigma^*_{act})\ |s| \leq n-1\ \wedge\ (\exists y' \in Y_r)\ \eta(y, s) = y'\ \wedge\ (\forall \sigma \in \Sigma_{act})\ \neg\, \eta(y', \sigma)! \qquad (1)$$

Now we will present our argument to show that *tick* transition is defined at state y' in $\boldsymbol{\mathcal{S}}$.

By (1), we have that no activity events are possible at state y' in $\boldsymbol{\mathcal{S}}$. As $\Sigma_u \subseteq \Sigma_{act}$, this means that no uncontrollable events are possible at y' in $\boldsymbol{\mathcal{S}}$.

By our initial assumption, we have that $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$. This implies that no uncontrollable events are defined at the corresponding state in $\mathbf{G}$, as $\mathbf{S}$ would not have restricted them, and there are none possible in $\boldsymbol{\mathcal{S}}$.

Lets refer to this state of $\mathbf{G}$ as $q' \in Q$.

As $\mathbf{G}$ has proper time behaviour, this implies that *tick* is defined at state q' in $\mathbf{G}$.

By (1), we have that no activity events are possible at state y' in $\boldsymbol{\mathcal{S}}$. As $\Sigma_{hib} \subseteq \Sigma_{act}$, this means no prohibitable events are enabled at y' in $\boldsymbol{\mathcal{S}}$.

We have that *tick* event is defined at state q' in $\mathbf{G}$ and no prohibitable event is eligible at state y' to preempt the *tick* in $\boldsymbol{\mathcal{S}}$. As $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$, this implies that neither $\mathbf{S}$ nor the $||_{SD}$ operator can disable the *tick* event at this point in time. Thus, the *tick* event must be enabled at state y' in $\boldsymbol{\mathcal{S}}$.

Thus, we have: $\eta(y', \tau)!$

$\Rightarrow \eta(\eta(y, s), \tau)!$ *by* (1)

$\Rightarrow \eta(y, s\tau)!$ *by definition of transition function*

Hence, we conclude $(\forall y \in Y_r)\ (\exists s \in \Sigma^*_{act})\ \eta(y, s\tau)!$. □

5 Equivalence of SD and SD Synchronous Product Setting

Now that we have described our SD synchronous product setting in detail, our next target is to establish equivalence between the SD setting (Section 3) and our $||_{SD}$ setting (Section 4). This section serves as the first stepping stone to achieve this goal.

We begin this section by presenting a discussion on why this equivalence between the two settings is needed, how it will be established, and how it will pave the way for proving controllability, nonblocking and all SD verification results in our $||_{SD}$ setting (Section 8). After this discussion, we state some assumptions that apply to our complete study. This is followed by our language equivalence results, where we establish and formally prove equivalence between the closed and marked languages of the SD and $||_{SD}$ setting. Utilizing these results, we then demonstrate the equivalence between various SD properties in the two settings.

5.1 Establishing Equivalence

In this section, we present a detailed discussion on establishing equivalence between the SD and $||_{SD}$ settings. First, we explain why we opted for establishing equivalence between the two settings. After that, we provide a complete road map to establish our desired equivalence by presenting a comprehensive description of how did we plan to prove this equivalence and make use of it while performing our controllability and nonblocking verification in the $||_{SD}$ setting.

5.1.1 Why Equivalence is Needed?

In our $||_{SD}$ setting, we have presented a novel way of constructing the closed loop system by synchronizing TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ using the $||_{SD}$ operator. By doing this, we have essentially changed the way of obtaining closed and marked languages for the system. Since our $||_{SD}$ operator generates a new system language that, in most cases, will not be the same as the language generated by synchronous product in the SD setting, the controllability and nonblocking verification results of the SD setting do not remain valid in our $||_{SD}$ setting. This means that we need to reprove all verification results of the SD setting for our $||_{SD}$ setting.

There are two possible ways to perform controllability and nonblocking verification in our $||_{SD}$ setting: 1) prove all SD results from scratch, or 2) establish some kind of logical equivalence between the SD and $||_{SD}$ setting, so that the results that have already been proven in the SD setting remain applicable to our $||_{SD}$ setting as well. This will allow us to reuse and base our results on some of the existing results from the SD setting while performing the controllability and nonblocking verification in our $||_{SD}$ setting.

Hypothetically, we could follow the first approach and prove all SD results in our $||_{SD}$ setting from scratch. But the issue with this approach is that there is no closed and obvious form of the closed and marked language that is generated by synchronizing $\mathbf{G}$ and $\mathbf{S}$ using the $||_{SD}$ operator, i.e. $L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$. By this we mean that, apparently, we cannot easily express these languages in terms of the natural projection or its equivalent, as has been done for the synchronous product. For plant $\mathbf{G}$ and an arbitrary supervisor $\mathbf{S}$ that are defined over the same event set Σ, we know that $L(\mathbf{S}\,||\,\mathbf{G}) = L(\mathbf{S}) \cap L(\mathbf{G})$ and $L_m(\mathbf{S}\,||\,\mathbf{G}) = L_m(\mathbf{S}) \cap L_m(\mathbf{G})$. However, in most cases, we do not expect to have this kind of equality for our $||_{SD}$ operator, i.e. $L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \neq L(\mathbf{S}) \cap L(\mathbf{G})$ and $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \neq L_m(\mathbf{S}) \cap L_m(\mathbf{G})$. This is due to the automatic *tick* disablement mechanism of the $||_{SD}$ operator that is not present in the synchronous product operator.

In the absence of such a closed and clear cut form for $L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$, working with these languages and proving all SD verification results from scratch in the $||_{SD}$ setting does not look like a straightforward and trouble-free task. Due to these reasons, we opt for the second approach of establishing logical equivalence between the SD and $||_{SD}$ setting, and then utilize this equivalence to perform our controllability and nonblocking verification in the $||_{SD}$ setting.

5.1.2 How to Establish Equivalence?

Note: In this discussion, in fact in the rest of this report, we need to talk about two supervisors, one from the SD setting and the other from our $||_{SD}$ setting. Since these two supervisors will most likely be different (as they may satisfy different properties of the two settings), therefore, in order to avoid any ambiguity, we will use two different symbols to refer to them.

The supervisor of the SD setting will be stated as $\boldsymbol{\mathcal{S}}$ ($\boldsymbol{\mathcal{S}}$ maps and refers to $\mathbf{S}$ of Section 3), whereas the supervisor of our $||_{SD}$ setting will be referred to as $\mathbf{S}$.

In the SD setting, since TDES plant $\mathbf{G}$ and TDES supervisor $\boldsymbol{\mathcal{S}}$ are assumed to be combined with the synchronous product, therefore all verification results have been proven using the closed language $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and marked language $L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$. The authors have assumed that $\boldsymbol{\mathcal{S}}$ is an arbitrary supervisor that satisfies certain SD properties, independently and when combined with $\mathbf{G}$ to form the closed-loop system, $\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}$. This supervisor $\boldsymbol{\mathcal{S}}$ is then used to generate its corresponding controller implementation in the SD setting using the translation method described in Section 3.7. Therefore, in order to make the SD results valid in our $||_{SD}$ setting and derive our results based on these existing results, we need to satisfy all these conditions and prove equivalence at all levels, i.e. 1) prove language equivalence, 2) satisfy all properties that have been considered as preconditions for concluding the controllability and nonblocking verification results, and 3) prove controller's equivalence.

To establish the required logical equivalence, first and foremost, we need to establish language equivalence between the two settings. Since the closed and marked languages generated in the SD and $||_{SD}$ settings are $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $L(\mathbf{S} \,||_{SD}\, \mathbf{G})$, and $L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$ and $L_m(\mathbf{S} \,||_{SD}\, \mathbf{G})$ respectively, we can potentially establish language equivalence between the two settings if we could somehow prove that $L(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $L_m(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$. To do this, we need to find an appropriate and concrete definition for supervisor $\boldsymbol{\mathcal{S}}$ of the SD setting, that is not only guaranteed to exist, but should be based on or somehow related to our $\mathbf{S} \,||_{SD}\, \mathbf{G}$ to achieve the above-mentioned equivalence.

An intriguing idea is what if we define $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$? Can we establish our desired language equivalence between the two settings with this definition of $\boldsymbol{\mathcal{S}}$? Can we demonstrate that $\boldsymbol{\mathcal{S}}$ satisfies all the properties as required by existing verification results of the SD setting? Can we prove that the controller implementation of $\boldsymbol{\mathcal{S}}$ will be according to the requirements of the SD setting? If we can prove these things, then we can certainly define $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ to prove the desired equivalence, and make use of the existing SD results while verifying our $||_{SD}$ setting.

This is exactly the approach that we adopt for proving equivalence between the two settings. We start by establishing language equivalence between the two settings and proving that if $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, then both settings have the same closed and marked behaviours. Specifically, we prove that $L(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $L_m(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$ (Section 5.3).

Then, we focus on satisfying the preconditions (various SD properties) of the SD verification results. Specifically, we demonstrate that if certain SD properties are satisfied in the $||_{SD}$ setting with respect to our supervisor $\mathbf{S}$, this implies that their corresponding SD properties are guaranteed to be satisfied with respect to supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ in the SD setting (Section 5.4). We also show how to process $\boldsymbol{\mathcal{S}}$ to satisfy some other properties that are required in the SD setting but may not be directly satisfied as $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ (Section 6).

Finally, we prove that the SD controller that is obtained by translating $\mathbf{S}$ in our $||_{SD}$ setting is output equivalent to the controller that is generated by supervisor $\boldsymbol{\mathcal{S}}$ of the SD setting with respect to valid input strings, i.e. strings that are possible in the two closed-loop behaviours (Section 7). In other words, controller implementation of the two supervisors, $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}$, will exhibit exactly the same control behaviour with respect to TDES plant $\mathbf{G}$.

Once we have this formal equivalence between the two settings in place, we will successfully satisfy all the assumptions and preconditions that have been identified for proving all verification results in the SD setting. Since we have fulfilled all the prerequisites, we can

rightly conclude the SD verification results. In this way, the existing SD results become valid in the $||_{SD}$ setting and we can easily reuse them to build our controllability and nonblocking verification results of the $||_{SD}$ setting (Section 8).

Before closing this section, it is also important to clearly state the relationship that we have established between the two settings to do our formal theoretical verification. The basic idea is that in the $||_{SD}$ setting, supervisor $\mathbf{S}$ is expected to be manually designed by the designers for plant $\mathbf{G}$, and is required to satisfy certain SD properties with $||_{SD}$, defined in Section 4. It is worth-mentioning here that while designing $\mathbf{S}$, designers do not need to manually take care of the tricky condition imposed by Point ii ($\Rightarrow$) of the SD controllability definition, as required in the SD setting. This $\mathbf{S}$ should then be synchronized with $\mathbf{G}$ using our $||_{SD}$ operator to construct $\mathbf{S}\,||_{SD}\,\mathbf{G}$.

Instead of using this $\mathbf{S}\,||_{SD}\,\mathbf{G}$ as our closed-loop system for theoretical verification of the $||_{SD}$ setting, we treat this $\mathbf{S}\,||_{SD}\,\mathbf{G}$ as the "supervisor" of the SD setting, i.e. $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$. We will be able to do this because of our equivalence results, since these results ensure that $\boldsymbol{\mathcal{S}}$ is guaranteed to satisfy all properties that a supervisor of the SD setting is required to satisfy. It is noteworthy that $\boldsymbol{\mathcal{S}}$ is also guaranteed to automatically satisfy Point ii ($\Rightarrow$) of the SD controllability definition with respect to $\mathbf{G}$ because of the synchronization mechanism of our $||_{SD}$ operator that is used to construct $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$.

This $\boldsymbol{\mathcal{S}}$ is then assumed to be synchronized with $\mathbf{G}$ using the synchronous product to construct the closed-loop system $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$, with closed language $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and marked language $L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$, as done in the existing SD setting. All the existing SD verification results then follow immediately, as they have been proven using the same closed and marked languages in the SD setting.

We would like to clarify that software and hardware practitioners do not actually need to construct supervisor $\boldsymbol{\mathcal{S}}$ or closed-loop system $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ of the SD setting in practice. Also, they are not required to physically implement $\boldsymbol{\mathcal{S}}$ as their controller. This additional step is only considered and discussed here with respect to theoretical verification of our $||_{SD}$ setting. Practically, designers and practitioners only need to design supervisor $\mathbf{S}$ with the desired SD properties of the $||_{SD}$ setting. This supervisor can then be translated to generate its corresponding controller implementation using the translation method described in Section 3.7.

In this way, our $||_{SD}$ setting inherently liberates the designers from manually designing the potentially intricate supervisor of the SD setting that must satisfy all SD conditions, especially the stringent SD controllability Point ii ($\Rightarrow$). Using our approach, they should now be able to design a much simpler and less complicated supervisor $\mathbf{S}$ of the $||_{SD}$ setting that, when combined with $\mathbf{G}$ using the $||_{SD}$ operator, is equivalent in its closed-loop behaviour, control action and controller implementation to the one required by the SD setting. In other words, by introducing the concrete definition of $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, we essentially provide a way to automatically generate a TDES $\boldsymbol{\mathcal{S}}$ that qualifies as the supervisor of the SD setting by satisfying all the required properties and conditions.

5.2 Implicit Assumptions

In this section, we list down our implicit assumptions that hold true for our $||_{SD}$ setting. Since these assumptions apply to our complete study, we are stating them together at one place, and will not repeat them in any of the upcoming sections.

1. For TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ of the $||_{SD}$ setting, we assume that both $\mathbf{G}$ and $\mathbf{S}$ are always defined over the same event set. However, in the case where $\mathbf{G}$ and $\mathbf{S}$

are not defined over the same alphabet, we can simply add selfloops to each TDES for the missing events at every state to extend them over the same event set, without any loss of generality. If we assume otherwise in any particular section of this report, we will explicitly state that.

2. Let TDES $\boldsymbol{\mathcal{S}}$ be constructed by synchronizing plant $\mathbf{G}$ and supervisor $\mathbf{S}$ using the SD synchronous product operator, i.e. $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$. Since both $\mathbf{G}$ and $\mathbf{S}$ are defined over the same event set, by definition of the $||_{SD}$ operator, the resultant TDES $\boldsymbol{\mathcal{S}}$ will also have the same event set as $\mathbf{G}$ and $\mathbf{S}$.
3. As $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ are defined over the same event set, by Definition 2.19 of the synchronous product, we have that $L(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$. In the rest of this report, we might interchangeably use these two representations of synchronous product without explicit explanation.
4. In the SD supervisory control theory, it has been assumed that the set of prohibitable events (Σ_{hib}) is exactly equal to the set of forcible events (Σ_{for}), i.e. $\Sigma_{for} = \Sigma_{hib}$. Since we are using this methodology as the basis of our work, this assumption holds true for our study as well.
5. All TDES discussed in this report are assumed to be reachable and deterministic with a finite state space and a finite event set.

5.3 Equivalence of Languages

In this section, we present our desired language equivalence results for the SD and $||_{SD}$ setting. Specifically, we formally prove that the closed and marked languages generated in the two settings are equivalent.

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let $\boldsymbol{\mathcal{S}} = (Y, \Sigma, \eta, y_o, Y_m)$ be a TDES constructed as $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$.

We start by proving two propositions that will help us in showing our main language equivalence result. The basic idea of these two propositions has been taken from Definition 4.1 of our SD synchronous product operator.

By looking at the synchronization mechanism of the $||_{SD}$ operator, we note that $||_{SD}$ 'potentially' adds a transition to $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, only if that transition is defined in both $\mathbf{S}$ and $\mathbf{G}$. It does not add any transition to $\boldsymbol{\mathcal{S}}$ that is not defined in either $\mathbf{S}$ or $\mathbf{G}$. This implies that the strings defined in $L(\boldsymbol{\mathcal{S}})$ are going to be a subset of the strings that are defined in both $L(\mathbf{S})$ and $L(\mathbf{G})$. The proposition given below uses proof by induction to formally prove this notion.

Proposition 5.1. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, then: **(i)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$, and **(ii)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{G})$.

Proof. We will prove these two points together.

Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ (1)

Must show: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$ and $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{G})$

Sufficient to show: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S}) \cap L(\mathbf{G})$

Let $s \in L(\boldsymbol{\mathcal{S}})$. Must show this implies: $s \in L(\mathbf{S}) \cap L(\mathbf{G})$

We will use induction on the length of s to show: $s \in L(\mathbf{S}) \cap L(\mathbf{G})$

Base Case: $s = \epsilon$

As $\mathbf{S}$ contains an initial state x_o, and $\mathbf{G}$ contains an initial state q_o, it follows that $\epsilon \in L(\mathbf{S})$ and $\epsilon \in L(\mathbf{G})$.

$\Rightarrow \epsilon \in L(\mathbf{S}) \cap L(\mathbf{G})$

Base case complete.

Inductive Step: For some $k \geq 0$, we assume:

- $s = \sigma_1 \dots \sigma_k \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{S}) \cap L(\mathbf{G})$ (2)
- $s\sigma_{k+1} \in L(\boldsymbol{\mathcal{S}})$ (3)

We will now show this implies: $s\sigma_{k+1} \in L(\mathbf{S}) \cap L(\mathbf{G})$

By (2), we have: $s \in L(\boldsymbol{\mathcal{S}})$

$\Rightarrow \eta(y_o, s)!$ *by definition of* $L(\boldsymbol{\mathcal{S}})$ (4)

We have $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ by (1). The $||_{SD}$ operator defines the initial state of $\boldsymbol{\mathcal{S}}$ as an ordered pair of the initial states of $\mathbf{S}$ and $\mathbf{G}$.

$\Rightarrow \eta((x_o, q_o), s)!$ *by definition of* y_o *in* $||_{SD}$ *definition* (5)

As $\mathbf{S}$ and $\mathbf{G}$ are defined over the same event set Σ, by *Point i* of the $||_{SD}$ operator's definition, we have that a transition will be defined at a state in $\boldsymbol{\mathcal{S}}$, only if it is defined at corresponding states in both $\mathbf{S}$ and $\mathbf{G}$. Also, we know that $||_{SD}$ operator is defined in such a way that it may remove a *tick* transition from $\boldsymbol{\mathcal{S}}$ under certain conditions, even though that *tick* transition is possible in both $\mathbf{S}$ and $\mathbf{G}$, but it cannot add any *tick* or non-*tick* transition to $\boldsymbol{\mathcal{S}}$ that is not defined in either $\mathbf{S}$ or $\mathbf{G}$.

Since, by (4), we have that string s is defined at state y_o in $\boldsymbol{\mathcal{S}}$, by (5) this implies that s is defined at state x_o in $\mathbf{S}$ and state q_o in $\mathbf{G}$.

$\Rightarrow \xi(x_o, s)! \wedge \delta(q_o, s)!$ *by Point i of* $||_{SD}$ *definition* (6)

By (3), we have: $s\sigma_{k+1} \in L(\boldsymbol{\mathcal{S}})$

$\Rightarrow \eta(y_o, s\sigma_{k+1})!$ *by definition of* $L(\boldsymbol{\mathcal{S}})$

$\Rightarrow \eta(\eta(y_o, s), \sigma_{k+1})!$ *by* (4) *and definition of transition function*

$\Rightarrow \eta(\eta((x_o, q_o), s), \sigma_{k+1})!$ *by* (5)

As σ_{k+1} transition is defined in $\boldsymbol{\mathcal{S}}$, by *Point i* of $||_{SD}$ definition, this implies that σ_{k+1} transition is defined at corresponding states in both $\mathbf{S}$ and $\mathbf{G}$.

$\Rightarrow \xi(\xi(x_o, s), \sigma_{k+1})! \wedge \delta(\delta(q_o, s), \sigma_{k+1})!$ *by* (6) *and Point i of* $||_{SD}$ *definition*

$\Rightarrow \xi(x_o, s\sigma_{k+1})! \wedge \delta(q_o, s\sigma_{k+1})!$ *by definition of transition function*

$\Rightarrow s\sigma_{k+1} \in L(\mathbf{S}) \wedge s\sigma_{k+1} \in L(\mathbf{G})$ *by definition of* $L(\mathbf{S})$ *and* $L(\mathbf{G})$

$\Rightarrow s\sigma_{k+1} \in L(\mathbf{S}) \cap L(\mathbf{G})$

Inductive step complete.

By our base case and inductive step, we have shown that for some arbitrary string $s \in L(\boldsymbol{\mathcal{S}})$ implies $s \in L(\mathbf{S}) \cap L(\mathbf{G})$. Thus, we have shown that $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S}) \cap L(\mathbf{G})$.

Hence, we conclude: **(i)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$, and **(ii)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{G})$. □

In the next proposition, we prove same idea with respect to marked languages of $\mathbf{S}$, $\mathbf{G}$ and $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$. Specifically, we show that the marked strings that make up $L_m(\boldsymbol{\mathcal{S}})$ is a subset of the marked strings that are defined in both $L_m(\mathbf{S})$ and $L_m(\mathbf{G})$. This proof is partially based on the result of our previous proposition.

Proposition 5.2. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, then: **(i)** $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{S})$, and **(ii)** $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{G})$.

Proof. We will prove these two points together.

Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$. Sufficient to show: $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{S}) \cap L_m(\mathbf{G})$

Let $s \in L_m(\boldsymbol{\mathcal{S}})$. Must show this implies: $s \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$

We have: $s \in L_m(\boldsymbol{\mathcal{S}})$

$\Rightarrow \eta(y_o, s)! \wedge \eta(y_o, s) \in Y_m$ *by definition of* $L_m(\boldsymbol{\mathcal{S}})$

$\Rightarrow s \in L(\boldsymbol{\mathcal{S}}) \wedge \eta(y_o, s) \in Y_m$ *by definition of* $L(\boldsymbol{\mathcal{S}})$ (1)

As $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, thus by Proposition 5.1, we have: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$ and $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{G})$

$\Rightarrow s \in L(\mathbf{S}) \wedge s \in L(\mathbf{G})$ *by Proposition* 5.1 (2)

The $||_{SD}$ operator defines the initial state of $\boldsymbol{\mathcal{S}}$ as an ordered pair of the initial states of $\mathbf{S}$ and $\mathbf{G}$, and the set of marked states of $\boldsymbol{\mathcal{S}}$ as cross product of the set of marked states of $\mathbf{S}$ and $\mathbf{G}$.

By (1), we have: $\eta(y_o, s) \in Y_m$

$\Rightarrow \eta((x_o, q_o), s) \in X_m \times Q_m$ *by* (2) *and definition of* y_o *and* Y_m *in* $||_{SD}$ *definition*

$\Rightarrow \xi(x_o, s) \in X_m \wedge \delta(q_o, s) \in Q_m$ *by Point i and definition of* Y_m *in* $||_{SD}$ *definition*

$\Rightarrow s \in L_m(\mathbf{S}) \cap L_m(\mathbf{G})$

Hence, we conclude: **(i)** $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{S})$, and **(ii)** $L_m(\boldsymbol{\mathcal{S}}) \subseteq L_m(\mathbf{G})$. □

Based on the above two propositions, we now present our main result of proving language equivalence between the SD and our $||_{SD}$ setting. In the following proposition, we prove that the closed and marked languages generated by synchronizing TDES supervisor $\mathbf{S}$ and TDES plant $\mathbf{G}$ using $||_{SD}$ operator in the $||_{SD}$ setting are the same as the closed and marked languages obtained by combining TDES supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ and TDES plant $\mathbf{G}$ using synchronous product operator in the SD setting.

Proposition 5.3. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, then: **(i)** $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, and **(ii)** $L_m(\boldsymbol{\mathcal{S}}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$.

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$

i) Show: $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

Sufficient to show: **(1)** $L(\boldsymbol{\mathcal{S}}) \subseteq L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, and **(2)** $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \subseteq L(\boldsymbol{\mathcal{S}})$.

1) Show: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

Let $s \in L(\boldsymbol{\mathcal{S}})$. Must show this implies: $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ (1)

As $s \in L(\boldsymbol{\mathcal{S}})$ by (1), sufficient to show: $s \in L(\mathbf{G})$

As $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, thus by Proposition 5.1, we have: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S}) \cap L(\mathbf{G})$

$\Rightarrow s \in L(\mathbf{G})$ *by* (1) *and Proposition* 5.1

We thus conclude that $L(\boldsymbol{\mathcal{S}}) \subseteq L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.

2) Show: $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \subseteq L(\boldsymbol{\mathcal{S}})$

This follows automatically from the definition of set intersection.

By Parts (1) and (2), we conclude that $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.

ii) Show: $L_m(\boldsymbol{\mathcal{S}}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$

Proof is identical to Part (i) up to relabelling closed languages $L(\mathbf{S})$, $L(\mathbf{G})$ and $L(\boldsymbol{\mathcal{S}})$ to marked languages $L_m(\mathbf{S})$, $L_m(\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}})$ respectively, and replacing Proposition 5.1 with Proposition 5.2 in Part (1). □

By our assumptions, we know that $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$, $L(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, and $L_m(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$. This means that Proposition 5.3 can be stated in multiple ways. Below we derive a corollary based on our main language equivalence result. We will then refer to the various points of this corollary to directly cite the result in the required form in the upcoming proofs.

Corollary 5.1. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, then:

i) $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G})$ **ii)** $L_m(\boldsymbol{\mathcal{S}}) = L_m(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G})$

iii) $L(\mathbf{S} ||_{SD} \mathbf{G}) = L(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G})$ **iv)** $L_m(\mathbf{S} ||_{SD} \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G})$

v) $L(\mathbf{S} ||_{SD} \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ **vi)** $L_m(\mathbf{S} ||_{SD} \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$

i) Show: $L(\boldsymbol{\mathcal{S}}) = L(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G})$

As both $\boldsymbol{\mathcal{S}}$ and $\mathbf{G}$ are defined over Σ, we thus have: $L(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

The result follows automatically from Proposition 5.3.

ii) Show: $L_m(\boldsymbol{\mathcal{S}}) = L_m(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G})$

Proof is identical to Part (i) up to relabelling closed languages $L(\mathbf{S})$, $L(\mathbf{G})$ and $L(\boldsymbol{\mathcal{S}})$ to marked languages $L_m(\mathbf{S})$, $L_m(\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}})$ respectively.

iii) Show: $L(\mathbf{S} ||_{SD} \mathbf{G}) = L(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G})$

As $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$, the result follows immediately from Part (i).

iv) Show: $L_m(\mathbf{S} ||_{SD} \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}} \,||\, \mathbf{G})$

As $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$, the result follows immediately from Part (ii).

v) Show: $L(\mathbf{S} ||_{SD} \mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

As $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$, the result follows immediately from Proposition 5.3(i).

vi) Show: $L_m(\mathbf{S} ||_{SD} \mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$

As $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$, the result follows immediately from Proposition 5.3(ii). □

5.4 Equivalence of SD Properties

In this section, we prove equivalence between the two versions of various properties that are defined in the SD and $||_{SD}$ setting.

In our $||_{SD}$ setting, we expect TDES supervisor $\mathbf{S}$ to be manually designed by software designers, and is required to satisfy certain properties. By introducing the $||_{SD}$ setting, we are devising a way to automatically construct the supervisor of the SD setting as $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$. This means we must also provide a way to automatically satisfy various properties that the supervisor of the SD setting is required to satisfy. This is discussed in the following subsections. Specifically, in these subsections, we formally prove that if $\mathbf{S}$ satisfies certain properties with respect to TDES plant $\mathbf{G}$ in our $||_{SD}$ setting, this implies that TDES supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$ is guaranteed to satisfy the corresponding SD properties with respect to $\mathbf{G}$ in the SD setting.

5.4.1 Plant Completeness

In the SD setting, it is required that plant TDES should be complete for the supervisor TDES. In the following proposition, we prove that if plant $\mathbf{G}$ is complete with $||_{SD}$ for supervisor $\mathbf{S}$ in our $||_{SD}$ setting, then this is sufficient to ensure that $\mathbf{G}$ is complete for supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S}||_{SD}\,\mathbf{G}$ in the SD setting.

Proposition 5.4. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, and TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor. If $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$, then $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$.

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S}||_{SD}\,\mathbf{G}$, and $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$ (1)

To show that $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, it is sufficient to show:

$$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}))\ (\forall \sigma \in \Sigma_{hib})\ s\sigma \in L(\boldsymbol{\mathcal{S}}) \Rightarrow s\sigma \in L(\mathbf{G})$$

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and let $\sigma \in \Sigma_{hib}$. Assume: $s\sigma \in L(\boldsymbol{\mathcal{S}})$ (2)

Must show this implies: $s\sigma \in L(\mathbf{G})$

By (2), we have: $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

$\Rightarrow s \in L(\mathbf{S}||_{SD}\,\mathbf{G})$ *by* (1) *and Corollary* 5.1(v) (3)

As $\boldsymbol{\mathcal{S}} = \mathbf{S}||_{SD}\,\mathbf{G}$ by (1), thus by Proposition 5.1, we have: $L(\boldsymbol{\mathcal{S}}) \subseteq L(\mathbf{S})$

$\Rightarrow s\sigma \in L(\mathbf{S})$ *by* (2) *and Proposition* 5.1 (4)

$\Rightarrow s\sigma \in L(\mathbf{G})$ *by* (1-4)

Hence, we conclude that $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$. □

5.4.2 S-Singular Prohibitable Behaviour

One of the assumptions made in the SD setting is that controllers allow prohibitable events to occur only once per sampling period. This should be reflected in the plant model as well. Hence, plant $\mathbf{G}$ is required to satisfy $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour with respect to supervisor $\boldsymbol{\mathcal{S}}$ in the SD setting. The following proposition proves that if $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$ with respect to supervisor $\mathbf{S}$ in the $||_{SD}$ setting, then $\mathbf{G}$ is guaranteed to have $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour with respect to supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S}||_{SD}\,\mathbf{G}$ in the SD setting.

Proposition 5.5. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, and TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor. If $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, then $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour.

Proof. Assume: $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, and $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$ (1)

To show that $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, it is sufficient to show:

$$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp})\ (\forall s' \in \Sigma^*_{act})\ ss' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \Rightarrow$$
$$(\forall \sigma \in Occu(s') \cap \Sigma_{hib})\ \sigma \notin Elig_{L(\mathbf{G})}(ss')$$

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp}$, and let $s' \in \Sigma^*_{act}$. Assume: $ss' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ (2)

Let $\sigma \in Occu(s') \cap \Sigma_{hib}$. Must show: $\sigma \notin Elig_{L(\mathbf{G})}(ss')$ (3)

By (2), we have: $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp}$

$\Rightarrow s \in L(\mathbf{S}||_{SD}\,\mathbf{G}) \cap L_{samp}$ *by* (1) *and Corollary* 5.1(v) (4)

By (2), we have: $ss' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

$\Rightarrow ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ *by* (1) *and Corollary* 5.1(v) (5)
$\Rightarrow \sigma \notin Elig_{L(\mathbf{G})}(ss')$ *by* (1-5)
Hence, we conclude that $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour. □

5.4.3 Timed Controllability

In the SD setting, supervisor TDES is assumed to be timed controllable with respect to plant TDES. The following proposition proves that while designing supervisor $\mathbf{S}$ of the $||_{SD}$ setting, if designers make sure that $\mathbf{S}$ is timed controllable with $||_{SD}$ with respect to plant $\mathbf{G}$, this guarantees that supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is timed controllable with respect to $\mathbf{G}$ in the SD setting.

Proposition 5.6. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor, and let $\Sigma_{for} = \Sigma_{hib}$. If $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$, then $\boldsymbol{\mathcal{S}}$ is timed controllable for $\mathbf{G}$.

Proof. Let $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, and $\Sigma_{for} = \Sigma_{hib}$. (1)
Assume: $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$ (Definition 4.4) (2)
Must show: $\boldsymbol{\mathcal{S}}$ is timed controllable for $\mathbf{G}$
Substituting (1) in Definition 2.22 of timed controllability, it is sufficient to show:
$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}))$

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases} \quad (3)$$

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.
$\Rightarrow s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ *by* (1) *and Corollary* 5.1(v) (4)
As $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ by (1), applying Corollary 5.1(v) on the R.H.S of (3), we get:

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases} \quad (5)$$

By (4) and (5), we thus have $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and:

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$$

This is true by our assumption of (2), as s is an arbitrary string.
Hence, we conclude that $\boldsymbol{\mathcal{S}}$ is timed controllable for $\mathbf{G}$. □

5.4.4 SD Controllability

One of the most important assumptions made by the authors while proving controllability and nonblocking verification results in the SD setting is that the supervisor TDES is SD controllable with respect to the plant TDES. The proposition given below provides sufficient conditions to automatically satisfy this property in the SD setting. It proves that in the $||_{SD}$ setting, if designers create a supervisor $\mathbf{S}$ that is SD controllable with $||_{SD}$ with respect to plant $\mathbf{G}$, then supervisor $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is guaranteed to be SD controllable with respect to $\mathbf{G}$ in the SD setting.

Proposition 5.7. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor, and let $\Sigma_{for} = \Sigma_{hib}$. If $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$, then $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$.

Proof. Let $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, and $\Sigma_{for} = \Sigma_{hib}$ (1)

Assume: $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$ (Definition 4.6) (2)

Must show $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$. By Definition 3.7 of SD controllability, it is sufficient to show the following:

$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}))$

i) $Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\boldsymbol{\mathcal{S}})}(s)$

ii) If $\tau \in Elig_{L(\mathbf{G})}(s)$, then $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s) \Leftrightarrow Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

iii) If $s \in L_{samp}$ then

1) $(\forall s' \in \Sigma^*_{act})\,[ss' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})] \Rightarrow$
$[Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(ss') \cup Occu(s')] \cap \Sigma_{hib} = Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib}$

2) $(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \wedge Occu(s') = Occu(s'')] \Rightarrow$
$ss' \equiv_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})} ss'' \wedge ss' \equiv_{L_m(\boldsymbol{\mathcal{S}})\cap L_m(\mathbf{G})} ss''$

iv) $L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G}) \subseteq L_{samp}$

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.
$\Rightarrow s \in L(\mathbf{S} \,||_{SD}\, \mathbf{G})$ *by* (1) *and Corollary* 5.1(v) (3)

Now we will analyze the four points of the SD controllability definition individually.

i) To show Point i, we need to show: $Elig_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq Elig_{L(\boldsymbol{\mathcal{S}})}(s)$
This concept can be restated as: $Elig_{L(\boldsymbol{\mathcal{S}})}(s) \supseteq Elig_{L(\mathbf{G})}(s) \cap \Sigma_u$ (4)
In the next step, we will combine this with Part(a) of Point ii, and show this matches Point i of Definition 4.6, and is thus satisfied by (2).

ii) Point ii of the SD controllability definition represents an "if and only if" statement. We will analyze it in two parts.
If $\tau \in Elig_{L(\mathbf{G})}(s)$ then:

a) Reverse implication ($\Leftarrow$): $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s) \Leftarrow Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

b) Forward implication ($\Rightarrow$): $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s) \Rightarrow Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

Part a) The reverse implication can be restated as:

$$Elig_{L(\boldsymbol{\mathcal{S}})}(s) \supseteq Elig_{L(\mathbf{G})}(s) \cap \{\tau\} \;\text{ if } Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \tag{5}$$

Combining (4) and (5), we get:

$$Elig_{L(\boldsymbol{\mathcal{S}})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases}$$

Applying (1) on the L.H.S., and (1) and Corollary 5.1(v) on the R.H.S, we get:

$$Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \supseteq \begin{cases} Elig_{L(\mathbf{G})}(s) \cap (\Sigma_u \cup \{\tau\}) & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset \\ Elig_{L(\mathbf{G})}(s) \cap \Sigma_u & \text{if } Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \neq \emptyset \end{cases} \tag{6}$$

As this now matches Point i of Definition 4.6, it is satisfied by (2).

Part b) The forward implication says that if *tick* is possible in $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, then no prohibitable events are possible after string s in $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$.

From (3), we have: $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ and $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$

We now need to show: $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s) \Rightarrow Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

Assume: $\tau \in Elig_{L(\boldsymbol{\mathcal{S}})}(s)$

$\Rightarrow \tau \in Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \quad as\ \boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}\ by\ (1)$ (7)

We now need to show this implies: $Elig_{L(\boldsymbol{\mathcal{S}})\cap L(\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

By (1) and Corollary 5.1(v), it is sufficient to show: $Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} = \emptyset$

As we have $\tau \in Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s)$ by (7), this follows automatically from Point i of Definition 4.1 of the $||_{SD}$ operator. (8)

Combining with Point i and Part(a) of Point ii, we now have satisfied both Points i and ii of the SD controllability definition.

iii) From Corollary 5.1(v,vi), we have:

$$L(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \quad \text{and} \quad L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$$

We can thus rewrite Point iii of Definition 3.7 using these identities as follows:

If $s \in L_{samp}$ then (9)

1) $(\forall s' \in \Sigma^*_{act})\,[ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})] \Rightarrow$

$$[Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(ss') \cup Occu(s')] \cap \Sigma_{hib} = Elig_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}(s) \cap \Sigma_{hib} \tag{10}$$

2) $(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \wedge Occu(s') = Occu(s'')] \Rightarrow$

$$ss' \equiv_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})}\ ss'' \wedge ss' \equiv_{L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})}\ ss'' \tag{11}$$

As this now exactly matches Point ii of Definition 4.6, it is satisfied by (2).

iv) From Corollary 5.1(vi), we have: $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})$

We can thus rewrite Point iv of Definition 3.7 as: $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \subseteq L_{samp}$ (12)

As this now exactly matches Point iii of Definition 4.6, it is satisfied by (2).

Combining (3), (6) and (8-12), we have shown that Points (i-iv) of the SD controllability definition are satisfied for $\boldsymbol{\mathcal{S}}$ and $\mathbf{G}$, as required.

Hence, we conclude that $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$. □

5.4.5 ALF

In order to show our equivalence result with respect to the ALF property, we will make use of one of the propositions from [42]. Proposition 5.8 stated below says that the synchronous product of two TDES will be ALF, if one TDES is ALF, and the ALF TDES contains all events in the event set of the other TDES.

Proposition 5.8. [42] Let $\mathbf{G}_1 = (Q_1, \Sigma_1, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma_2,$ $\delta_2, q_{o,2}, Q_{m,2})$ be two TDES. If $\mathbf{G}_1$ is ALF and $\Sigma_1 \supseteq \Sigma_2$, then $\mathbf{G}_1\,||\,\mathbf{G}_2$ is also ALF.

In the SD setting, one of the preconditions of the controllability and nonblocking verification results is that the closed-loop system constructed by synchronizing the plant and supervisor models using the synchronous product is ALF. In order to automatically satisfy this condition of the SD setting, we require that the closed-loop system constructed as $\mathbf{S}\,||_{SD}\,\mathbf{G}$ in the $||_{SD}$ setting must be ALF. This is because if $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF, then the closed loop

system constructed as $\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}$ in the SD setting is guaranteed to be ALF by Proposition 5.8 (as both $\boldsymbol{\mathcal{S}}$ and $\mathbf{G}$ are defined over the same event set (Section 5.2)). Please note that in Section 4.6, we have already presented an easy and modular way of making $\mathbf{S} ||_{SD} \mathbf{G}$ ALF.

6 Equivalence using Minimal Automaton

In this section, we present some more results with respect to establishing equivalence between the SD and our $||_{SD}$ setting. The primary focus of this section is on describing the approach that we have formulated to process TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$ (if required) and ensure that $\boldsymbol{\mathcal{S}}$ satisfies the property of concurrent string (CS) deterministic supervisors, as required by the supervisor of the SD setting.

This section begins with a discussion on why supervisor $\boldsymbol{\mathcal{S}}$ needs to be CS deterministic, and the significance of minimizing $\boldsymbol{\mathcal{S}}$. Then, we present our algorithms to obtain the minimal version of $\boldsymbol{\mathcal{S}}$ from its non-minimal TDES automaton. After that, we identify sufficient conditions and formally prove that minimized $\boldsymbol{\mathcal{S}}$ is guaranteed to be CS deterministic. Finally, we finish this section off by revisiting and re-evaluating our equivalence results presented in the previous section to make sure that they remain valid with the minimal version of $\boldsymbol{\mathcal{S}}$ as well.

6.1 Why Minimal Automaton is Needed?

The SD supervisory control methodology (Section 3) presents a formal translation method to translate a TDES supervisor into an SD controller. This translation process requires that the TDES supervisor must be CS deterministic (Definition 3.5). Otherwise, this conversion technique is not guaranteed to work. Since we are defining a concrete way to automatically construct TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$ that we intend to use as the supervisor of the SD setting, we need $\boldsymbol{\mathcal{S}}$ to be CS deterministic.

We also want to make $\boldsymbol{\mathcal{S}}$ CS deterministic because in the SD setting, the developed translation method is used to convert the CS deterministic TDES supervisor into an SD controller (in fact, the controller would otherwise be non-deterministic). This CS deterministic supervisor and its corresponding SD controller have then been used in the SD setting as the basis to conclude various SD controllability and nonblocking verification results. As we want to make these existing SD verification results valid in our $||_{SD}$ setting, and use them to derive and conclude our controllability and nonblocking verification results of the $||_{SD}$ setting, we must make sure that all preconditions of the SD verification results are satisfied.

Moreover, one of the goals of defining our $||_{SD}$ setting is to enable the software and hardware practitioners to design and implement our TDES supervisor $\mathbf{S}$ instead of the potentially more complex supervisor of the SD setting. In order to be able to do that, we need to show that the SD controller generated by translating $\mathbf{S}$ in our $||_{SD}$ setting is output equivalent (Definition 7.2) to the SD controller that is obtained by converting $\boldsymbol{\mathcal{S}}$ in the SD setting (this is demonstrated in Section 7). To theoretically prove this equivalence, we assume and require that the two supervisors $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}$ have been translated into their corresponding SD controllers. For this reason, both supervisors must be CS deterministic, as their translation into SD controllers is not possible otherwise.

In our $||_{SD}$ setting, since we want practitioners to design and implement our TDES supervisor $\mathbf{S}$, therefore we require them to design $\mathbf{S}$ in such a way that it must satisfy the property of CS deterministic supervisor. However, making $\mathbf{S}$ CS deterministic does not guarantee that $\boldsymbol{\mathcal{S}} = \mathbf{S} ||_{SD} \mathbf{G}$ will be CS deterministic. This is owing to the fact that in order to construct $\boldsymbol{\mathcal{S}}$,

$\mathbf{S}$ needs to be synchronized with TDES plant $\mathbf{G}$ using $||_{SD}$ operator, and neither $\mathbf{G}$ nor the $||_{SD}$ operator guarantees to preserve the property of CS deterministic supervisor in any way. This means if $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ is not CS deterministic, then we need to somehow process $\boldsymbol{\mathcal{S}}$ to make it CS deterministic.

Our approach of making $\boldsymbol{\mathcal{S}}$ CS deterministic relies on generating its minimal version. As we are proposing a strategy of obtaining a CS deterministic version of $\boldsymbol{\mathcal{S}}$, it is also important to show that $\boldsymbol{\mathcal{S}}$ will indeed become CS deterministic after applying our state space minimization algorithms (presented in the next section), and satisfying some other conditions. We formally prove this in Section 6.3.1.

In summary, if $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ is CS deterministic in its original form, we can directly use it as the supervisor of the SD setting and generate its corresponding SD controller. However, if $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ is not CS deterministic in its current form, then $\boldsymbol{\mathcal{S}}$ must be minimized using our state space minimization algorithms to make it CS deterministic, and essentially make it work within our setting for use in our proofs. In practice, we would never need to actually minimize $\boldsymbol{\mathcal{S}}$, as once we have proven equivalence, we would just implement $\mathbf{S}$.

6.2 Obtaining a Minimal Automaton

In this section, we present our approach to *minimize* a given TDES automaton, i.e. obtain an equivalent TDES automaton that has as few states as possible as any automaton accepting the same closed and marked languages. This minimal TDES automaton is unique for the given language up to relabelling of states.

Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a TDES automaton. Without any loss of generality, we assume that $\mathbf{G}$ is a reachable automaton. We will describe our approach of obtaining the minimal version of TDES automaton $\mathbf{G}$ in two steps: 1) identify distinct λ-equivalent states of $\mathbf{G}$, and 2) construct minimal TDES automaton $\mathbf{G}'$. We elaborate these two steps and present their corresponding algorithms in the following two subsections. Please note that these algorithms are generic (not specific to our $\boldsymbol{\mathcal{S}}$) and can be used to generate the minimal version of any given TDES automaton.

6.2.1 Identify Distinct $\lambda-$Equivalent States

Our Algorithm 2 identifies distinct λ-equivalent states of a generator $\mathbf{G}$ (where δ is a partial function). The algorithm begins by unflagging all state pairs at **Step 0**. We have added this step just to ensure the accuracy of our results. Our approach to find all possible sets of λ-equivalent states of $\mathbf{G}$, is by flagging state pairs that are not λ-equivalent at **Steps 1-4**. It is notable that as the relation λ is symmetric, for states $q_1, q_2 \in Q$, if we flag state pair (q_1, q_2), we must also flag pair (q_2, q_1).

At **Step 1**, we flag every state pair such that one state of the pair is marked and the other state is unmarked, as marked and unmarked states are not λ-equivalent. **Step 2** is performed for every remaining unflagged state pair $(q_1, q_2) \in Q \times Q$. At **Step 2.1**, we look for some event $\sigma \in \Sigma$, such that σ is defined at exactly one state of the state pair, i.e. either at q_1 or q_2. If such σ exists, we flag state pairs (q_1, q_2) and (q_2, q_1) (**Step 2.1.1**). This is because q_1 and q_2 are not λ-equivalent, as they have different sets of σ transitions leaving them.

At **Step 3**, we initialize our boolean variable $flagging$ to $True$. **Step 4** is repeated as long as $flagging$ is $True$, i.e. there is a possibility to flag more state pairs. At **Step 4.1**, we set $flagging$ to $False$ by assuming that no more state pairs could be flagged in the current

Algorithm 2 Identify Distinct λ-Equivalent States of Generator $\mathbf{G}$

Step 0: For every pair $(q_1, q_2) \in Q \times Q$, unflag (q_1, q_2).
Step 1: For every pair $(q_1, q_2) \in Q \times Q$, if $(q_1 \in Q_m \wedge q_2 \notin Q_m) \vee (q_1 \notin Q_m \wedge q_2 \in Q_m)$, then:
  Step 1.1: Flag $(q_1, q_2), (q_2, q_1)$.
Step 2: For every pair $(q_1, q_2) \in Q \times Q$ not flagged at Step 1:
  Step 2.1: For some $\sigma \in \Sigma$, if $(\delta(q_1, \sigma)! \wedge \neg\delta(q_2, \sigma)!) \vee (\neg\delta(q_1, \sigma)! \wedge \delta(q_2, \sigma)!)$, then:
    Step 2.1.1: Flag $(q_1, q_2), (q_2, q_1)$.
Step 3: Set $flagging := True$.
Step 4: While $(flagging)$:
  Step 4.1: Set $flagging := False$.
  Step 4.2: For every pair $(q_1, q_2) \in Q \times Q$ not flagged at Steps 1 and 2:
    Step 4.2.1: For some $\sigma \in \Sigma$ such that $\delta(q_1, \sigma)! \wedge \delta(q_2, \sigma)!$, if $(\delta(q_1, \sigma), \delta(q_2, \sigma))$ is flagged, then:
      Step 4.2.1.1: Flag $(q_1, q_2), (q_2, q_1)$.
      Step 4.2.1.2: Set $flagging := True$.
Step 5: Add all unflagged, non-singular pairs (no pairs $(q, q) \in Q \times Q$) to list $\mathbf{L}$.
Step 6: Set $k := 0$.
Step 7: While $\mathbf{L} \neq \emptyset$:
  Step 7.1: Set $k := k + 1$.
  Step 7.2: Take a pair (q_1, q_2) from $\mathbf{L}$. Create a new set $\mathbf{E}_k$ and add both states q_1 and q_2 of the pair to $\mathbf{E}_k$. Remove all occurrences of the pair (q_1, q_2) and (q_2, q_1) from $\mathbf{L}$.
  Step 7.3: For every pair (q_1', q_2') in $\mathbf{L}$, if the pair has exactly one state in common with $\mathbf{E}_k$, then add the uncommon state of the pair to $\mathbf{E}_k$. Remove all occurrences of of the pair (q_1', q_2') and (q_2', q_1') from $\mathbf{L}$. Then, repeat this step until no pair in $\mathbf{L}$ has exactly one state in common with $\mathbf{E}_k$.

iteration. However, if we are able to flag more state pairs, then we set $flagging$ to $True$ again (**Step 4.2.1.2**) to repeat **Step 4** one more time. This is because there is a possibility that flagging might propagate from the recently flagged state pairs to some unflagged state pair(s) in the next iteration. However, if we do not flag any state pairs in the current iteration of **Step 4**, $flagging$ remains $False$, and while loop of **Step 4** terminates.

Step 4.2 is performed for every unflagged state pair $(q_1, q_2) \in Q \times Q$. At **Step 4.2.1**, we check to see if there is some event σ, such that σ is defined at both q_1 and q_2, and σ leads them to a state pair that is flagged. If so, we flag (q_1, q_2) and (q_2, q_1) (**Step 4.2.1.1**). The reason is that σ takes q_1 and q_2 to some destination states that are not λ-equivalent. Once **Step 4** finishes, the flagging process is complete and the state pairs that are not flagged correspond to states that are λ-equivalent.

At **Step 5**, we create a list $\mathbf{L}$, and add all non-singular (a state pair with distinct states) unflagged state pairs to $\mathbf{L}$. This means that if a state is only λ-equivalent to itself, then $(q, q) \in Q \times Q$ will not be added to $\mathbf{L}$. Only unflagged state pairs $(q_1, q_2) \in Q \times Q$, with $q_1 \neq q_2$, will be added to $\mathbf{L}$. At **Step 6**, we initialize our counter variable k to 0, and increment it by 1 (**Step 7.1**) every time we construct a new set of λ-equivalent states, E_k.

At **Step 7**, we use the list $\mathbf{L}$ to form disjoint sets of λ-equivalent states in such a way that each state is exactly in one set, all states in the same set are λ-equivalent, and no two states

from different sets are λ-equivalent. These sets will thus contain at least two (and possibly more) distinct λ-equivalent states that need to be combined. We use the *transitive property* (i.e. if $x \equiv y$ and $y \equiv z$, then $x \equiv z$) of the λ-equivalence relation to form these sets. **Step 7** is repeated until $\mathbf{L}$ becomes empty.

At **Step 7.2**, we create a new set $\mathbf{E}_k$ by removing one state pair (q_1, q_2) from $\mathbf{L}$, and adding both states of the pair to $\mathbf{E}_k$. As these two states of the pair are λ-equivalent, they must be in the same set. We then remove all occurrences of (q_1, q_2) and (q_2, q_1) from $\mathbf{L}$. This ensures that each state pair is added to only one set exactly once, and guarantees that all sets of λ-equivalent states are disjoint.

At **Step 7.3**, we check to see if there exists a state pair (q_1', q_2') in $\mathbf{L}$ that has exactly one state in common with $\mathbf{E}_k$. If yes, this means the common state is λ-equivalent to all other states of $\mathbf{E}_k$. As two states of the pair are λ-equivalent, this step adds the uncommon state of the pair to $\mathbf{E}_k$ as well. This ensures that all states in the same set are λ-equivalent. As both states of this pair have now been added to the appropriate set, we remove all occurrences of (q_1', q_2') and (q_2', q_1') from $\mathbf{L}$. This step is repeated until there does not exist any state pair in $\mathbf{L}$ that has exactly one state in common with $\mathbf{E}_k$. It is notable that if no state of the pair is in common with $\mathbf{E}_k$, then the states of the pair are not λ-equivalent to the states of $\mathbf{E}_k$. In this case, they must not be added to $\mathbf{E}_k$, as only λ-equivalent states must be in the same set.

After **Step 7.3** is complete for set $\mathbf{E}_k$, there is no state pair in $\mathbf{L}$ that has one or more states in common with $\mathbf{E}_k$. For other state pairs that are in $\mathbf{L}$ but not λ-equivalent to the states of $\mathbf{E}_k$, we repeat **Step 7** and create new sets, as needed.

Upon completion, Algorithm 2 creates one or more disjoint sets of λ-equivalent states of the input TDES automaton $\mathbf{G}$, if $\mathbf{G}$ was not minimal. However, if $\mathbf{G}$ was already in its minimal form, our algorithm will flag all state pairs at **Steps 1-4**, as no two distinct states of $\mathbf{G}$ are λ-equivalent. In this case, there will be no non-singular (i.e. no pairs $(q, q) \in Q \times Q$) unflagged pairs to be added to list $\mathbf{L}$ at **Step 5**. As $\mathbf{L}$ is empty, **Step 7** is not executed and no sets of λ-equivalent states will be formed by the algorithm.

6.2.2 Construct a Minimal Automaton

A TDES automaton is said to be *minimal* (Definition 2.15) if it does not have two distinct states that are λ-equivalent. This means in order to obtain a minimal version of a non-minimal TDES, all distinct λ-equivalent states of the non-minimal automaton should be merged and replaced by a single "aggregate" state. This process is called *state aggregation* [12]. For example, if the non-minimal TDES automaton $\mathbf{G}$ has $n > 1$ distinct λ-equivalent states $q_1, \ldots, q_n \in Q$, these n states should be replaced by a single aggregate state, say q, in the minimal TDES automaton, such that q behaves like $q_1, \ldots, q_n$. There can be one or more groups of distinct λ-equivalent states in the non-minimal automaton. The minimal automaton will have an aggregate state corresponding to each one of these groups.

Let TDES automaton $\mathbf{G}' = (Q', \Sigma, \delta', q_o', Q_m')$ be the minimum-state version of the non-minimal TDES automaton $\mathbf{G}$. Here, the state space Q' represents the smallest set of states after combining all states within each group of distinct λ-equivalent states of $\mathbf{G}$. Σ is the event set of $\mathbf{G}'$ and is same as the event set of $\mathbf{G}$. δ' is the resulting transition function, $q_o' \in Q'$ is the initial state, and $Q_m' \subseteq Q'$ is the set of marked states of the minimal automaton $\mathbf{G}'$. To clearly argue about the transitions of $\mathbf{G}$ and $\mathbf{G}'$, we will express transitions as a 3-tuple (as described in Section 2.2.1).

By utilizing the disjoint sets of distinct λ-equivalent states of $\mathbf{G}$ identified by Algorithm

Algorithm 3 Construct Minimal TDES Automaton $\mathbf{G}'$ from $\mathbf{G}$

Step 1: $\mathbf{G}' := \mathbf{G}$, such that $Q' := Q, \Sigma := \Sigma, \delta' := \delta, q'_o := q_o, Q'_m := Q_m$.
Step 2: For every set $\mathbf{E}_k$ of distinct λ-equivalent states of $\mathbf{G}$:
  Step 2.1: For all $q \in \mathbf{E}_k$, remove q from Q'.
  Step 2.2: Add q' to Q', such that $q' \notin Q$ and $q' \notin Q'$.
  Step 2.3: If $q_o \in \mathbf{E}_k$, then $q'_o := q'$.
  Step 2.4: If $(\mathbf{E}_k \cap Q_m \neq \emptyset)$, then:
    Step 2.4.1: For all $q'' \in \mathbf{E}_k$, remove q'' from Q'_m.
    Step 2.4.2: Add q' to Q'_m.
  Step 2.5: For every transition $(q_1, \sigma, q_2) \in \delta'$:
    Step 2.5.1: If $q_1 \in \mathbf{E}_k$, then replace q_1 with q' in the transition triple in δ'.
    Step 2.5.2: If $q_2 \in \mathbf{E}_k$, then replace q_2 with q' in the transition triple in δ'.

2, Algorithm 3 presents steps for the iterative construction of minimal automaton $\mathbf{G}'$. The algorithm begins by copying the non-minimal automaton $\mathbf{G}$ to $\mathbf{G}'$. **Step 1** copies the state set Q to Q', event set Σ to Σ, transition function δ to δ', initial state q_o to q'_o, and the set of final states Q_m to Q'_m. At **Step 2**, we iteratively update the automaton structure of $\mathbf{G}'$ to make it minimal. This step is repeated for each set of λ-equivalent states $\mathbf{E}_k$, where $1 \leq k \leq t$ and $t \geq 1$ is the total number of sets of distinct λ-equivalent states formed by Algorithm 2.

Steps 2.1 and **2.2** merge all λ-equivalent states of set $\mathbf{E}_k$ and replace them with a single aggregate state in $\mathbf{G}'$. In other words, we remove all distinct λ-equivalent states of $\mathbf{E}_k$ from Q' and add one state, q', corresponding to $\mathbf{E}_k$ in Q'. It is important to make sure that the state label q' does not already exist in Q or Q'. At **Step 2.3**, we check to see if $\mathbf{E}_k$ contains the initial state of $\mathbf{G}$. If so, we make q' the initial state of $\mathbf{G}'$. The set of marked states of $\mathbf{G}'$ should include all aggregate states that correspond to sets that contain the marked states of $\mathbf{G}$. At **Step 2.4**, we determine if E_k contains any marked state. If so, all the λ-equivalent states of $\mathbf{E}_k$ are removed from Q'_m and replaced by the corresponding aggregate state q'.

At **Step 2.5**, we perform relabelling of λ-equivalent states of $\mathbf{E}_k$ in the transitions of δ'. This is required because all the distinct λ-equivalent states of set $\mathbf{E}_k$ have been replaced by a single aggregate state in $\mathbf{G}'$. Therefore, all the transitions, copied from δ to δ' at **Step 1**, that have these λ-equivalent states as their exit and/or entrance states should now have the corresponding aggregate state q' as their exit and/or entrance states respectively in δ'.

It is important to clarify here that **Step 2.5** does not add or remove any transitions from δ'. It just relabels the exit and/or entrance states of transitions in δ' by replacing the state labels of λ-equivalent states with their corresponding aggregate state labels. In other words, we can say that δ' is essentially δ, with the distinct λ-equivalent states of $\mathbf{G}$ being replaced by their corresponding aggregate states in $\mathbf{G}'$. In this way, every iteration of **Step 2** updates the automaton structure of $\mathbf{G}'$ to make it minimal.

It is noteworthy that Algorithm 3 does not make any changes to states that are identified by Algorithm 2 as not being λ-equivalent. At **Step 1**, Algorithm 3 copies the entire automaton structure of $\mathbf{G}$ to $\mathbf{G}'$. Thus, these non-λ-equivalent states and their transitions are a part of $\mathbf{G}'$ and remain unchanged throughout the execution of **Step 2**, as they do not belong to any set $\mathbf{E}_k$. Therefore, the automaton structure of $\mathbf{G}'$ with respect to these non-λ-equivalent states is the same as $\mathbf{G}$.

Once Algorithm 3 completes its execution, $\mathbf{G}'$ will have as few states as any automaton

accepting the same closed and marked language as $\mathbf{G}$. In other words, $\mathbf{G}'$ now represents the minimal version of $\mathbf{G}$.

6.3 SD Properties with Minimal Automata

In this section, we discuss our equivalence results for all SD properties with respect to replacing $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ with a minimal version of $\boldsymbol{\mathcal{S}}$, referred to as $min(\boldsymbol{\mathcal{S}})$ (i.e. the result of applying Algorithms 2 and 3 to TDES $\boldsymbol{\mathcal{S}}$). We would need to do this if $\boldsymbol{\mathcal{S}}$ is not CS deterministic in its current form, and would use $min(\boldsymbol{\mathcal{S}})$ to address this (Section 6.3.1). If we make this change, we will need to re-evaluate our previous equivalence results from Section 5 with respect to $min(\boldsymbol{\mathcal{S}})$, and present new results for the property of CS deterministic supervisors with $min(\boldsymbol{\mathcal{S}})$.

In order to assess our previous results with respect to replacing $\boldsymbol{\mathcal{S}}$ by $min(\boldsymbol{\mathcal{S}})$, we first note that our equivalence results of the SD and $||_{SD}$ setting for language equivalence (Section 5.3), plant completeness (Section 5.4.1), $\mathbf{S}$-singular prohibitable behaviour (Section 5.4.2), timed controllability (Section 5.4.3) and SD controllability (Section 5.4.4) are all proved in terms of the closed and/or marked languages of the involved TDES, and not the actual automaton structure. As the state space minimization Algorithms 2 and 3 produce minimal automaton with the same closed and marked languages as the original, i.e. $L(min(\boldsymbol{\mathcal{S}})) = L(\boldsymbol{\mathcal{S}})$ and $L_m(min(\boldsymbol{\mathcal{S}})) = L_m(\boldsymbol{\mathcal{S}})$, it thus follows that the results from Sections 5.3 and 5.4.1-5.4.4 remain valid if we replace $\boldsymbol{\mathcal{S}}$ by $min(\boldsymbol{\mathcal{S}})$. As a result, we do not need to adapt or reprove these results.

The only definition that is given in terms of the states of TDES automaton is the definition of ALF (Definition 2.30). Since we intend to minimize $\boldsymbol{\mathcal{S}}$ by merging various groups of distinct λ-equivalent states, the state space of $min(\boldsymbol{\mathcal{S}})$ will be different than the non-minimal $\boldsymbol{\mathcal{S}}$. This implies that while talking about $min(\boldsymbol{\mathcal{S}})$, we can no longer argue in terms of the states and state tuples of $\boldsymbol{\mathcal{S}}$. Hence, we will revisit our ALF equivalence results (discussed in Section 5.4.5) later in this section to make them work with $min(\boldsymbol{\mathcal{S}})$. However, we will first describe our new CS deterministic result with respect to $\boldsymbol{\mathcal{S}}$ and $min(\boldsymbol{\mathcal{S}})$.

6.3.1 CS Deterministic Supervisors

Our ultimate goal of generating the minimal version of $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ is to make it CS deterministic, if it is not already. However, minimizing $\boldsymbol{\mathcal{S}}$ alone does not guarantee that $min(\boldsymbol{\mathcal{S}})$ will always be CS deterministic. We also need to make sure that our TDES supervisor $\mathbf{S}$ is SD controllable with $||_{SD}$ for TDES plant $\mathbf{G}$ to guarantee that $min(\boldsymbol{\mathcal{S}})$ is CS deterministic. This is proved in our next proposition (Proposition 6.2). In order to prove our desired result, we will use Proposition 6.1 from [45]. This proposition says that for a given TDES $\mathbf{G}$, two strings s and s' are nerode equivalent with respect to $L(\mathbf{G})$ and $L_m(\mathbf{G})$ if and only if both of these strings start from the initial state and take us to states that are λ-equivalent.

Proposition 6.1. [45] For a generator $\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$, we have $(\forall s, s' \in \Sigma^*)\, \eta(y_o, s) \equiv \eta(y_o, s') \,(\text{mod } \lambda) \Leftrightarrow s \equiv_{L(\mathbf{G})} s' \wedge s \equiv_{L_m(\mathbf{G})} s'$.

Proposition 6.2. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G}) = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor, where $min(\mathbf{S} \,||_{SD}\, \mathbf{G})$ is constructed using Algorithms 2 and 3. If $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$, then $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Proof. Assume initial conditions.

Must show: $\boldsymbol{\mathcal{S}}$ is CS deterministic. By Definition 3.5, it is sufficient to show:

$$(\forall s \in L(\boldsymbol{\mathcal{S}}) \cap L_{samp})\,(\forall s', s'' \in L_{conc})\,[ss', ss'' \in L(\boldsymbol{\mathcal{S}}) \wedge Occu(s') = Occu(s'')] \Rightarrow$$
$$[ss' \equiv_{L(\boldsymbol{\mathcal{S}})} ss'' \wedge ss' \equiv_{L_m(\boldsymbol{\mathcal{S}})} ss'' \wedge \eta(y_o, ss') = \eta(y_o, ss'')]$$

We have $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$, and thus $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}}) = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$. (1)

Let $s \in L(\boldsymbol{\mathcal{S}}) \cap L_{samp}$, and let $s', s'' \in L_{conc}$. (2)

By (1), this implies: $s \in L(\mathbf{S}||_{SD}\,\mathbf{G}) \cap L_{samp}$ (3)

Assume: $ss', ss'' \in L(\boldsymbol{\mathcal{S}})$ and $Occu(s') = Occu(s'')$ (4)

By (1), this implies: $ss', ss'' \in L(\mathbf{S}||_{SD}\,\mathbf{G})$ (5)

Must show this implies: $ss' \equiv_{L(\boldsymbol{\mathcal{S}})} ss'' \wedge ss' \equiv_{L_m(\boldsymbol{\mathcal{S}})} ss'' \wedge \eta(y_o, ss') = \eta(y_o, ss'')$

We have that $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$. By (2-5), we note that all assumptions of *Point ii.2* of the SD controllability with $||_{SD}$ definition are satisfied.

$\Rightarrow ss' \equiv_{L(\mathbf{S}\,||_{SD}\,\mathbf{G})} ss'' \wedge ss' \equiv_{L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})} ss''$

$\Rightarrow ss' \equiv_{L(\boldsymbol{\mathcal{S}})} ss'' \wedge ss' \equiv_{L_m(\boldsymbol{\mathcal{S}})} ss''$ *by* (1) (6)

$\Rightarrow \eta(y_o, ss') \equiv \eta(y_o, ss'')\,(\text{mod } \lambda)$ *by Proposition* 6.1

$\Rightarrow \eta(y_o, ss') = \eta(y_o, ss'')$ *by Definition* 2.15 *of minimal* $\boldsymbol{\mathcal{S}}$ (7)

By (6) and (7), we have thus shown that $\boldsymbol{\mathcal{S}}$ is CS deterministic. □

The above proposition tells us that as long as $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$, $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ will be CS deterministic. We note that if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is already minimal, then Algorithms 2 and 3 will not make any changes, and $\mathbf{S}\,||_{SD}\,\mathbf{G} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$. This implies that $\mathbf{S}\,||_{SD}\,\mathbf{G}$ will be CS deterministic in this case. However, if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is not minimal, then we just take $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$, and we have a CS deterministic supervisor in both cases.

As discussed in Section 5.1, we intend to base our controllability and nonblocking verification results of the $||_{SD}$ setting on some of the existing results of the SD setting. To do this, we will need to construct an SD controller from $\boldsymbol{\mathcal{S}}$, a prerequisite of which is that $\boldsymbol{\mathcal{S}}$ must be CS deterministic. We now know that this will require considering $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is minimal, or $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is not minimal.

It is worth noting that in either case, both $\mathbf{S}\,||_{SD}\,\mathbf{G}$ and $min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ will have the same closed and marked languages, i.e. $L(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L(min(\mathbf{S}\,||_{SD}\,\mathbf{G}))$ and $L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L_m(min(\mathbf{S}\,||_{SD}\,\mathbf{G}))$. This in turn means that all equivalence results that are solely related to the closed and marked languages remain applicable to both $\mathbf{S}\,||_{SD}\,\mathbf{G}$ and $min(\mathbf{S}\,||_{SD}\,\mathbf{G})$.

However, whether we use $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ or $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ will affect our argument about defining the states of $\boldsymbol{\mathcal{S}}$ in terms of the states of $\mathbf{S}$ and $\mathbf{G}$. Precisely, if $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, then state $y \in Y$ of $\boldsymbol{\mathcal{S}}$ will be a cross product of the states $x \in X$ of $\mathbf{S}$ and $q \in Q$ of $\mathbf{G}$, i.e. $y = (x, q)$. But this might not be true if we minimize the automaton $\mathbf{S}\,||_{SD}\,\mathbf{G}$ and use it as our $\boldsymbol{\mathcal{S}}$, i.e. $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$. Therefore, in our future proofs, whenever we want to argue in terms of the states of $\boldsymbol{\mathcal{S}}$, we will consider two ways of constructing $\boldsymbol{\mathcal{S}}$ separately.

6.3.2 ALF

In order to keep our ALF result of Section 5.4.5 valid for $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$, we need to show that the ALF property is preserved by the TDES minimization process. Before we prove this, we first give three utility propositions. Their goal is to allow us to convert key information

(information about λ-equivalent states and their successors, converting transition in $\mathbf{G}'$ to transition in $\mathbf{G}$, and translating state reachability) from $\mathbf{G}' = min(\mathbf{G})$ to equivalent results about $\mathbf{G}$. This will be key in removing redundancies from later proofs to make them more compact and easier to read.

Please note that for a non-minimal TDES $\mathbf{G}$, Algorithm 2 will create $t \geq 1$ distinct sets of λ-equivalent states (Definition 2.14) of $\mathbf{G}$. For each such set E_k $(1 \leq k \leq t)$, Algorithm 3 will replace all instances of state $q \in E_k$ from $\mathbf{G}'$. Each instance would be replaced by a unique aggregate state q', such that $q' \notin Q$ and $q' \notin Q'$ (before the replacement). As each E_k is associated with a unique state q' by this replacement, we will refer to E_k as $E_{q'}$ in the following propositions to make it clear that the λ-equivalent states in $E_{q'}$ were replaced by q' when Algorithm 3 was executed and $\mathbf{G}'$ was constructed.

Proposition 6.3. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a non-minimal TDES and $\mathbf{G}' = min(\mathbf{G}) = (Q', \Sigma, \delta', q'_o, Q'_m)$ be the TDES constructed using Algorithms 2 and 3. Then:

i) $(\forall q_a, q_b \in Q)\, q_a \equiv q_b \,(\text{mod } \lambda) \Rightarrow (\forall s \in \Sigma^*)\, \delta(q_a, s)! \Rightarrow \delta(q_a, s) \equiv \delta(q_b, s) \,(\text{mod } \lambda)$

ii) $(\forall q', q'' \in Q')\, (\forall s \in \Sigma^*)\, \delta'(q', s) = q'' \Rightarrow (\exists\, q_a, q_b \in Q)\, \delta(q_a, s) = q_b \wedge (q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''})$

iii) $(\forall q_a, q_b \in Q)\, (\forall s \in \Sigma^*)\, \delta(q_a, s) = q_b \Rightarrow (\exists q', q'' \in Q')\, \delta'(q', s) = q'' \wedge (q' = q_a \vee q_a \in E_{q'}) \wedge (q'' = q_b \vee q_b \in E_{q''})$

Proof. Assume initial conditions.

i) Show: $(\forall q_a, q_b \in Q)\, q_a \equiv q_b \,(\text{mod } \lambda) \Rightarrow (\forall s \in \Sigma^*)\, \delta(q_a, s)! \Rightarrow \delta(q_a, s) \equiv \delta(q_b, s) (\text{mod } \lambda)$

Let $q_a, q_b \in Q$. Assume: $q_a \equiv q_b \,(\text{mod } \lambda)$ (1)

Let $s \in \Sigma^*$. Assume: $\delta(q_a, s)!$

$\Rightarrow \delta(q_b, s)!$ *by* (1)

By Definition 2.14, it is sufficient to show Parts (1) and (2) below.

Part 1) Show: $(\forall s' \in \Sigma^*)\, \delta(\delta(q_a, s), s')! \Leftrightarrow \delta(\delta(q_b, s), s')!$

By definition of δ, it is sufficient to show: $(\forall s' \in \Sigma^*)\, \delta(q_a, ss')! \Leftrightarrow \delta(q_b, ss')!$

This follows automatically from (1), Point 1 of the λ-equivalence definition, and the fact that $s, s' \in \Sigma^*$ implies $ss' \in \Sigma^*$.

Part 2) Show: $(\forall s' \in \Sigma^*)\, \delta(\delta(q_a, s), s')! \wedge \delta(\delta(q_a, s), s') \in Q_m \Leftrightarrow \delta(\delta(q_b, s), s')! \wedge \delta(\delta(q_b, s), s') \in Q_m$

By definition of δ, it is sufficient to show:

$$(\forall s' \in \Sigma^*)\, \delta(q_a, ss')! \wedge \delta(q_a, ss') \in Q_m \Leftrightarrow \delta(q_b, ss')! \wedge \delta(q_b, ss') \in Q_m$$

This follows automatically from (1), Point 2 of the λ-equivalence definition, and the fact that $s, s' \in \Sigma^*$ implies $ss' \in \Sigma^*$.

By Parts (1) and (2), we have proven **Part (i)**.

ii) Show: $(\forall q', q'' \in Q')\, (\forall s \in \Sigma^*)\, \delta'(q', s) = q'' \Rightarrow (\exists\, q_a, q_b \in Q)\, \delta(q_a, s) = q_b \wedge (q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''})$

Let $q', q'' \in Q'$ and $s \in \Sigma^*$. Assume: $\delta'(q', s) = q''$ (2)

To be consistent with Algorithm 3, we will treat $\delta \subseteq Q \times \Sigma \times Q$ as a relation, where $(q_1, \sigma, q_2) \in \delta$ if and only if $\delta(q_1, \sigma) = q_2$. Similarly, we will treat $\delta' \subseteq Q' \times \Sigma \times Q'$, where $(q'_1, \sigma, q'_2) \in \delta'$ if and only if $\delta'(q'_1, \sigma) = q'_2$.

As $s \in \Sigma^*$, we have two cases: **(1)** $s = \epsilon$, or **(2)** $s \in \Sigma^+$.

Case 1) $s = \epsilon$

As $\delta'(q', s) = q''$ by (2), this implies $q' = q''$. (3)

We now have two cases: **(a)** $q' \in Q$, or **(b)** $q' \notin Q$.

Case 1.a) $q' \in Q$

We can then take $q_a = q_b = q' = q''$, and we immediately have $\delta(q_a, \epsilon) = \delta(q_a, s) = q_a = q_b$.

Case 1.b) $q' \notin Q$

$\Rightarrow \exists q_a \in E_{q'}$, as $E_{q'}$ is not empty by Algorithm 2.

We thus have $q_a \in Q$ (by Algorithms 2 and 3), and can set $q_b = q_a$, and we have $\delta(q_a, \epsilon) = \delta(q_a, s) = q_a = q_b$.

As $q' = q''$ by (3), we have $E_{q'} = E_{q''}$, thus $q_b \in E_{q''}$ as $q_a = q_b$ and $q_a \in E_{q'}$.

By Cases (1.a) and (1.b), we have proven the desired condition for **Case (1)** ($s = \epsilon$).

Case 2) $s \in \Sigma^+$

Let $n = |s| \geq 1$.

$\Rightarrow (\exists \sigma_1, \sigma_2, \ldots, \sigma_n \in \Sigma)\, s = \sigma_1 \sigma_2 \ldots \sigma_n$

As $\delta'(q', s) = q''$ by (2), and $\delta' \subseteq Q' \times \Sigma \times Q'$, we can conclude there exists states $q'_1, q'_2, \ldots, q'_{n+1} \in Q'$ such that they form the following sequence of transitions in δ':

$$(q'_1, \sigma_1, q'_2), (q'_2, \sigma_2, q'_3), \ldots, (q'_n, \sigma_n, q'_{n+1}), \text{where } q'_1 = q' \text{ and } q'_{n+1} = q'' \tag{4}$$

By Algorithm 3, there exists states $q_1, q_2, \ldots, q_n, q_{n+1} \in Q$, and that $\delta \subseteq Q \times \Sigma \times Q$ contains the corresponding sequence of transitions:

$$\begin{gathered}(q_1, \sigma_1, q_2), (q_2, \sigma_2, q_3), \ldots, (q_n, \sigma_n, q_{n+1}), \\ \text{where for } 1 \leq i \leq n+1, q_i = q'_i \text{ or } q_i \in E_{q'_i}\end{gathered} \tag{5}$$

We thus have: $\delta(q_1, s) = q_{n+1}$

We can thus take $q_a = q$, and $q_b = q_{n+1}$, and we have $\delta(q_a, s) = q_b$, $q_a = q'$ or $q_a \in E_{q'}$, and $q_b = q''$ or $q_b \in E_{q''}$ by (4) and (5).

Case (2) complete.

By Cases (1) and (2), we have constructed suitable $q_a, q_b \in Q$ with properties:

$$\delta(q_a, s) = q_b \wedge (q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''})$$

Part (ii) complete.

iii) Show: $(\forall q_a, q_b \in Q)\, (\forall s \in \Sigma^*)\, \delta(q_a, s) = q_b \Rightarrow (\exists q', q'' \in Q')\, \delta'(q', s) = q'' \wedge (q' = q_a \vee q_a \in E_{q'}) \wedge (q'' = q_b \vee q_b \in E_{q''})$

Let $q_a, q_b \in Q$ and $s \in \Sigma^*$. Assume: $\delta(q_a, s) = q_b$ (6)

As $s \in \Sigma^*$, we have two cases: **(1)** $s = \epsilon$, or **(2)** $s \in \Sigma^+$.

Case 1) $s = \epsilon$

As $\delta(q_a, s) = q_b$ by (6), this implies: $q_a = q_b$ (7)

We now have two cases: **(a)** $q_a \in Q'$, or **(b)** $q_a \notin Q'$.

Case 1.a) $q_a \in Q'$

We can thus take $q' = q'' = q_a = q_b$, and we immediately have $\delta'(q', \epsilon) = q' = q''$.

Case 1.b) $q_a \notin Q'$

By Algorithms 2 and 3, this implies: $(\exists q' \in Q')\, q_a \in E_{q'}$ (8)

As $q_a = q_b$ by (7), we also set $q'' = q'$, and we have $q'' \in Q'$ with $q_b \in E_{q''}$.
As $q' \in Q'$ by (8), we have $\delta'(q', \epsilon) = q' = q''$.

By Cases (1.a) and (1.b), we have proven the desired condition for **Case (1)** ($s = \epsilon$).

Case 2) $s \in \Sigma^+$
Let $n = |s| \geq 1$.
$\Rightarrow (\exists \sigma_1, \sigma_2, \ldots, \sigma_n \in \Sigma)\, s = \sigma_1 \sigma_2 \ldots \sigma_n$
As $\delta(q_a, s) = q_b$ by (6), and $\delta \subseteq Q \times \Sigma \times Q$, we can conclude there exists states $q_1, q_2, \ldots, q_{n+1} \in Q$ such that they form the following sequence of transitions in δ:

$$(q_1, \sigma_1, q_2), (q_2, \sigma_2, q_3), \ldots, (q_n, \sigma_n, q_{n+1}), \text{where } q_1 = q_a \text{ and } q_{n+1} = q_b \tag{9}$$

By Algorithm 3, there exists states $q'_1, q'_2, \ldots, q'_n, q'_{n+1} \in Q'$, and that $\delta' \subseteq Q' \times \Sigma \times Q'$ contains the corresponding sequence of transitions:

$$\begin{gathered}(q'_1, \sigma_1, q'_2), (q'_2, \sigma_2, q'_3), \ldots, (q'_n, \sigma_n, q'_{n+1}), \\ \text{where for } 1 \leq i \leq n+1, q'_i = q_i \text{ or } q_i \in E_{q'_i}\end{gathered} \tag{10}$$

We thus have: $\delta'(q'_1, s) = q'_{n+1}$
We can thus take $q' = q'_1$, and $q'' = q'_{n+1}$, and by (9) and (10) we have:

$$(q', q'' \in Q') \wedge (\delta'(q', s) = q'') \wedge (q' = q_a \vee q_a \in E_{q'}) \wedge (q'' = q_b \vee q_b \in E_{q''})$$

Case (2) complete.

By Cases (1) and (2), we have constructed suitable $q', q'' \in Q'$ with the desired properties.

Part (iii) complete.

By Parts (i)-(iii), we conclude that Points (i-iii) of the proposition are satisfied. □

Proposition 6.4. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a non-minimal TDES and $\mathbf{G}' = min(\mathbf{G}) = (Q', \Sigma, \delta', q'_o, Q'_m)$ be the TDES constructed using Algorithms 2 and 3. Then for $q', q'' \in Q'$ and $s \in \Sigma^*$ such that $\delta'(q', s) = q''$, the following properties hold:
i) $q', q'' \in Q \Rightarrow \delta(q', s) = q''$ **ii)** $q', q'' \notin Q \Rightarrow (\forall q_1 \in E_{q'})\, (\exists q_2 \in E_{q''})\, \delta(q_1, s) = q_2$
iii) $[q' \notin Q \wedge q'' \in Q] \Rightarrow (\forall q \in E_{q'})\, \delta(q, s) = q''$
iv) $[q' \in Q \wedge q'' \notin Q] \Rightarrow (\exists q \in E_{q''})\, \delta(q', s) = q$

Proof. Assume initial conditions.
Let $q', q'' \in Q'$, and $s \in \Sigma^*$. Assume: $\delta'(q', s) = q''$
By Proposition 6.3(ii), we can conclude:

$$(\exists q_a, q_b \in Q)\, \delta(q_a, s) = q_b \wedge (q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''}) \tag{1}$$

We will now show that Points (i-iv) are satisfied.

i) Show: $q', q'' \in Q \Rightarrow \delta(q', s) = q''$
Assume: $q', q'' \in Q$
It thus follows by Algorithm 3 that both q' and q'' are λ-equivalent only to themselves, and we can thus conclude by (1) that $q_a = q'$ and $q_b = q''$.
$\Rightarrow \delta(q', s) = q''$ *by* (1)
Part (i) complete.

ii) Show: $q', q'' \notin Q \Rightarrow (\forall q_1 \in E_{q'})\, (\exists q_2 \in E_{q''})\, \delta(q_1, s) = q_2$
Assume: $q', q'' \notin Q$ (2)
By (1), we have: $(q_a = q' \vee q_a \in E_{q'}) \wedge (q_b = q'' \vee q_b \in E_{q''})$

$\Rightarrow q_a \in E_{q'}$ and $q_b \in E_{q''}$ *by* (2) *and Algorithm* 3

Let $q_1 \in E_{q'}$.

As $\delta(q_a, s) = q_b$ by (1), and $q_a \in E_{q'}$, it follows that $\delta(q_1, s)!$.

By Proposition 6.3(i), we have: $\delta(q_a, s) \equiv \delta(q_1, s) \pmod{\lambda}$

$\Rightarrow \delta(q_1, s) \in E_{q''}$ *as* $q_b \in E_{q''}$

We can thus take $q_2 = \delta(q_1, s)$, and we have $q_1 \in E_{q'}, q_2 \in E_{q''}$, and $\delta(q_1, s) = q_2$, as required.

Part (ii) complete.

iii) Show: $[q' \notin Q \wedge q'' \in Q] \Rightarrow (\forall q \in E_{q'})\, \delta(q, s) = q''$

Assume: $q' \notin Q \wedge q'' \in Q$ (3)

By (1) and Algorithm 3, we can conclude: $q_a \in E_{q'}$ and $q_b = q''$

$\Rightarrow \delta(q_a, s) = q''$ *by* (1) (4)

Let $q \in E_{q'}$.

As $\delta(q_a, s) = q''$ and $q_a \in E_{q'}$, it follows that $\delta(q, s)!$.

By Proposition 6.3(i), we have: $\delta(q_a, s) \equiv \delta(q, s) \pmod{\lambda}$

As $q'' \in Q$ by (3), it follows by Algorithm 3 that q'' is only λ-equivalent to itself.

$\Rightarrow \delta(q, s) = q''$ *by* (4)

We thus have $q \in E_{q'}$ and $\delta(q, s) = q''$, as required.

Part (iii) complete.

iv) Show: $[q' \in Q \wedge q'' \notin Q] \Rightarrow (\exists q \in E_{q''})\, \delta(q', s) = q$

Assume: $q' \in Q \wedge q'' \notin Q$

By (1) and Algorithm 3, we can conclude: $q_a = q', q_b \in E_{q''}$ and $\delta(q', s) = q_b$

We can thus take $q = q_b$, and we have $q \in E_{q''}$ with $\delta(q', s) = q$, as required.

Part (iv) complete.

By Parts (i)-(iv), we conclude that Points (i-iv) of the proposition are satisfied. □

Proposition 6.5. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a non-minimal TDES and $\mathbf{G}' = min(\mathbf{G}) = (Q', \Sigma, \delta', q_o', Q_m')$ be the TDES constructed using Algorithms 2 and 3. Then for $q' \in Q_r'$, the following properties hold: **(i)** $q' \in Q \Rightarrow q' \in Q_r$, and **(ii)** $q' \notin Q \Rightarrow (\exists q \in E_{q'})\, q \in Q_r$.

Proof. Let $q' \in Q_r'$ and assume initial conditions.

As $q' \in Q_r'$, we have: $(\exists s \in \Sigma^*)\, \delta'(q_o', s) = q'$

i) Show: $q' \in Q \Rightarrow q' \in Q_r$

Assume: $q' \in Q$ (1)

We have two cases: **(1)** $q_o' \in Q$, or **(2)** $q_o' \notin Q$.

Case 1) $q_o' \in Q$

$\Rightarrow q_o', q' \in Q$ *by* (1)

By Proposition 6.4(i), we have: $\delta(q_o', s) = q'$ (2)

As $q_o' \in Q$, we have: $q_o' = q_o$ *by Steps* 1 *and* 2.3 *of Algorithm* 3

$\Rightarrow \delta(q_o, s) = q'$ *by* (2)

$\Rightarrow q' \in Q_r$

Case 2) $q'_o \notin Q$

$\Rightarrow q'_o \notin Q$ and $q' \in Q$ *by* (1)

By Proposition 6.4(iii), we have: $(\forall q \in E_{q'_o})\, \delta(q, s) = q'$

As $q_o \in E_{q'_o}$ by Algorithm 3, we thus have: $\delta(q_o, s) = q'$

$\Rightarrow q' \in Q_r$

By Cases (1) and (2), we have $q' \in Q_r$, as required.

Part (i) complete.

ii) Show: $q' \notin Q \Rightarrow (\exists q \in E_{q'})\, q \in Q_r$

Assume: $q' \notin Q$ (3)

We have two cases: **(1)** $q'_o \in Q$, or **(2)** $q'_o \notin Q$.

Case 1) $q'_o \in Q$

$\Rightarrow q'_o \in Q$ and $q' \notin Q$ *by* (3)

By Proposition 6.4(iv), we have: $(\exists q \in E_{q'})\, \delta(q'_o, s) = q$ (4)

As $q'_o \in Q$, we have: $q'_o = q_o$ *by Steps* 1 *and* 2.3 *of Algorithm* 3

$\Rightarrow \delta(q_o, s) = q$ *by* (4)

$\Rightarrow q \in Q_r$

Case 2) $q'_o \notin Q$

$\Rightarrow q'_o, q' \notin Q$ *by* (3)

By Proposition 6.4(ii), we have: $(\forall q_1 \in E_{q'_o})\, (\exists q \in E_{q'})\, \delta(q_1, s) = q$

As $q_o \in E_{q'_o}$ by Algorithm 3, we thus have: $\delta(q_o, s) = q$

$\Rightarrow q \in Q_r$

By Cases (1) and (2), we have constructed $q \in E_{q'}$ with $q \in Q_r$.

Part (ii) complete.

By Parts (i) and (ii), we conclude that Points (i-ii) of the proposition are satisfied. □

Now we will present our main ALF result. The theorem given below proves that if a non-minimal TDES having a finite state space is ALF, then the minimal version of this TDES will also be ALF. This will allow us to use our ALF result of Section 5.4.5 whether $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ or $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G})$.

Theorem 6.1. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a TDES with finite state space. Let TDES $\mathbf{G}' = min(\mathbf{G}) = (Q', \Sigma, \delta', q'_o, Q'_m)$ be the minimal version of $\mathbf{G}$ that is constructed using Algorithms 2 and 3. If $\mathbf{G}$ is ALF, then $\mathbf{G}'$ is ALF.

Proof. Assume initial conditions.

Assume $\mathbf{G}$ is ALF and has a finite state space. (1)

If $\mathbf{G}$ is minimal, then $\mathbf{G} = \mathbf{G}'$, and it follows immediately that $\mathbf{G}'$ is ALF.

We now consider the case that $\mathbf{G}$ is non-minimal.

To show that $\mathbf{G}'$ is ALF, it is sufficient to show: $(\forall q' \in Q'_r)(\forall s \in \Sigma^+_{act})\, \delta'(q', s) \neq q'$

We will use proof by contradiction to show our desired result.

Assume $\mathbf{G}'$ is not ALF.

$\Rightarrow (\exists q' \in Q'_r)\, (\exists s \in \Sigma^+_{act})\, \delta'(q', s) = q'$ (2)

We will now show this implies **G** is not ALF, contradicting (1).

To do this, we will need to construct $q \in Q_r$ and $s' \in \Sigma^+_{act}$ such that $\delta(q, s') = q$.

We have two cases: **(i)** $q' \in Q$, or **(ii)** $q' \notin Q$.

Case i) $q' \in Q$

As $\delta'(q', s) = q'$ by (2), we apply Proposition 6.4(i) and conclude: $\delta(q', s) = q'$

As $q' \in Q'_r$ by (2), and $q' \in Q$, we apply Proposition 6.5(i) and conclude: $q' \in Q_r$

We thus take $q = q'$, $s' = s$, and we have $q \in Q_r$, $s' \in \Sigma^+_{act}$ by (2), and $\delta(q, s') = q$, thus contradicting **G** being ALF.

Case ii) $q' \notin Q$

As $\delta'(q', s) = q'$ by (2), we apply Proposition 6.4(ii) and conclude:

$$(\forall q_1 \in E_{q'})\,(\exists q_2 \in E_{q'})\,\delta(q_1, s) = q_2 \tag{3}$$

As $q' \in Q'_r$ by (2), we apply Proposition 6.5(ii) and conclude: $(\exists q_1 \in E_{q'})\, q_1 \in Q_r$ (4)

Applying this to (3), we have: $(\exists q_2 \in E_{q'})\,\delta(q_1, s) = q_2$

As $q_2 \in E_{q'}$, we could apply (3) to q_2 and so on to create a chain of transitions.

Let $n = |E_{q'}|$. As $E_{q'} \subseteq Q$ by Algorithm 2 and the fact that Q is finite by (1), we have $n < \infty$. (5)

Starting with q_1, we could apply (3) repeatedly n times and construct a chain of transitions in δ as: $q_1 \xrightarrow{s} q_2 \xrightarrow{s} \cdots \xrightarrow{s} q_n \xrightarrow{s} q_{n+1}$, where for $1 \leq i \leq n+1, q_i \in E_{q'}$. (6)

As $q_1 \in Q_r$ by (4), it follows that each $q_i \in Q_r$. (7)

We note that as $n < \infty$ by (5), after the n^{th} transition ($q_1 \to q_n$), it is possible that each q_i was a distinct state in $E_{q'}$, but the transition $\delta(q_n, s) = q_{n+1}$ must then involve a duplicate state. (8)

This means states $q_1, \ldots, q_{n+1}$ must contain two duplicate states.

Let $1 \leq i < n+1$ be the index for the first duplicate state, and let $1 < j \leq n+1$ be the index for the second occurrence of this state (i.e. $q_i = q_j$). (9)

Let $k = j - i$. This is the number of transitions separating the two states (i.e. for q_2 and q_1, 2-1=1).

We then take $q = q_i$ and $s' = s \ldots s$. We thus have $q \in Q_r$ by (7), $s' \in \Sigma^+_{act}$ as $s \in \Sigma^+_{act}$ by (2), and $\delta(q, s') = q$ by (6), (8) and (9), which contradicts **G** being ALF.

By **Cases (i)** and **(ii)**, we have proven that **G** is not ALF, which contradicts (1).

As our assumption that $\mathbf{G}'$ is not ALF caused a contradiction, we thus conclude that $\mathbf{G}'$ is ALF. □

In the SD setting, one of the preconditions of the SD controllability and nonblocking verification results is that the closed-loop system formed by synchronizing TDES plant and supervisor models using the synchronous product will not "stop the clock". In [29], this has been proven using the following proposition.

Proposition 6.6. [29] If TDES plant $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ and TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ both have finite state spaces, **G** has proper time behaviour, $\mathbf{S} \,||\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ is ALF, and **S** is timed controllable for **G**, then $(\forall y \in Y_r)\,(\exists s \in \Sigma^*_{act})\,\eta(y, s\tau)!$.

Since we wish to make our $\boldsymbol{\mathcal{S}}$ eligible to be used as the supervisor of the SD setting, we need to show that this result is satisfied by our $\boldsymbol{\mathcal{S}}$ as well. As there are two possible ways to construct $\boldsymbol{\mathcal{S}}$, i.e. $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ or $\boldsymbol{\mathcal{S}} = min(\mathbf{S} \,||_{SD}\, \mathbf{G})$, below we show this result with respect to both cases.

Proposition 6.7. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, and TDES $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ be a supervisor. Let TDES $\boldsymbol{\mathcal{S}}' = \min(\boldsymbol{\mathcal{S}}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 2 and 3. Let the closed-loop system be $\boldsymbol{\mathcal{S}}' \,||\, \mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$. If both $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces, $\mathbf{G}$ has proper time behaviour, $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}}$ is ALF, then:

$$(\forall y \in Y_r)\,(\exists s \in \Sigma^*_{act})\ \eta(y, s\tau)!$$

Proof. Assume initial conditions.

To obtain our desired result, we will show that the assumptions of Proposition 6.6 are satisfied. It is notable that if $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ is already minimal, then $\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{S}}'$. Thus, we need to prove conditions for both $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$.

First, we have that $\mathbf{G}$ has finite state space and proper time behaviour. (1)

Next, we have that both $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces.

$\Rightarrow \boldsymbol{\mathcal{S}}$ has a finite state space (2)

$\Rightarrow \boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}})$ has a finite state space *by Algorithms 2 and 3* (3)

By our initial assumptions, $\mathbf{S}$ is timed controllable with $||_{SD}$ for $\mathbf{G}$.

$\Rightarrow \boldsymbol{\mathcal{S}}$ is timed controllable for $\mathbf{G}$ *by Proposition 5.6* (4)

As state space minimization does not change the automaton's closed behaviour, we have $L(\boldsymbol{\mathcal{S}}') = L(\boldsymbol{\mathcal{S}})$. As timed controllability is a language based property, this implies that $\boldsymbol{\mathcal{S}}'$ is timed controllable for $\mathbf{G}$. (5)

We have that $\boldsymbol{\mathcal{S}} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ is ALF.

$\Rightarrow \boldsymbol{\mathcal{S}} \,||\, \mathbf{G}$ is ALF *by Proposition 5.8* (6)

We now have two cases: **(i)** $\boldsymbol{\mathcal{S}}$ is minimal, or **(ii)** $\boldsymbol{\mathcal{S}}$ is non-minimal.

Case i) $\boldsymbol{\mathcal{S}}$ is minimal

This means $\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}})$ as Algorithms 2 and 3 will make no changes. We can thus use properties for $\boldsymbol{\mathcal{S}}$.

By (1), (2), (4) and (6), all assumptions of Proposition 6.6 are satisfied.

Case ii) $\boldsymbol{\mathcal{S}}$ is non-minimal

This means $\boldsymbol{\mathcal{S}} \neq \boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}})$, so we must use the results for $\boldsymbol{\mathcal{S}}'$.

As $\boldsymbol{\mathcal{S}}$ is ALF, we can conclude by Theorem 6.1 that $\boldsymbol{\mathcal{S}}'$ is ALF.

$\Rightarrow \boldsymbol{\mathcal{S}}' \,||\, \mathbf{G}$ is ALF *by Proposition 5.8* (7)

By (1), (3), (5) and (7), all assumptions of Proposition 6.6 are satisfied.

By **Cases (i)** and **(ii)**, we can apply Proposition 6.6 and conclude:

$$(\forall y \in Y_r)\,(\exists s \in \Sigma^*_{act})\ \eta(y, s\tau)! \qquad \square$$

7 Equivalence of SD Controllers

In this section, we present our final set of results with respect to establishing equivalence between the SD and our $||_{SD}$ setting. Specifically, this section proves the output equivalence

between the two SD controllers that are generated by translating CS deterministic TDES supervisors $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}$ of the $||_{SD}$ and SD settings respectively. In other words, we will show that the two SD controllers generate the same sequence of outputs in response to a given valid (possible according to system model) sequence of inputs.

We begin this section by stating some preliminary definitions that we have defined for our $||_{SD}$ setting. Then, we present our supporting propositions that will help us in proving our final result that the two SD controllers translated from $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}$ produce the same output information for the same valid input sequence with respect to the closed-loop behaviour.

Please note that the functions and notation used in this section have already been introduced in Section 3. We will provide a brief introduction, but recommend the reader to refresh the details (specifically Sections 3.6 and 3.7) before reading this section.

7.1 Preliminary Definitions

In this section, we present some definitions that we have adapted from [42] to define the concepts related to SD controllers in our $||_{SD}$ setting.

One of the goals of devising the $||_{SD}$ setting is to liberate the software and hardware practitioners from designing and implementing a potentially intricate supervisor in the SD setting. Rather, we want them to design and implement a much simpler TDES supervisor $\mathbf{S}$ in the $||_{SD}$ setting. In order to do that, it is important to show that the SD controller generated by translating CS deterministic supervisor $\mathbf{S}$ in our $||_{SD}$ setting is output equivalent to the SD controller that is obtained by translating CS deterministic supervisor $\boldsymbol{\mathcal{S}}$ (possibly $min(\boldsymbol{\mathcal{S}})$) of the SD setting. In other words, we wish to prove that whether practitioners physically implement $\mathbf{S}$ or $\boldsymbol{\mathcal{S}}$, they are going to achieve the same physical control action with respect to a given TDES plant $\mathbf{G}$. This result will also be essential for the proofs of Section 8 so we can use the SD controller for $\mathbf{S}$ and apply it for proofs using $\boldsymbol{\mathcal{S}}$.

It is important to clarify that we do not require the two SD controllers to be identical. We only wish to demonstrate that they produce the same enablement and forcing information for a given plant $\mathbf{G}$ for valid input sequences.

First, we provide a definition for valid input sequences with respect to the closed-loop behaviour. It is worth-mentioning that the definition given below is generic with respect to forming the closed-loop system, $\mathbf{G}_{cl}$. By this we mean that our definition is independent of the operator that is used to form $\mathbf{G}_{cl}$. $\mathbf{G}_{cl}$ could be constructed by synchronizing TDES plant and supervisor models using the $||_{SD}$ operator, the synchronous product, the meet or the product operator. For this definition, we are only interested in the language obtained as a result of combining the plant and supervisor models, i.e. $L(\mathbf{G}_{cl})$. Our goal is to ensure that whichever operator we use to obtain $L(\mathbf{G}_{cl})$, our definition will remain applicable and valid.

Definition 7.1. For TDES plant $\mathbf{G} = (Q, \Sigma_{\mathbf{G}}, \delta, q_o, Q_m)$ and CS deterministic TDES supervisor $\mathbf{S} = (X, \Sigma_{\mathbf{S}}, \xi, x_o, X_m)$, let $\mathbf{G}_{cl} = (Y, \Sigma, \eta, y_o, Y_m)$ be the closed-loop system constructed by synchronizing $\mathbf{S}$ and $\mathbf{G}$. For system event set Σ, with canonical event mapping function γ_g, global input vector $\mathbf{i}_g$, and activity event set Σ_{act}, a canonical input sequence $\{\mathbf{i}_g(k)\}$ is said to be *input valid for* $L(\mathbf{G}_{cl})$, if:

$$(\forall k \in \{1, 2, \ldots\})\,(\exists\, s_1, s_2, \ldots, s_k \in L_{conc})\,[s_1 s_2 \ldots s_k \in L(\mathbf{G}_{cl})] \wedge$$
$$[(\forall n \in \{1, 2, \ldots, k\})\,(\forall \sigma \in \Sigma_{act})\, i_{g,\gamma_g(\sigma)}\,(n) = 1 \text{ iff } \sigma \in Occu(s_n)]$$

In the above definition, γ_g is a bijective map that associates each $\sigma \in \Sigma_{act}$ with a unique element of input vector $\mathbf{i}_g = [i_{g,0}, i_{g,1}, \ldots, i_{g,v-1}]$ $(v = |\Sigma_{act}|)$, $\{\mathbf{i}_g(k)\} = \{i_g(1), i_g(2), \ldots\}$ is a

sequence of input vectors taken at different sampling instances, $i_{g,\gamma_g(\sigma)}(n)$ is element $i_{g,\gamma_g(\sigma)}$ (the element γ_g associate with σ) of the n^{th} vector in sequence $\{\mathbf{i}_g(k)\}$, and $\sigma \in Occu(s_n)$ means the string s_n contains event σ. For more information, see Sections 3.4, 3.6 and 3.7.

Essentially, in this definition, we require the input sequence $\{\mathbf{i}_g(k)\}$ to correspond to a sequence of concurrent strings that our closed-loop system $\mathbf{G}_{cl}$ will accept. This is necessary as the TDES supervisor to SD controller translation method (Section 3.7) only dictates outputs for these inputs and leaves the outputs for other inputs unspecified. This means controllers could differ for input sequences that are not possible in the system.

Before we proceed to our next definition, please note that in our $||_{SD}$ setting, we construct our closed-loop system as $\mathbf{S}\,||_{SD}\,\mathbf{G}$, whereas the SD setting constructs the closed-loop system as $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$. Because of our language equivalence results (Section 5.3), we know that both closed-loop systems have the same closed and marked languages. This implies that input sequences that represent valid input strings in the behaviour of the two closed-loop systems will be the same.

In order to prove our controller equivalence results, we wish to prove that two SD controllers, $\mathbf{C}_1$ and $\mathbf{C}_2$, are output equivalent with respect to the closed-loop behaviour for plant $\mathbf{G}$ and supervisors $\mathbf{S}_1$ and $\mathbf{S}_2$, where $\mathbf{C}_1$ is constructed from $\mathbf{S}_1$ and $\mathbf{C}_2$ is constructed from $\mathbf{S}_2$. For this definition, we are assuming that the two closed-loop systems, represented by TDES $\mathbf{G}_{cl,1}$ and $\mathbf{G}_{cl,2}$, have the same closed languages, i.e. $L(\mathbf{G}_{cl,1}) = L(\mathbf{G}_{cl,2})$. Before we present our formal definition, three points are notable and worth elaborating.

1. This definition is independent of the synchronization operators that are used to form the closed-loop systems. As long as the closed-loop behaviour of the two systems is the same, the definition remains applicable and valid, and the choice of synchronization operator(s) is trivial.
2. This definition is stated with respect to the closed-loop behaviour of the two systems, and not in terms of the actual system automata. This is because the TDES representation of the two closed-loop systems might not be exactly the same due to different state labels or if one TDES is non-minimal, despite them having the same closed behaviour.
3. As the closed behaviour of the two closed-loop systems is the same, i.e. $L(\mathbf{G}_{cl,1}) = L(\mathbf{G}_{cl,2})$, using either $L(\mathbf{G}_{cl,1})$ or $L(\mathbf{G}_{cl,2})$ will not make any difference because we are eventually referring to the same language. However, to be clear and avoid any ambiguity, instead of using either one of these two labels, we will refer to this language using a more generic label $L(\mathbf{G}_{cl})$, without any loss of generality.

Definition 7.2. For TDES plant $\mathbf{G} = (Y, \Sigma_{\mathbf{G}}, \delta, y_o, Y_m)$, let $\mathbf{S}_j = (X_j, \Sigma_j, \xi_j, x_{o,j}, X_{m,j})$, $(j = 1, 2)$ be two CS deterministic TDES supervisors. Let $\mathbf{G}_{cl,j}$ be the closed-loop system formed by synchronizing $\mathbf{G}$ and $\mathbf{S}_j$. Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be control equivalent for $\mathbf{G}$, i.e. $L(\mathbf{G}_{cl}) = L(\mathbf{G}_{cl,1}) = L(\mathbf{G}_{cl,2})$. For system event set Σ, with canonical event mapping function γ_g, and activity event set Σ_{act}, let $\mathbf{C}_j = (I_j, Z_j, Q_j, \Omega_j, \Phi_j, \mathbf{q}_{res,j})$ be the SD controller constructed from $\mathbf{S}_j$. Let r_j be the number of output variables for a vector in Z_j, and η_j be the output event mapping function for $\mathbf{C}_j$. $\mathbf{C}_1$ and $\mathbf{C}_2$ are said to be *output equivalent with respect to the closed-loop behaviour* $L(\mathbf{G}_{cl})$ if, for any canonical input sequence $\{\mathbf{i}_g(k)\}$ that is input valid for $L(\mathbf{G}_{cl})$ and induced output $\mathbf{z}_j(k') = [z_{j,0}(k'), z_{j,1}(k'), \ldots, z_{j,r_{j-1}}(k')] \in Z_j$ at time $k' = \{0, 1, 2, \ldots\}$, the following conditions are satisfied:

1. $r_1 = r_2$
2. $(\forall\, 0 \leq i < r_1)\, \eta_1^{-1}(i) = \eta_2^{-1}(i)$

3. $(\forall k' \in \{0, 1, 2, \ldots\})\, \mathbf{z}_1(k') = \mathbf{z}_2(k')$

In the above definition, $\mathbf{z}_j(k')$ is the current output vector for controller $\mathbf{C}_j$, at time k'. Also, η_j is a bijective map that associates each $\sigma \in \Sigma_{hib} \cap \Sigma_j$ with a unique element in $\mathbf{z}_j(k')$ in a way that respects the event ordering of γ_g. See Section 3.7.1 for details.

Point 1 requires that output vectors of the two controllers must be the same size, i.e. they must have same number of output variables. Point 2 enforces the condition that the two output vectors must have the same prohibitable events stored in exactly the same order/sequence. Finally, Point 3 imposes the constraint that for any value of k', the two output vectors must have the same enablement information, i.e. one controller should enable a prohibitable event if and only if the other does. This means that the two controllers must agree with respect to the enablement of prohibitable events at the reset state, and must continue to agree in the future as well.

The above definition gives us a way to compare the output information of the SD controller translated from supervisor $\mathbf{S}$ in the $||_{SD}$ setting to the SD controller translated from $\boldsymbol{\mathcal{S}}$ (or $min(\boldsymbol{\mathcal{S}})$, if $\boldsymbol{\mathcal{S}}$ is not minimal) in the SD setting. If the two controllers are output equivalent with respect to the shared closed-loop behaviour, then they will assert the same forcing and enablement information on plant $\mathbf{G}$. This will allow us to implement the controller for $\mathbf{S}$, but apply the controllability and nonblocking results of the SD setting to $\boldsymbol{\mathcal{S}}$ and this controller.

7.2 Supporting Propositions

In this section, we introduce two supporting propositions that will be used in the next section to prove our main result that the corresponding SD controllers generated in the SD and $||_{SD}$ settings are output equivalent. Please recall that as per our assumptions (Section 5.2), all TDES are deterministic automata.

To convert a TDES supervisor to an SD controller, it must be CS deterministic. As discussed in Section 6.3.1, we might need to minimize the TDES supervisor $\boldsymbol{\mathcal{S}}$ in order to make it CS deterministic. As λ-equivalent states (Definition 2.14) are combined during the state minimization process, this will make it complicated to compare $\boldsymbol{\mathcal{S}}$ to our supervisor $\mathbf{S}$ of the $||_{SD}$ setting. As a result, in the proofs presented in this section, we will refer to the distinct sets of λ-equivalent states, labelled as $E_k(|E_k| \geq 2)$, that are created by Algorithm 2 during the minimization process. We will refer to E_k as $E_{q'}$, where q' is the aggregate state label associated with E_k by Algorithm 3. Please refer to Section 6.3.2 for details.

In Proposition 7.1 given below, X_{samp} (Definition 3.2) is the set of sampled states for TDES supervisor $\mathbf{S}$. These are the states of $\mathbf{S}$ that are reached from the initial state by a sampled string (Definition 3.1). The prohibited action function ζ (Definition 3.10) is associated with a specific supervisor, and maps sampled states of the supervisor to the set of prohibitable events enabled at these states.

Proposition 7.1. Let $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a non-minimal TDES supervisor and $\mathbf{S}' = min(\mathbf{S}) = (X', \Sigma, \xi', x'_o, X'_m)$ be the minimal TDES constructed using Algorithms 2 and 3. Let ζ be the prohibited action function for $\mathbf{S}$ and ζ' be the prohibited action function for $\mathbf{S}'$. Then, for $x_1, x_2 \in X$ and $x' \in X'$, the following properties hold:

i) $x_1 \equiv x_2 \,(\text{mod } \lambda) \Rightarrow (\forall \sigma \in \Sigma)\, \xi(x_1, \sigma)! \Rightarrow \xi(x_2, \sigma)!$

ii) $[x_1 \equiv x_2 \,(\text{mod } \lambda) \wedge x_1, x_2 \in X_{samp}] \Rightarrow \zeta(x_1) = \zeta(x_2)$

iii) $[x' \notin X \wedge x' \in X'_{samp}] \Rightarrow (\forall x \in E_{x'} \cap X_{samp})\, \zeta(x) = \zeta'(x')$

iv) $[x' \in X \wedge x' \in X'_{samp}] \Rightarrow x' \in X_{samp} \wedge \zeta(x') = \zeta'(x')$

Proof. Let $x_1, x_2 \in X$ and $x' \in X'$. Assume initial conditions. (1)

i) Show: $x_1 \equiv x_2 \,(\text{mod } \lambda) \Rightarrow (\forall \sigma \in \Sigma)\, \xi(x_1, \sigma)! \Rightarrow \xi(x_2, \sigma)!$
Assume: $x_1 \equiv x_2 \,(\text{mod } \lambda)$
Let $\sigma \in \Sigma$. Let $s = \sigma$.
As $s \in \Sigma^*$, $\xi(x_1, s)! \Leftrightarrow \xi(x_2, s)!$ follows automatically from Definition 2.14 of λ-equivalence.
Part (i) complete.

ii) Show: $[x_1 \equiv x_2 \,(\text{mod } \lambda) \wedge x_1, x_2 \in X_{samp}] \Rightarrow \zeta(x_1) = \zeta(x_2)$
Assume: $x_1 \equiv x_2 \,(\text{mod } \lambda)$ and $x_1, x_2 \in X_{samp}$ (2)
To show $\zeta(x_1) = \zeta(x_2)$, by Definition 3.10 of ζ it is sufficient to show:

$$\{\sigma \in \Sigma_{hib} \,|\, \xi(x_1, \sigma)!\} = \{\sigma \in \Sigma_{hib} \,|\, \xi(x_2, \sigma)!\}$$

As $\Sigma_{hib} \subseteq \Sigma$ and $x_1 \equiv x_2 \,(\text{mod } \lambda)$ by (2), this follows automatically from Part (i).
Part (ii) complete.

iii) Show: $[x' \notin X \wedge x' \in X'_{samp}] \Rightarrow (\forall x \in E_{x'} \cap X_{samp})\, \zeta(x) = \zeta'(x')$
Assume: $x' \notin X$ and $x' \in X'_{samp}$ (3)
As $x' \notin X$, this means that x' was added to $\mathbf{S}'$ by Algorithm 3.
Let $x \in E_{x'} \cap X_{samp}$. (4)
We first note that by Definition 3.10, we have:

$$\zeta(x) = \{\sigma \in \Sigma_{hib} \,|\, \xi(x, \sigma)!\} \text{ and } \zeta'(x') = \{\sigma \in \Sigma_{hib} \,|\, \xi'(x', \sigma)!\}$$

To show that $\zeta(x) = \zeta'(x')$, it is sufficient to show: $(\forall \sigma \in \Sigma_{hib})\, \xi(x, \sigma)! \Leftrightarrow \xi'(x', \sigma)!$
Let $\sigma \in \Sigma_{hib}$.
Part 1) Show: $\xi(x, \sigma)! \Rightarrow \xi'(x', \sigma)!$
Assume: $\xi(x, \sigma)!$
Let $x_b = \xi(x, \sigma)$, and thus $x_b \in X$.
By Proposition 6.3(iii), we can conclude:

$$(\exists x'_1, x'_2 \in X')\, \xi'(x'_1, \sigma) = x'_2 \wedge (x'_1 = x \vee x \in E_{x'_1}) \wedge (x'_2 = x_b \vee x_b \in E_{x'_2}) \quad (5)$$

As $x \in E_{x'}$ by (4), it follows that $x \notin X'$.
As $(x'_1 = x \vee x \in E_{x'_1})$ by (5), it follows that $x \neq x'_1$, thus following that $x \in E_{x'_1}$.
$\Rightarrow x' = x'_1$ *as Algorithm 2 will put a state in X into at most one distinct set of λ-equivalent states*
$\Rightarrow \xi'(x', \sigma) = x'_2$ *by* (5)
$\Rightarrow \xi'(x', \sigma)!$
Part 2) Show: $\xi'(x', \sigma)! \Rightarrow \xi(x, \sigma)!$
Assume: $\xi'(x', \sigma)!$
Let $x'' = \xi'(x', \sigma)$. (6)
We have two cases: **(a)** $x'' \in X$, or **(b)** $x'' \notin X$.
Case 2.a) $x'' \in X$
$\Rightarrow x' \notin X$ and $x'' \in X$ *by* (3)
By Proposition 6.4(iii), we can conclude: $(\forall x_a \in E_{x'})\, \xi(x_a, \sigma) = x''$

As $x \in E_{x'}$ by (4), we have: $\xi(x,\sigma) = x''$
$\Rightarrow \xi(x,\sigma)!$

Case 2.b) $x'' \notin X$
$\Rightarrow x', x'' \notin X$ and $\xi'(x',\sigma) = x''$ *by* (3) *and* (6)
By Proposition 6.4(ii), we can conclude: $(\forall x_a \in E_{x'})\,(\exists x_b \in E_{x''})\,\xi(x_a,\sigma) = x_b$
As $x \in E_{x'}$ by (4), we have: $\xi(x,\sigma) = x_b$
$\Rightarrow \xi(x,\sigma)!$

By Cases (2.a) and (2.b), we have $\xi(x,\sigma)!$. We thus conclude $\xi'(x',\sigma)! \Rightarrow \xi(x,\sigma)!$.
By Parts (1) and (2), we conclude that for $\sigma \in \Sigma_{hib}$, $\xi(x,\sigma)! \Leftrightarrow \xi'(x',\sigma)!$.
We thus conclude $\zeta(x) = \zeta(x')$.
Part (iii) complete.

iv) Show: $[x' \in X \wedge x' \in X'_{samp}] \Rightarrow x' \in X_{samp} \wedge \zeta(x') = \zeta'(x')$
Assume: $x' \in X$ and $x' \in X'_{samp}$ (7)
We will now show this implies: $x' \in X_{samp}$ and $\zeta(x') = \zeta'(x')$

Part 1) Show: $x' \in X_{samp}$
By Definition 3.2 of sampled states, it is sufficient to show:

$$x' \in \{x_a \in X \mid (\exists s \in L(\mathbf{S}) \cap L_{samp})\, x_a = \xi(x_o, s)\}$$

As $x' \in X$ by (7), all that remains is to show: $(\exists s \in L(\mathbf{S}) \cap L_{samp})\, x' = \xi(x_o, s)$
As $x' \in X'_{samp}$ by (7), it follows that: $(\exists s \in L(\mathbf{S}') \cap L_{samp})\, x' = \xi'(x'_o, s)$ (8)
We have two cases: **(a)** $x'_o \in X$, or **(b)** $x'_o \notin X$.

Case 1.a) $x'_o \in X$
As $x' \in X$ by (7), and $x'_o \in X$, we apply Proposition 6.4(i) and conclude: $\xi(x'_o, s) = x'$
As $x'_o \in X$, it follows by Algorithms 2 and 3 that $x'_o = x_o$.
$\Rightarrow \xi(x_o, s) = x'$

Case 1.b) $x'_o \notin X$
As $x' \in X$ by (7), $x'_o \notin X$, and $\xi'(x'_o, s) = x'$ by (8), we can apply Proposition 6.4(iii) and conclude: $(\forall x_a \in E_{x'_o})\,\xi(x_a, s) = x'$
As $x'_o \notin X$, by Algorithms 2 and 3 we can conclude $x_o \in E_{x'_o}$.
$\Rightarrow \xi(x_o, s) = x'$
By Cases (1.a) and (1.b), we have: $\xi(x_o, s) = x'$
$\Rightarrow s \in L(\mathbf{S}) \cap L_{samp}$ *by* (8)
$\Rightarrow x' \in X_{samp}$
Part (1) complete.

Part 2) Show: $\zeta(x') = \zeta'(x')$
To show $\zeta(x') = \zeta'(x')$, by Definition 3.10 it is sufficient to show:

$$(\forall \sigma \in \Sigma_{hib})\,\xi(x',\sigma)! \Leftrightarrow \xi'(x',\sigma)!$$

Let $\sigma \in \Sigma_{hib}$.
Part 2.a) Show: $\xi(x',\sigma)! \Rightarrow \xi'(x',\sigma)!$
Assume: $\xi(x',\sigma)!$

Let $x_b = \xi(x', \sigma)$.

By Proposition 6.3(iii), we can conclude:

$$(\exists x'_a, x'_b \in X')\, \xi'(x'_a, \sigma) = x'_b \wedge (x'_a = x' \vee x' \in E_{x'_a}) \wedge (x'_b = x_b \vee x_b \in E_{x'_b})$$

As $x' \in X \cap X'$ by (1) and (7), by Algorithms 2 and 3 we have $x'_a = x'$, as x' is λ-equivalent only to itself.

$\Rightarrow \xi'(x', \sigma) = x'_b$

$\Rightarrow \xi'(x', \sigma)!$

Part (2.a) complete.

Part 2.b) Show: $\xi'(x', \sigma)! \Rightarrow \xi(x', \sigma)!$

Assume: $\xi'(x', \sigma)!$

Let $x'_b = \xi'(x', \sigma)$.

$\Rightarrow x'_b \in X', x' \in X \cap X'$, and $\xi'(x', \sigma) = x'_b$ *by* (1) *and* (7) (9)

By Proposition 6.3(ii), we can conclude:

$$(\exists x_a, x_b \in X)\, \xi(x_a, \sigma) = x_b \wedge (x_a = x' \vee x_a \in E_{x'}) \wedge (x_b = x'_b \vee x_b \in E_{x'_b})$$

As $x' \in X$ by (9), by Algorithms 2 and 3 this implies that x' is λ-equivalent only to itself.

$\Rightarrow x_a = x'$

$\Rightarrow \xi(x', \sigma) = x_b$

$\Rightarrow \xi(x', \sigma)!$

Part (2.b) complete.

By Parts (2.a) and (2.b), we conclude $(\forall \sigma \in \Sigma_{hib})\, \xi(x', \sigma)! \Leftrightarrow \xi'(x', \sigma)!$.

$\Rightarrow \zeta(x') = \zeta'(x')$

Part (2) complete.

By Parts (1) and (2), we thus conclude $x' \in X_{samp}$ and $\zeta(x') = \zeta'(x')$.

Part (iv) complete.

By Parts (i)-(iv), we conclude that Points (i-iv) of the proposition are satisfied. □

We now introduce another supporting proposition that will be key in proving our main result of output equivalent controllers. The following proposition shows that a sampled string accepted by the closed-loop system of our $||_{SD}$ setting, $\mathbf{S}\,||_{SD}\,\mathbf{G}$, will take each supervisor $\mathbf{S}$, $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$ to a state with the same prohibitable events enabled. This means whether we use $\boldsymbol{\mathcal{S}}$ or $\boldsymbol{\mathcal{S}}'$, we get the same result that matches our supervisor $\mathbf{S}$.

Proposition 7.2. Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant and TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor. Let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G} = (Y, \Sigma, \eta, y_o, Y_m)$ be a supervisor and $\boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}}) = (Y', \Sigma, \eta', y'_o, Y'_m)$ be the minimal TDES constructed using Algorithms 2 and 3. Let $\zeta_{\mathbf{S}}$, $\zeta_{\boldsymbol{\mathcal{S}}}$ and $\zeta_{\boldsymbol{\mathcal{S}}'}$ be the prohibited action functions for supervisors $\mathbf{S}$, $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$ respectively. Then:

$$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp})\, \zeta_{\mathbf{S}}(\xi(x_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}}(\eta(y_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}'}(\eta'(y'_o, s))$$

Proof. Assume initial conditions. Let $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and $\boldsymbol{\mathcal{S}}' = min(\boldsymbol{\mathcal{S}})$. (1)

Let $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$. (2)

Let X_{samp}, Y_{samp} and Y'_{samp} be the sets of sampled states for $\mathbf{S}$, $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$ respectively.

First, we note that $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ means that $s \in L(\boldsymbol{\mathcal{S}})$ by definition. As state minimization

does not affect the closed behaviour of an automaton, this implies $s \in L(\boldsymbol{\mathcal{S}}')$. (3)

$\Rightarrow \eta(y_o, s)!$ and $\eta'(y'_o, s)!$

Let $y = \eta(y_o, s)$ and $y' = \eta'(y'_o, s)$. (4)

By Definition 4.1 of $||_{SD}$ operator, we have: $(\exists x \in X)\,(\exists q \in Q)\, y = (x, q)$ (5)

As both $\mathbf{G}$ and $\mathbf{S}$ are defined over Σ, it follows by the definition of $||_{SD}$ that:

$\xi(x_o, s) = x$ and $\delta(q_o, s) = q$ (6)

$\Rightarrow s \in L(\mathbf{S})$ and $s \in L(\mathbf{G})$ (7)

$\Rightarrow s \in L(\mathbf{S}) \cap L_{samp}$ *by* (2)

$\Rightarrow x \in X_{samp}$ *by Definition* 3.2 *of sampled states*

As $s \in L(\boldsymbol{\mathcal{S}})$ by (3), by (2) we have: $s \in L(\boldsymbol{\mathcal{S}}) \cap L_{samp}$

As $y = \eta(y_o, s)$ by (4), by Definition 3.2 we have: $y \in Y_{samp}$ (8)

Similarly, we have: $y' \in Y'_{samp}$ (9)

This means that $\zeta_{\mathbf{S}}(x), \zeta_{\boldsymbol{\mathcal{S}}}(y)$ and $\zeta_{\boldsymbol{\mathcal{S}}'}(y')$ are defined.

We will now show: $\zeta_{\mathbf{S}}(\xi(x_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}}(\eta(y_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}'}(\eta'(y'_o, s))$

By (4) and (6), it is sufficient to show: $\zeta_{\mathbf{S}}(x) = \zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$

We will show this in two steps.

Part 1) Show: $\zeta_{\mathbf{S}}(x) = \zeta_{\boldsymbol{\mathcal{S}}}(y)$

By Definition 3.10 of ζ, it is sufficient to show: $\{\sigma \in \Sigma_{hib} \,|\, \xi(x, \sigma)!\} = \{\sigma \in \Sigma_{hib} \,|\, \eta(y, \sigma)!\}$

This is equivalent to showing: $(\forall \sigma \in \Sigma_{hib})\, \xi(x, \sigma)! \Leftrightarrow \eta(y, \sigma)!$

Let $\sigma \in \Sigma_{hib}$. (10)

Part 1.a) Show: $\xi(x, \sigma)! \Rightarrow \eta(y, \sigma)!$

Assume: $\xi(x, \sigma)!$ (11)

$\Rightarrow s\sigma \in L(\mathbf{S})$ *by* (6) *and* (7)

As $s \in L(\mathbf{S}) \cap L(\mathbf{G})$ by (7), $\sigma \in \Sigma_{hib}$ by (10), and $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$ by (1), we can conclude: $s\sigma \in L(\mathbf{G})$

$\Rightarrow \delta(q, \sigma)!$ *by* (6)

As $\xi(x, \sigma)!$ by (11), $\delta(q, \sigma)!$, $\sigma \in \Sigma_{hib}$ by (10), and $y = (x, q)$ by (5), by the definition of $||_{SD}$, we conclude: $\eta((x, q), \sigma)!$

$\Rightarrow \eta(y, \sigma)!$

Part 1.b) Show: $\eta(y, \sigma)! \Rightarrow \xi(x, \sigma)!$

Assume: $\eta(y, \sigma)!$

$\Rightarrow \eta((x, q), \sigma)!$ *by* (5)

$\Rightarrow \xi(x, \sigma)!$ *by definition of* $||_{SD}$ *and the fact that* $\mathbf{G}$ *and* $\mathbf{S}$ *are defined over* Σ

By Parts (1.a) and (1.b), we can conclude: $(\forall \sigma \in \Sigma_{hib})\, \xi(x, \sigma)! \Leftrightarrow \eta(y, \sigma)!$

$\Rightarrow \zeta_{\mathbf{S}}(x) = \zeta_{\boldsymbol{\mathcal{S}}}(y)$

Part (1) complete.

Part 2) Show: $\zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$

By (4), we have: $y = \eta(y_o, s)$

Using $\eta(y_o, s) = y$, we can apply Proposition 6.3(iii) and conclude:

$$(\exists y'_a, y'_b \in Y')\, \eta'(y'_a, s) = y'_b \wedge (y'_a = y_o \vee y_o \in E_{y'_a}) \wedge (y'_b = y \vee y \in E_{y'_b}) \quad (12)$$

From Algorithms 2 and 3, we know that y_o belongs to at most one set of λ-equivalent states (E_k), and that either $y_o = y'_o$ or $y_o \in E_{y'_o}$.
$\Rightarrow \eta'(y'_o, s) = y'_b$
$\Rightarrow y'_b = y' \quad as\ y' = \eta'(y'_o, s)\ by\ (4)$
$\Rightarrow y' = y$ or $y \in E_{y'} \quad by\ (12)$
We thus have two cases: **(a)** $y' = y$, or **(b)** $y \in E_{y'}$.

Case 2.a) $y' = y$ (13)
$\Rightarrow y' \in Y$
As we have $y' \in Y \cap Y'$ by (4), and $y' \in Y'_{samp}$ by (9), we can apply Proposition 7.1(iv) and we have: $\zeta_{\boldsymbol{\mathcal{S}}}(y') = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$
$\Rightarrow \zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y') \quad by\ (13)$

Case 2.b) $y \in E_{y'}$ (14)
By Algorithms 2 and 3, this implies: $y' \notin Y$
As $y' \in Y'$ by (4), $y' \notin Y$, and $y' \in Y'_{samp}$ by (9), we can apply Proposition 7.1(iii) and conclude: $(\forall y_a \in E_{y'} \cap Y_{samp})\, \zeta_{\boldsymbol{\mathcal{S}}}(y_a) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$
As $y \in E_{y'}$ by (14) and $y \in Y_{samp}$ by (8), we can conclude: $\zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$
By Cases (2.a) and (2.b), we have: $\zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$
Part (2) complete.
Combining Parts (1) and (2), we can conclude: $\zeta_{\mathbf{S}}(x) = \zeta_{\boldsymbol{\mathcal{S}}}(y) = \zeta_{\boldsymbol{\mathcal{S}}'}(y')$
By (4) and (6), we can conclude $\zeta_{\mathbf{S}}(\xi(x_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}}(\eta(y_o, s)) = \zeta_{\boldsymbol{\mathcal{S}}'}(\eta'(y'_o, s))$, as required. □

7.3 Output Equivalent Controllers

In this section, we present our main result for output equivalence between two SD controllers that are translated using the method presented in Section 3.7.

Theorem 7.1 given below proves that an SD controller translated from supervisor $\mathbf{S}$ will be output equivalent to a controller translated from supervisor $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$. In this theorem, we only consider the controller for $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$, and not $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$. This is because if $\boldsymbol{\mathcal{S}}$ is already minimal, then $min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = \boldsymbol{\mathcal{S}}$. Thus, examining $\boldsymbol{\mathcal{S}}'$ without assuming that $\boldsymbol{\mathcal{S}}$ is minimal will cover both cases. Also, in Proposition 7.2, we have already proven that for a valid sampled string, both $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}'$ provide the same enablement information for prohibitable events.

Theorem 7.1. Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$, and let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES supervisor $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ constructed using Algorithms 2 and 3 be CS deterministic. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller translated from $\mathbf{S}$, and $\boldsymbol{\mathcal{C}}' = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller translated from $\boldsymbol{\mathcal{S}}'$. Then, $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ are output equivalent with respect to the closed-loop behaviour $L(\mathbf{G}_{cl})$, with $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$.

Proof. Assume initial conditions.
First, we will describe our setting and notation for the proof.
Let $\Sigma_{act} \subseteq \Sigma$ be the set of activity events and $\Sigma_{hib} \subseteq \Sigma_{act}$ be the set of prohibitable events.
Let $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and $\boldsymbol{\mathcal{S}}' = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$. (1)

As state minimization process does not change the closed behaviour of an automaton, thus we have: $L(\mathbf{G}_{cl}) = L(\boldsymbol{\mathcal{S}}')$

By Corollary 5.1(iii), we have $L(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}' \,||\, \mathbf{G})$. We thus have: $L(\mathbf{G}_{cl}) = L(\mathbf{S} \,||_{SD}\, \mathbf{G}) = L(\boldsymbol{\mathcal{S}}' \,||\, \mathbf{G})$

As $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ are control equivalent (Definition 2.28), this means we can apply Definition 7.2 of output equivalence to $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$.

Let $X_{samp} \subseteq X$ and $X'_{samp} \subseteq X'$ be the sets of sampled states for $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ respectively.

Let $\Lambda : X_{samp} \to Q$ and $\Lambda' : X'_{samp} \to Q'$ be the injective state mapping functions (Definition 3.15) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively.

Let $\Gamma_Z : \mathrm{Pwr}(\Sigma_{hib}) \to Z$ and $\Gamma_{Z'} : \mathrm{Pwr}(\Sigma_{hib}) \to Z'$ be the bijective output set mapping functions (Definition 3.17) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively.

Let $\Phi : Q \to Z$ and $\Phi' : Q' \to Z'$ be the state-to-output maps (Definition 3.19) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively.

Let $\zeta : X_{samp} \to \mathrm{Pwr}(\Sigma_{hib})$ and $\zeta' : X'_{samp} \to \mathrm{Pwr}(\Sigma_{hib})$ be the prohibited action functions (Definition 3.10) for $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ respectively.

Let γ_g be the canonical event mapping function (Definition 3.11) for the system. This is the default way to order event variables in vectors.

Let $v = |\Sigma_{act}|$.

As both $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ are defined over Σ, it follows that each input vector $i \in I$ and $i' \in I'$ is the same size, i.e. each contains v variables.

Let $\gamma : \Sigma_{act} \to \{0, 1, \ldots, v-1\}$ and $\gamma' : \Sigma_{act} \to \{0, 1, \ldots, v-1\}$ be the input event mapping functions (Definition 3.12) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively. By definition of γ and γ', we have:

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{act})\, \gamma_g(\sigma_1) < \gamma_g(\sigma_2) \Rightarrow \gamma(\sigma_1) < \gamma(\sigma_2) \wedge \gamma'(\sigma_1) < \gamma'(\sigma_2)$$

This implies that there is only one way to define γ and γ', and they both must equal γ_g, i.e. $\gamma = \gamma' = \gamma_g$. (2)

Let $\{\mathbf{i}_g(k'')\}$ be a canonical input sequence with respect to γ_g (i.e. its event variables ordering matches γ_g), and let the sequence be input valid for $L(\mathbf{G}_{cl})$. (3)

As $\gamma = \gamma' = \gamma_g$ by (2), it follows that $\{\mathbf{i}_g(k'')\}$ can be used as input vectors for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ directly, without any conversion.

Let $r = |\Sigma_{hib}|$.

As both $\mathbf{S}$ and $\boldsymbol{\mathcal{S}}'$ are defined over Σ, this implies that each output vector $\mathbf{z} \in Z$ and $\mathbf{z}' \in Z'$ is the same size, i.e. each contains r variables. (4)

Let $\eta : \Sigma_{hib} \to \{0, 1, \ldots, r-1\}$ and $\eta' : \Sigma_{hib} \to \{0, 1, \ldots, r-1\}$ be the bijective output event mapping functions (Definition 3.13) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}'$ respectively. By definition of η and η', we have:

$$(\forall \sigma_1, \sigma_2 \in \Sigma_{hib})\, \gamma_g(\sigma_1) < \gamma_g(\sigma_2) \Rightarrow \eta(\sigma_1) < \eta(\sigma_2) \wedge \eta'(\sigma_1) < \eta'(\sigma_2)$$

This implies that there is only one way to define η and η'. Thus we have $\eta = \eta'$. (5)

We note that the definition of Γ_Z and $\Gamma_{Z'}$ is defined in terms of η and η' respectively. Since $\eta = \eta'$ by (5), this implies that $\Gamma_Z = \Gamma_{Z'}$. (6)

This implies that output vectors $\mathbf{z}$ and $\mathbf{z}'$ are the same size (r) and represent prohibitable events in exactly the same order.

For input sequence $\{\mathbf{i}_g(k'')\}$, let $\mathbf{z}(k) \in Z$ and $\mathbf{z}'(k) \in Z'$ be the induced output vector at time k for controllers $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ respectively.

Let $\mathbf{q}(k) \in Q$ and $\mathbf{q}'(k) \in Q'$ be the induced state vectors at time k for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ respectively.

Now, we will prove our main result.

To show that $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ are output equivalent with respect to $L(\mathbf{G}_{cl})$, by Definition 7.2 we need to show:

1. Both output vectors $\mathbf{z}$ and $\mathbf{z}'$ are of size r.

2. $(\forall\, 0 \leq i < r)\, \eta^{-1}(i) = \eta'^{-1}(i)$

3. $(\forall k \in \{0, 1, 2, \ldots\})\, \mathbf{z}(k) = \mathbf{z}'(k)$

We note that Points 1 and 2 follow immediately from (4) and (5) respectively.

Now all that remains is to show: $(\forall k \in \{0, 1, 2, \ldots\})\, \mathbf{z}(k) = \mathbf{z}'(k)$

Let $k \in \{0, 1, \ldots\}$.

We first note that by the TDES to FSM translation method (Section 3.7), we have:

$$\mathbf{z}(k) = \Phi(\mathbf{q}(k)) \text{ and } \mathbf{z}'(k) = \Phi'(\mathbf{q}'(k))$$

By definition of Φ and Φ', we have:

$$\begin{gathered}\mathbf{z}(k) = \Phi(\mathbf{q}(k)) = \Gamma_Z(\zeta(x)) \text{ and } \mathbf{z}'(k) = \Phi'(\mathbf{q}'(k)) = \Gamma_{Z'}(\zeta'(x')) \\ \text{where } \mathbf{q}(k) = \Lambda(x) \text{ and } \mathbf{q}'(k) = \Lambda'(x') \text{ for some } x \in X_{samp} \text{ and } x' \in X'_{samp}\end{gathered} \tag{7}$$

As $\Gamma_Z = \Gamma_{Z'}$ by (6), all we need to complete the proof is to construct a suitable $x \in X_{samp}$ and $x' \in X'_{samp}$ and show that $\zeta(x) = \zeta'(x')$.

We have two cases: **(1)** $k = 0$, and **(2)** $k \in \{1, 2, \ldots\}$.

Case 1) $k = 0$

By definition of the TDES to FSM translation method (Section 3.7), we have:

$$\mathbf{q}(0) = \mathbf{q}_{res} = \Lambda(x_o) \text{ and } \mathbf{q}'(0) = \mathbf{q}'_{res} = \Lambda'(x'_o) \tag{8}$$

Let $s = \epsilon$.

$\Rightarrow \xi(x_o, s) = x_o$ and $\xi'(x'_o, s) = x'_o$ (9)

As $s = \epsilon \in L_{samp} = \{\epsilon\} \cup \Sigma^*.\tau$, and $\mathbf{S}$ and $\mathbf{G}$ have initial states implies that $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has an initial state, it follows that $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$.

By applying Proposition 7.2, we conclude: $\zeta(\xi(x_o, s)) = \zeta'(\xi'(x'_o, s))$

$\Rightarrow \zeta(x_o) = \zeta'(x'_o)$ *by* (9)

We note that by Definition 3.2 of sampled states, initial states are always sampled states. We then take $x = x_o$ and $x' = x'_o$. We thus have $x \in X_{samp}$ and $x' \in X'_{samp}$, $\mathbf{q}(k) = \Lambda(x)$, $\mathbf{q}'(k) = \Lambda'(x')$ and $k = 0$ by (8), and $\zeta(x) = \zeta'(x')$.

Case 2) $k \in \{1, 2, \ldots\}$

As $\{\mathbf{i}_g(k'')\}$ is input valid for $L(\mathbf{G}_{cl})$ by (3), we have:

$$\begin{gathered}(\exists s_1, s_2, \ldots, s_k \in L_{conc})\, [s_1 s_2 \ldots s_k \in L(\mathbf{G}_{cl})] \wedge \\ [(\forall n \in \{1, 2, \ldots, k\})\, (\forall \sigma \in \Sigma_{act})\, i_{g,\gamma_g(\sigma)}\,(n) = 1 \text{ iff } \sigma \in Occu(s_n)]\end{gathered}$$

Let $s = s_1 s_2 \ldots s_k$, and we have $s \in L_{samp}$ as $L_{conc} = \Sigma^*_{act}.\tau$ (Definition 3.1). (10)

As $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$ and $\boldsymbol{\mathcal{S}'} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ by (1), we have: $s \in L(\boldsymbol{\mathcal{S}'}) \cap L_{samp}$ (11)

As $L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \subseteq L(\mathbf{S})$ by Proposition 5.1, we have: $s \in L(\mathbf{S}) \cap L_{samp}$

By applying Proposition 3.2, we conclude: $\mathbf{q}(k) = \Lambda(\xi(x_o, s))$ and $\mathbf{q}'(k) = \Lambda'(\xi'(x'_o, s))$ (12)

Let $x = \xi(x_o, s)$ and $x' = \xi'(x'_o, s)$. (13)

$\Rightarrow x \in X_{samp}$ and $x' \in X'_{samp}$ *as* $s \in L_{samp}$ *by* (10) (14)

As $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ by (11), by applying Proposition 7.2 we conclude:

$\zeta(\xi(x_o, s)) = \zeta'(\xi'(x'_o, s))$

$\Rightarrow \zeta(x) = \zeta'(x')$ *by* (13)

We thus have $x \in X_{samp}$ and $x' \in X'_{samp}$ by (14), $\mathbf{q}(k) = \Lambda(x)$ and $\mathbf{q}'(k) = \Lambda'(x')$ by (12) and (13), and $\zeta(x) = \zeta'(x')$.

By Cases (1) and (2), we have constructed a suitable x and x' with $\zeta(x) = \zeta'(x')$.

We thus conclude by (7) that $\mathbf{z}(k) = \mathbf{z}'(k)$, as required.

Hence, we conclude that $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ are output equivalent with respect to the closed-loop behaviour $L(\mathbf{G}_{cl})$. □

We close this section with a proposition that we will find useful in Section 8. It essentially shows that SD controllers $\boldsymbol{\mathcal{C}}$ and $\boldsymbol{\mathcal{C}'}$ will produce the same output for every sampled string accepted by the closed-loop system, $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$.

Proposition 7.3. Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$, and let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES supervisor $\boldsymbol{\mathcal{S}'} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ constructed using Algorithms 2 and 3 be CS deterministic. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller translated from $\mathbf{S}$, and $\boldsymbol{\mathcal{C}'} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller translated from $\boldsymbol{\mathcal{S}'}$. Let Λ and Λ' be the state mapping functions for $\boldsymbol{\mathcal{C}}$ and $\boldsymbol{\mathcal{C}'}$ respectively. Then:

$$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp})\ \Phi(\Lambda(\xi(x_o, s))) = \Phi'(\Lambda'(\xi'(x'_o, s)))$$

Proof. Assume initial conditions.

Let ζ and ζ' be the prohibited action functions (Definition 3.10) for $\mathbf{S}$ and $\boldsymbol{\mathcal{S}'}$ respectively.

Let η and η' be the bijective output event mapping functions (Definition 3.13) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ respectively.

Let Γ_Z and $\Gamma_{Z'}$ be the bijective output set mapping functions (Definition 3.17) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ respectively.

Let $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$. (1)

Applying Proposition 7.2, we conclude: $\zeta(\xi(x_o, s)) = \zeta'(\xi'(x'_o, s))$ (2)

Let $\mathbf{q} = \Lambda(\xi(x_o, s))$ and $\mathbf{q}' = \Lambda'(\xi'(x'_o, s))$. (3)

Applying Theorem 7.1, we can conclude that $\mathbf{C}$ and $\boldsymbol{\mathcal{C}'}$ are output equivalent with respect to $L(\mathbf{G}_{cl})$.

This implies $\eta = \eta'$, and thus $\Gamma_Z = \Gamma_{Z'}$, as they are defined in terms of η and η' respectively. (4)

By Definition 3.19 of Φ, we have: $\Phi(\mathbf{q}) = \Gamma_Z(\zeta(x))$ if $(\exists x \in X_{samp})\,\mathbf{q} = \Lambda(x)$

As $s \in L_{samp}$ by (1), we can take $x = \xi(x_o, s)$ and we have $x \in X_{samp}$, and $\Lambda(x) = \mathbf{q}$ by (3).

We thus have: $\Phi(\mathbf{q}) = \Gamma_Z(\zeta(\xi(x_o, s)))$

Similarly, we can take $x' = \xi'(x'_o, s)$ and we have $x' \in X'_{samp}$, and $\Lambda'(x') = \mathbf{q}'$ by (3).

$\Rightarrow \Phi'(\mathbf{q}') = \Gamma_{Z'}(\zeta'(\xi'(x'_o, s)))$

As $\Gamma_Z = \Gamma_{Z'}$ by (4), and $\zeta(\xi(x_o, s)) = \zeta'(\xi'(x'_o, s))$ by (2), we have: $\Phi(\mathbf{q}) = \Phi'(\mathbf{q}')$

$\Rightarrow \Phi(\Lambda(\xi(x_o, s))) = \Phi'(\Lambda'(\xi'(x'_o, s)))$ *by* (3) □

8 Controllability and Nonblocking Results for SD Synchronous Product Setting

In this section, we present the controllability and nonblocking verification results for our $||_{SD}$ setting. This section begins with the construction of a TDES supervisory control V, stating the relevant definitions, and proving its various properties. After that, we thoroughly describe and formally prove our controllability and nonblocking results. Essentially, we show that if our theoretical $||_{SD}$ system is controllable, nonblocking and abide by the specified control laws, then the physically implemented system will also have these properties, given that the $||_{SD}$ system satisfies our adapted properties that were originally identified by the SD supervisory control methodology.

Please note that in this section, we will use the notation of TDES supervisor $\mathbf{S}$, SD controller $\mathbf{C}$ and TDES supervisory control V while discussing about our $||_{SD}$ setting. The notation of TDES supervisor $\boldsymbol{\mathcal{S}}$, SD controller $\boldsymbol{\mathcal{C}}$ and TDES supervisory control $\mathcal{V}$ will be used while referring to the SD setting, and they map to $\mathbf{S}$, $\mathbf{C}$ and V of Section 3 respectively. In this section, whenever we refer to an SD controller constructed from a supervisor, we will always assume that it is translated using the method described in Section 3.7.

From this section onwards, we will take our SD supervisor $\boldsymbol{\mathcal{S}}$ to be $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$, i.e. the minimal version of $\mathbf{S}\,||_{SD}\,\mathbf{G}$ that is constructed using Algorithms 2 and 3. The reason is that if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is already minimal, there is no change. However, if it is not already minimal, we must minimize it to ensure that $\boldsymbol{\mathcal{S}}$ is CS deterministic. By simply assuming that we are always using the minimal version will keep things simple.

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a plant, TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a supervisor, $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be an SD controller, and the closed-loop system of our $||_{SD}$ setting be $\mathbf{S}\,||_{SD}\,\mathbf{G}$. For the rest of this section, we require our system to satisfy the following properties: 1) $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces and finite event sets, 2) $\mathbf{G}$ has proper time behaviour, 3) $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$, 4) $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, 5) $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF, 6) $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$, 7) $\mathbf{S}$ is CS deterministic, and 8) $\mathbf{C}$ is an SD controller translated from $\mathbf{S}$ using the translation method described in Section 3.7.

By looking at Proposition 4.5, it is evident that these conditions are sufficient to guarantee that our system will not "stop the clock", i.e. for any string $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, our $||_{SD}$ system will always be able to do a *tick* event after at most a finite number of activity events. This ensures that after a sampled string, all new behaviour of the system can be represented as a sequence of concurrent strings.

8.1 Supervisory Control V

In the SD supervisory control theory [42, 29], the authors have pointed out that an SD controller is more constrained than a TDES supervisor. This is due to the fact that an SD controller only changes state on the occurrence of the *tick* event, whereas a supervisor can do so every time an event occurs. This in turn implies that the enablement and forcing information of an SD controller does not always exactly match with that of a TDES supervisor.

To address this issue in the SD setting, a TDES supervisory control is used to express the enablement and forcing behaviour of an SD controller in terms of strings. In [42], the authors presented Algorithm 1 to construct this supervisory control. This algorithm's definition is then used to argue about the behaviour of the SD controller in various controllability and nonblocking verification proofs of the SD setting. Since we are building our work on the SD

supervisory control methodology, we will adopt the same approach to capture the control action of SD controller in our $||_{SD}$ setting and prove our desired results.

In this section, we first discuss the construction of a TDES supervisory control V in our $||_{SD}$ setting (we will formally prove that V is indeed a TDES supervisory control later in Proposition 8.4). Specifically, we explain how we have adapted Algorithm 1 to make it compatible with our $||_{SD}$ setting. Then, we present some definitions in relation to our V. Finally, we prove some properties with respect to V that will help us in proving our $||_{SD}$ controllability and nonblocking verification results afterwards.

8.1.1 Construction of V

Note: To be clear in our discussion and avoid any ambiguity, we will refer to $\mathbf{S}$, $\mathbf{C}$, V and Σ_V of Algorithm 1 as $\boldsymbol{\mathcal{S}}$, $\boldsymbol{\mathcal{C}}$, $\mathcal{V}$ and $\Sigma_{\mathcal{V}}$ respectively.

In order to construct TDES supervisory control V from our SD controller $\mathbf{C}$ in the $||_{SD}$ setting, we adapt Algorithm 1 from [42]. Our algorithm for the $||_{SD}$ setting is presented as Algorithm 4. It is worth-mentioning that the two algorithms are logically identical, although they differ at **line 14**, where $L(\boldsymbol{\mathcal{S}})$ of Algorithm 1 has been replaced by $L(\mathbf{S} \,||_{SD}\, \mathbf{G})$ in Algorithm 4.

Please note that the complete description of Algorithm 1 to construct TDES supervisory control from an SD controller is given in Section 3.8. Most of the items given in Algorithm 4 are defined in Section 3.7. The map of *Occu* is defined in Section 3.4, and TDES supervisory control and $CB_{\mathbf{G}}$ are defined in Section 3.8. In this section, we only focus on explaining and comparing those aspects of the two algorithms that differ and need clarification.

For all strings $s \in L(\mathbf{G})$, Algorithm 1 sets the default enablement information at **lines 1-3** by adding all uncontrollable events and *tick* event to $\mathcal{V}(s)$. This is done to satisfy Definition 3.20 of TDES supervisory control. As this definition is given only in terms of $L(\mathbf{G})$, it remains valid in our $||_{SD}$ setting as well. Therefore, this part of Algorithm 1 remains unchanged in Algorithm 4.

By looking at Algorithm 1, we note that it updates the default enablement information only for those strings that represent valid behaviour in the closed-loop system. These strings are identified at **lines 13-14**. **Line 13** of Algorithm 1 considers all possible concurrent strings s' that extend a sampled string s in $L(\mathbf{G})$. The **if** statement at **line 14** then uses the following two conditions to filter out those strings that do not meet the required criteria.

1) $Occu(s') \cap \Sigma_{hib} \subseteq \Sigma_{\mathcal{V}}$

This condition excludes concurrent strings that are possible in the closed behaviour of $\mathbf{G}$, but their occurrence images contain prohibitable events that are not in $\Sigma_{\mathcal{V}}$. As these prohibitable events are disabled by controller $\boldsymbol{\mathcal{C}}$, therefore these strings will not occur in the physical system.

Since we are using the translation method of the SD setting to generate our SD controller $\mathbf{C}$ from TDES supervisor $\mathbf{S}$, therefore this condition does not need to be changed for our $||_{SD}$ setting, and shows up as it is in Algorithm 4.

2) $ss' \in L(\boldsymbol{\mathcal{S}})$

In Algorithm 1, this condition disregards concurrent strings that do not represent valid behaviour in $L(\boldsymbol{\mathcal{S}})$ after sampled string s. This ensures to restrict the set of valid strings to concurrent strings that are accepted by the supervisor. Ultimately, it results in restricting the valid strings overall to $L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$ in the SD setting.

Algorithm 4 Obtaining V from Controller $\mathbf{C}$, Acting on Plant $\mathbf{G}$

1: **for all** $s \in L(\mathbf{G})$ **do**
2: $\quad V(s) \leftarrow \Sigma_u \cup \{\tau\}$
3: **end for**
4: $Pend \leftarrow \{(\epsilon, \mathbf{q}_{res})\}$
5: **while** $Pend \neq \emptyset$ **do**
6: $\quad (s, \mathbf{q}) \leftarrow$ a member from $Pend$
7: $\quad Pend \leftarrow Pend - \{(s, \mathbf{q})\}$
8: $\quad \mathbf{z} \leftarrow \Phi(\mathbf{q})$
9: $\quad \Sigma_V \leftarrow \Gamma_Z^{-1}(\mathbf{z})$
10: $\quad$ **if** $\Sigma_V \neq \emptyset$ **then**
11: $\quad\quad V(s) \leftarrow (V(s) \cup \Sigma_V) - \{\tau\}$
12: $\quad$ **end if**
13: $\quad$ **for all** $s' \leftarrow \sigma_1\sigma_2 \ldots \sigma_j \in CB_{\mathbf{G}}(s)$ **do** // $\sigma_j = \tau$, *by definition of* L_{conc}
14: $\quad\quad$ **if** $(Occu(s') \cap \Sigma_{hib} \subseteq \Sigma_V) \wedge (ss' \in L(\mathbf{S} \,||_{SD}\, \mathbf{G}))$ **then**
15: $\quad\quad\quad \Sigma_{temp} \leftarrow \Sigma_V$
16: $\quad\quad\quad \mathbf{i} \leftarrow \Gamma_I(Occu(s') - \{\tau\})$
17: $\quad\quad\quad \mathbf{q}' \leftarrow \Omega(\mathbf{q}, \mathbf{i})$
18: $\quad\quad\quad Pend \leftarrow Pend \cup \{(ss', \mathbf{q}')\}$
19: $\quad\quad\quad$ **if** $j > 1$ **then**
20: $\quad\quad\quad\quad$ **for** $i \leftarrow 1$ to $j - 1$ **do**
21: $\quad\quad\quad\quad\quad \Sigma_{temp} \leftarrow \Sigma_{temp} - \sigma_i$
22: $\quad\quad\quad\quad\quad$ **if** $\Sigma_{temp} \neq \emptyset$ **then**
23: $\quad\quad\quad\quad\quad\quad V(s\sigma_1\sigma_2 \ldots \sigma_i) \leftarrow (V(s\sigma_1\sigma_2 \ldots \sigma_i) \cup \Sigma_V) - \{\tau\}$
24: $\quad\quad\quad\quad\quad$ **else**
25: $\quad\quad\quad\quad\quad\quad V(s\sigma_1\sigma_2 \ldots \sigma_i) \leftarrow (V(s\sigma_1\sigma_2 \ldots \sigma_i) \cup \Sigma_V)$
26: $\quad\quad\quad\quad\quad$ **end if**
27: $\quad\quad\quad\quad$ **end for**
28: $\quad\quad\quad$ **end if**
29: $\quad\quad$ **end if**
30: $\quad$ **end for**
31: **end while**
32: **return** V

Using the same logic for our $||_{SD}$ setting, we want to restrict the set of valid strings to our closed-loop behaviour, $L(\mathbf{S} \,||_{SD}\, \mathbf{G})$. In order to do that, we cannot simply replace $L(\boldsymbol{\mathcal{S}})$ at **line 14** of Algorithm 1 with $L(\mathbf{S})$ in our Algorithm 4. This is due to the fact that in the SD setting, supervisor $\boldsymbol{\mathcal{S}}$ is solely responsible for the enablement/disablement of *tick* event in the closed-loop system. However, in our $||_{SD}$ setting, the task of enabling/disabling the *tick* event is *cooperatively* performed by supervisor $\mathbf{S}$ and $||_{SD}$ operator. In our case, a *tick* event that is possible in $\mathbf{G}$ might be enabled by $\mathbf{S}$ too. Still, this *tick* event might not be possible in the closed-loop system as our $||_{SD}$ operator is authorized to remove *tick* from the closed-loop system in the presence of enabled prohibitable events. To further clarify, in most cases, our closed-loop behaviour $L(\mathbf{S} \,||_{SD}\, \mathbf{G}) \neq L(\mathbf{S}) \cap L(\mathbf{G})$ due to the synchronization mechanism of our $||_{SD}$ operator.

For this reason, in order to restrict the set of valid strings to our closed-loop behaviour,

we have replaced their $L(\boldsymbol{\mathcal{S}})$ with our $L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ at **line 14**. By making this change, we guarantee that if a string does not represent valid behaviour in our closed-loop system, then its enablement information, once assigned at **line 2**, will remain unmodified throughout the execution of Algorithm 4.

It is worth clarifying that this replacement does not change the original logic of Algorithm 1 for constructing supervisory control from the SD controller. In fact, this change at **line 14** actually ensures that the original logic of Algorithm 1 remains untouched in Algorithm 4. We will formally prove this in Proposition 8.2 by showing that the two supervisory controls V and $\mathcal{V}$ constructed using Algorithms 4 and 1 respectively are equal with respect to a given plant $\mathbf{G}$.

Another way, probably an easier and straightforward one, to look at this modification at **line 14** is that in our $||_{SD}$ setting, we have concretely defined $\boldsymbol{\mathcal{S}} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, or $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ for that matter. We have already proven in our previous sections that $\boldsymbol{\mathcal{S}}$ possesses all the required properties and does qualify to be used as the supervisor of the SD setting. Since $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, replacing $L(\boldsymbol{\mathcal{S}})$ with $L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ does not bring any logical change at **line 14**, as the two closed languages are same. Therefore, both algorithms will restrict the update of enablement information to the same set of valid strings in the closed-loop behaviour due to the way we have constructed $\boldsymbol{\mathcal{S}}$ in the $||_{SD}$ setting.

Another important point that we want to highlight is about **line 11** of Algorithm 1. If any prohibitable event is enabled at state $\mathbf{q}'$ in $\boldsymbol{\mathcal{C}}$ (**line 10**), this prohibitable event needs to be forced in the current sampling period. Therefore, **line 11** removes *tick* event from $\mathcal{V}(s)$ to satisfy Point ii ($\Rightarrow$) of the SD controllability definition. This *tick* was added at **line 2** while initializing $\mathcal{V}(s)$ with its default enablement information.

It is worth recalling here that Point ii ($\Rightarrow$) of the SD controllability definition does not exist in our definition of SD controllability with $||_{SD}$ property, and we are not checking this condition explicitly in our $||_{SD}$ setting. We are able to get rid of this explicit check because of the synchronization mechanism of our $||_{SD}$ operator that guarantees to automatically satisfy this condition while forming the closed-loop system. Therefore, although *tick* event does get removed at **line 11** in Algorithm 4, it is for a different reason. In our case, this removal of *tick* is not to satisfy any point of the SD controllability with $||_{SD}$ definition. Rather, it is to keep things consistent with the synchronization mechanism used by our $||_{SD}$ operator to construct the closed-loop system; hence, **line 11** remains unmodified in Algorithm 4.

8.1.2 Preliminary Definitions

In order to define the closed behaviour of $V/\mathbf{G}$, represented as $L(V/\mathbf{G})$, Definition 2.24 uses TDES plant $\mathbf{G}$ and supervisory control V. This definition neither takes into account the supervisor model nor the synchronization operator while defining $L(V/\mathbf{G})$; thus, this definition remains valid for our $||_{SD}$ setting and does not need to be redefined. Below, we present some definitions in relation to V that are specific to our $||_{SD}$ setting.

Definition 8.1. For TDES plant $\mathbf{G}$ and CS deterministic TDES supervisor $\mathbf{S}$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let $\mathbf{C}$ be the SD controller constructed from $\mathbf{S}$ using the translation method described in Section 3.7, and let V be the map constructed from $\mathbf{C}$ using Algorithm 4. In the $||_{SD}$ setting, the *marked behaviour* of $V/\mathbf{G}$, represented as $L_m(V/\mathbf{G})_{||_{SD}}$, is defined as:

$$L_m(V/\mathbf{G})_{||_{SD}} := L(V/\mathbf{G}) \cap L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$$

Definition 8.2. In the $||_{SD}$ setting, V is said to be *nonblocking for* $\mathbf{G}$ if:

$$\overline{L_m(V/\mathbf{G})_{||_{SD}}} = L(V/\mathbf{G})$$

8.1.3 Map V is Well Defined

In [42], map $\mathcal{V}$ generated from SD controller $\boldsymbol{\mathcal{C}}$ using Algorithm 1 is shown to be well defined. Since we have modified Algorithm 1 to suit our needs, it is important to show the same result in our $||_{SD}$ setting so that we can consider V as a potential TDES supervisory control.

The proposition given below proves that map V constructed from SD controller $\mathbf{C}$ using Algorithm 4 is well defined. As Algorithms 1 and 4 are logically identical, we have taken the basic idea of this proof from [42] to prove our desired result.

Proposition 8.1. For TDES plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, and CS deterministic TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let $\mathbf{C}$ be the SD controller constructed from $\mathbf{S}$ using the translation method described in Section 3.7, and let V be the map constructed from $\mathbf{C}$ using Algorithm 4. Then, map V is well defined.

Proof. Assume initial conditions.

In order to show that map V is well defined, we need to show that for all $s \in L(\mathbf{G})$, Algorithm 4 defines $V(s)$ in only one way. We will show this by analyzing the logic used by Algorithm 4 to construct $V(s)$ from $\mathbf{C}$.

By examining Algorithm 4, first we note that for all $s \in L(\mathbf{G})$, the algorithm initializes $V(s)$ at **line 2**, and then potentially updates it at **lines 11**, **23** and **25**.

Further examination reveals that for all $s \notin \overline{L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}}$, the algorithm adds Σ_u and $\{\tau\}$ to $V(s)$ at **line 2**, and these strings are not evaluated again in the algorithm. This means for all such s, $V(s)$ is defined only once at **line 2**. Therefore, it is evident that for all $s \notin \overline{L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}}$, $V(s)$ is well defined.

Now we will analyze all the remaining strings s, such that $s \in \overline{L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}}$.

Let $s \in \overline{L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}}$

$\Rightarrow (\exists u \in \Sigma^*)\; su \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ *by definition of prefix closure of* L

$\Rightarrow su \in L(\mathbf{S}) \cap L(\mathbf{G}) \cap L_{samp}$ *by Proposition* 5.1

$\Rightarrow su \in L(\mathbf{S}) \cap L_{samp}$

$\Rightarrow s \in L(\mathbf{S})$ *as* $L(\mathbf{S})$ *is a prefix-closed language* (1)

After **line 2**, the enablement information of $V(s)$ can be modified at **lines 11**, **23** and **25** of the algorithm. By analyzing these lines, we observe that **line 11** updates $V(s)$ if $s \in L_{samp}$. Otherwise, if $s \notin L_{samp}$, then $V(s)$ could possibly be updated once or more at **line 23** or **25**. Thus, we have two cases: **(1)** $s \in L_{samp}$, and **(2)** $s \notin L_{samp}$.

Case 1) $s \in L_{samp}$

Algorithm 4 evaluates string-state pairs $(s, \mathbf{q})$, by retrieving them one by one from the set *Pend* at **line 6**. This means the algorithm re-evaluates $V(s)$ of only those strings that were added to *Pend*.

If a sampled string s is never added to *Pend*, its $V(s)$ cannot be modified at **line 11**. Such sampled strings will retain their default enablement information that was assigned to them at **line 2**. Hence, for such s, $V(s)$ will always be well defined.

Thus, without any loss of generality, we assume that s was added to *Pend* at some point during the execution of Algorithm 4.

Line 11 updates $V(s)$ by adding the set of enabled prohibitable events, Σ_V, and removing the *tick* event. As we want to show that the algorithm defines $V(s)$ in only one way, it is sufficient to show that whenever **line 11** is executed for s, we always have the same Σ_V to append to $V(s)$. Clearly, as long as Σ_V is the same, executing **line 11** once or more will not make any difference, as $V(s)$ will be updated in the same way every time.

By reviewing the algorithm, we note that Σ_V is formed from output vector $\mathbf{z}$ of controller $\mathbf{C}$ at **line 9**. This output vector $\mathbf{z}$ is in turn obtained from state $\mathbf{q}$ of $\mathbf{C}$ at **line 8**. This means that Σ_V is uniquely defined by state $\mathbf{q}$. Thus, it is sufficient to show that sampled string s will always be paired with state $\mathbf{q}$ of controller $\mathbf{C}$.

As $s \in L_{samp}$, by Definition 3.1 of L_{samp}, we have two possible cases: **(a)** $s = \epsilon$, and **(b)** $s \in \Sigma^*.\tau$.

Case 1.a) $s = \epsilon$

The controller $\mathbf{C}$ always starts at its initial or reset state, $\mathbf{q}_{res}$. By the definition of SD controller, $\mathbf{q}_{res}$ corresponds to the empty string, ϵ.

From **line 4** of the algorithm, it is clear that ϵ is always paired with state $\mathbf{q}_{res}$ of $\mathbf{C}$.

Hence, we conclude that if $s = \epsilon$, then s is always paired with the same state $\mathbf{q}_{res}$ of $\mathbf{C}$.

Case (1.a) complete.

Case 1.b) $s \in \Sigma^*.\tau$

By examining the algorithm, we note that for every string-state pair $(s, \mathbf{q})$ added to *Pend*, the non-empty sampled string s of the pair is constructed by concatenating one or more concurrent strings together. Thus, for every such s, we have:

$$(\exists n \in \{1, 2, \ldots\})\,(\exists s_1, s_2, \ldots, s_n \in L_{conc})\; s_1 s_2 \ldots s_n = s$$

By Definition 3.3 of concurrent string, we have: $L_{conc} = \Sigma_{act}^*.\tau$

This implies that for a given sampled string s, there is only one way to define the sequence of concurrent strings $s_1 s_2 \ldots s_n$. In other words, the sequence of concurrent strings $s_1 s_2 \ldots s_n$ in one sampled string s will always be the same.

Except for the first pair $(\epsilon, \mathbf{q}_{res})$, all string-state pairs are added to *Pend* at **line 18**. These pairs are determined at **lines 16** and **17** of the algorithm. These two lines show that starting from the initial state $\mathbf{q}_{res}$, each subsequent state of $\mathbf{C}$ is determined by the current state and the occurrence image of the next concurrent string which is possible in the closed-loop system.

As $\mathbf{S}$ is a CS deterministic supervisor and $s \in L(\mathbf{S}) \cap L_{samp}$ by (1), by the definition of translation functions Γ_I (Definition 3.16), Ω (Definition 3.18), Λ (Definition 3.15) and Δ (Definition 3.9), it is evident that the sequence of states reached by the sequence of concurrent strings $s_1 s_2 \ldots s_n$ will be unique. This implies the state $\mathbf{q}$ of controller $\mathbf{C}$ that is reached by sampled string $s = s_1 s_2 \ldots s_n$ will also be unique.

Hence, we conclude that if $s = \Sigma^*.\tau$, then s is always paired with the same state $\mathbf{q}$ of $\mathbf{C}$.

Case (1.b) complete.

By Cases (1.a) and (1.b), we have shown that sampled string s will always be paired with the same state $\mathbf{q}$ of controller $\mathbf{C}$. In other words, whenever **line 11** is executed for $s \in L_{samp}$, we always have same Σ_V to append to $V(s)$.

Hence, we conclude that for $s \in L_{samp}$, Algorithm 4 defines $V(s)$ in only one way.

Case (1) complete.

Case 2) $s \notin L_{samp}$

If $s \notin L_{samp}$, this implies: $(\exists t \in L_{samp})\,(\exists t' \in L_{conc})\, t < s < tt'$

This in turn implies: $(\exists j > 1)\,(\exists \sigma_1, \ldots, \sigma_j \in \Sigma)\, t' = \sigma_1 \ldots \sigma_j$

As $t' \in L_{conc}$, by the definition of L_{conc}, $\sigma_j = \tau$.

We thus have: $(\exists i \in \{1, \ldots, j-1\})\, t\sigma_1, \ldots, \sigma_i = s$

In the above setting, we have $j > 1$. This is because if we consider $j = 0$ or $j = 1$, then $t < s < tt'$ would cause a contradiction.

If $j = 0$, then $t' = \epsilon$. As $t' \in L_{conc}$, by the definition of L_{conc}, $t' \neq \epsilon$. Moreover, $t' = \epsilon$ implies $tt' = t$. In this case, we would have $t < s < t$, that could not be true in any case.

If $j = 1$, then $t' = \tau$. Since we require $t < s$, s must contain at least one event more than t, and since $s \notin L_{samp}$, $s \neq \epsilon$ and must not end with a τ. As t' contains only one event, τ, this would not allow $s < tt'$ and $s \notin L_{samp}$. Thus, we must have $j > 1$.

We note that in Algorithm 4, if: i) t was never added to $Pend$, or ii) t was added to $Pend$ but for all such t' discussed above, if t' fail the condition at **line 14**, then $V(s)$ will never be updated in the algorithm after its initialization. This implies that $V(s)$ will keep the value assigned to it on **line 2**, even after the complete execution of the algorithm. In such case, we know that $V(s)$ will be well-defined.

Thus, without any loss of generality, we assume that t was added to $Pend$, and our t' passes the condition at **line 14**.

This implies: $t, tt' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$

We have: $t' = \sigma_1 \ldots \sigma_i\, \sigma_{i+1} \ldots \sigma_j \in L_{conc}$

As $s = t\sigma_1 \ldots \sigma_i$, it is obvious, from the definition of L_{conc}, that there is only one way to define activity events $\sigma_1 \ldots \sigma_i$, and thus the sampled string t. This implies that there is only one way to define $s = t\sigma_1 \ldots \sigma_i$. Of course, it is possible that there might be multiple ways to define $\sigma_{i+1} \ldots \sigma_j$.

From the result of Case (1), we know that whenever **line 11** is executed for a given $t \in L_{samp}$, we always have the same Σ_V to append to $V(t)$.

By examining Algorithm 4, we note that for $s \notin L_{samp}$, the portion of the algorithm that we are interested in, with respect to the modification of $V(s)$, is defined from **lines 19-28**.

In this section, we see that **lines 23** and **25** update $V(s)$ by appending Σ_V, which we know will always be same for $t \in L_{samp}$. In addition, $V(s)$ is determined by $t\sigma_1, \ldots, \sigma_i$ which is also unique for our s, as discussed above. Thus, it is evident that whenever **line 23** or **25** is executed, we always get the same updated $V(s)$.

Hence, we conclude that for $s \notin L_{samp}$, Algorithm 4 defines $V(s)$ in only one way.

Case (2) complete.

By Cases (1) and (2), we have shown that for all $s \in \overline{L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}}$, Algorithm 4 defines $V(s)$ in only one way.

Thus, we have shown that for all $s \in L(\mathbf{G})$, $V(s)$ is well defined.

Hence, we conclude that map V, constructed from SD controller $\mathbf{C}$ using Algorithm 4, is well defined. □

8.1.4 Equivalence of V and $\mathcal{V}$

As discussed in Section 8.1.1, Algorithm 4 is logically equivalent to Algorithm 1 in its way of constructing TDES supervisory control from the SD controller. By Theorem 7.1, we know that the two SD controllers $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$, of $||_{SD}$ and SD setting respectively, are output equivalent with respect to the closed-loop behaviour, $\mathbf{S}\,||_{SD}\,\mathbf{G}$. This means that for a given plant $\mathbf{G}$, two maps V and $\mathcal{V}$ constructed from SD controllers $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ using Algorithms 4 and 1 should also be equivalent. This is formally proven in our next proposition.

This equivalence result essentially bridges the gap between two settings in terms of their supervisory controls. This further paves our way for reusing some of the existing SD results in deriving and concluding our controllability and nonblocking verification results presented in the next section.

Proposition 8.2. Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant to be controlled. Let TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$, and let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES supervisor $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ constructed using Algorithms 2 and 3 be CS deterministic. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and V be the map constructed from $\mathbf{C}$ using Algorithm 4. Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 1. Then, $V = \mathcal{V}$.

Proof. Assume initial conditions.

We first note that $L(\boldsymbol{\mathcal{S}}) = L(min(\mathbf{S}\,||_{SD}\,\mathbf{G})) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, as state minimization does not change the closed-loop behaviour of an automaton. (1)

We next note that Algorithm 1 will be applied to $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{C}}$, while Algorithm 4 will be applied to $\mathbf{S}$ and $\mathbf{C}$.

We note that Algorithms 1 and 4 are identical except for **line 14**, where Algorithm 1 has $ss' \in L(\boldsymbol{\mathcal{S}})$ and Algorithm 4 has $ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$.

However, as $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ by (1), **line 14** is now identical for both. Hence, the two algorithms now only differ by the fact that Algorithm 4 is applied to $\mathbf{C}$ while Algorithm 1 is applied to $\boldsymbol{\mathcal{C}}$. (2)

We will now show that we can replace controller $\boldsymbol{\mathcal{C}}$ by $\mathbf{C}$ in Algorithm 1, and $V = \mathcal{V}$ will immediately follow.

First, we need to prove the following claim.

Claim: In the tuples added to *Pend* in either algorithm, the string t of the tuple will always satisfy: $t \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$

As for our purpose, the two algorithms are equal by (2), we will examine Algorithm 4 but the result will equally apply to Algorithm 1.

We will prove this by induction.

Base Case:

We first note that at **line 4**, *Pend* is initialized to $(\epsilon, \mathbf{q}_{res})$. We thus have $\epsilon \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$, as $\epsilon \in L_{samp}$ by Definition 3.2, and as $\mathbf{S}$ and $\mathbf{G}$ have initial states, and by Definition 4.1 of the $||_{SD}$ operator.

Inductive Step:

Show: $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ at **line 6** $\Rightarrow ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ at **line 18**

Assume: $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ at **line 6**

For ss' to reach **line 18**, we have $s' \in CB_{\mathbf{G}}(s)$ and $ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ from **lines 13** and **14**.
$\Rightarrow ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $s' \in L_{conc}$ *by Definition* 3.21 *of* $CB_{\mathbf{G}}$
$\Rightarrow ss' \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ *by Definition* 3.3 *of* L_{conc}

By base case and inductive step, we conclude that each string t of the tuple at **line 6** satisfies $t \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$.

Claim proven.

This implies that for both algorithms, we only care about the outputs of controllers $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ for strings $t \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$.

Applying Proposition 7.3, it follows that $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ provide exactly the same output for strings $t \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$.

This means we can replace controller $\boldsymbol{\mathcal{C}}$ by controller $\mathbf{C}$ in Algorithm 1 without affecting the algorithm.

The application of Algorithms 1 and 4 are now identical, so we immediately have $V = \mathcal{V}$, as required. □

8.2 Controllability and Nonblocking Verification

This section presents our controllability and nonblocking verification results for the $||_{SD}$ setting. Essentially, we show that the behaviour of TDES plant $\mathbf{G}$ under the action of SD controller $\mathbf{C}$ is same as the behaviour of $\mathbf{G}$ under the supervision of TDES supervisor $\mathbf{S}$, given that $||_{SD}$ system satisfies the properties that are stated in the beginning of this section. Our results clearly indicate that if the theoretical $||_{SD}$ system is controllable, nonblocking and satisfies the specified properties, then the physically implemented system will also have these properties, and the SD controller will behave as expected with respect to control action, event forcing and nonblocking.

As discussed before, instead of proving all results from scratch, we will use the existing SD results to derive and conclude some of our formal $||_{SD}$ verification results. We will do this by utilizing the equivalence that we have established between the SD and our $||_{SD}$ setting in the previous sections. As we will be needing these results in various proofs, we summarize them together in the following corollary. We will then simply cite this corollary in our upcoming proofs, instead of repeating the same argument in multiple proofs.

Corollary 8.1. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$ and has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S}\,||_{SD}\,\mathbf{G}$ be ALF. Let TDES $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 2 and 3. Then, the following properties are satisfied: **(1)** $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has a finite state space, **(2)** $\boldsymbol{\mathcal{S}}$ has a finite state space, **(3)** $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}}) = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$, **(4)** $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, **(5)** $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, **(6)** $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$, **(7)** $\boldsymbol{\mathcal{S}}$ is CS deterministic, **(8)** $\boldsymbol{\mathcal{S}}$ is ALF, and **(9)** $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF.

Proof. Assume initial conditions. (1)

1) Show: $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has a finite state space
By (1), we have that $\mathbf{G}$ and $\mathbf{S}$ have finite state spaces. It follows from Definition 4.1 of the $||_{SD}$ operator that $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has a finite state space.

2) Show: $\boldsymbol{\mathcal{S}}$ has a finite state space

By (1), we have that $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is constructed using Algorithms 2 and 3. It thus follows automatically from Point (1) that $\boldsymbol{\mathcal{S}}$ has a finite state space.

3) Show: $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}}) = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$
By (1), we have that $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is constructed using Algorithms 2 and 3. As state space minimization process does not affect the closed and marked languages of an automaton, we conclude that $L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(\boldsymbol{\mathcal{S}}) = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$.

4) Show: $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$
By (1), we have that $\mathbf{G}$ is complete with $||_{SD}$ for $\mathbf{S}$. As plant completeness is a language based property, by Point (3) and Proposition 5.4, we conclude that $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$.

5) Show: $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour
By (1), we have that $\mathbf{G}$ has $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$. By Point (3) and Proposition 5.5, we conclude that $\mathbf{G}$ has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour.

6) Show: $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$
By (1), we have that $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$. By Point (3) and Proposition 5.7, we conclude that $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$.

7) Show: $\boldsymbol{\mathcal{S}}$ is CS deterministic
By (1), we have that $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$ and $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is constructed using Algorithms 2 and 3. Applying Proposition 6.2, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

8) Show: $\boldsymbol{\mathcal{S}}$ is ALF
By Point (1) we have that $\mathbf{S}\,||_{SD}\,\mathbf{G}$ has a finite state space, and by (1) we have that $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is ALF and $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G})$ is constructed using Algorithms 2 and 3. Applying Theorem 6.1, we conclude that $\boldsymbol{\mathcal{S}}$ is ALF.

9) Show: $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF
By Point (8), we have that $\boldsymbol{\mathcal{S}}$ is ALF. As $\boldsymbol{\mathcal{S}}$ and $\mathbf{G}$ are defined over the same Σ, by Proposition 5.8 we conclude that $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF. □

8.2.1 SD Controller as a Supervisory Control

In the $||_{SD}$ setting, the controlled behaviour of closed-loop system, $\mathbf{S}\,||_{SD}\,\mathbf{G}$, is a combination of the control action of TDES supervisor $\mathbf{S}$ and the *tick* disablement mechanism of $||_{SD}$ operator. This means a *tick* event that is possible in TDES plant $\mathbf{G}$ might be enabled by $\mathbf{S}$ too. However, it still might not be possible in $\mathbf{S}\,||_{SD}\,\mathbf{G}$, as our $||_{SD}$ operator is capable of removing *tick* from the closed-loop system in the presence of enabled prohibitable events. As this path is not possible in the theoretical system model, we want to make sure that our SD controller forbids such strings from occurring in the implemented system as well, thus preventing the physical system to behave in an undesirable and unexpected way.

We show this in our next proposition by providing sufficient conditions and proving that if a concurrent string is not possible in our theoretical closed-loop system $\mathbf{S}\,||_{SD}\,\mathbf{G}$, then SD controller $\mathbf{C}$ will not allow it to occur in the physical implementation. By proving this result, we essentially guarantee that the physical system under the control action of SD controller $\mathbf{C}$ does not violate the behaviour, constraints and control laws specified by our theoretical $||_{SD}$ system.

It is important to point out that in the following proposition, we are not comparing the control action of $\mathbf{C}$ with $\mathbf{S}$ only. This is because in the $||_{SD}$ setting, designers are not required

to manually incorporate all of the logic of explicit *tick* disablement in the supervisor model, as they have the option of leaving it up to the $||_{SD}$ operator to automatically perform this task for them while constructing the closed-loop system. This implies that the individual control action of $\mathbf{S}$ might not always match with $\mathbf{C}$, which is neither required nor expected in the presence of $||_{SD}$ operator. Therefore, our goal is to ensure that the control action of $\mathbf{C}$ always remains exactly in line with the controlled behaviour of $\mathbf{S}\,||_{SD}\,\mathbf{G}$, and not only $\mathbf{S}$, which is what we are proving in the proposition given below.

Proposition 8.3. For TDES plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$, let TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$. Let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$ and have $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$.
$(\forall s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp})\,(\forall s' \in CB_{\mathbf{G}}(s))$
If s takes $\mathbf{C}$ to state $\mathbf{q}$ and $ss' \notin L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, then $\mathbf{C}$ will reject s'.

Proof. Assume initial conditions. (1)

Let $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ and $s' \in CB_{\mathbf{G}}(s)$. (2)

Assume: s takes $\mathbf{C}$ to state $\mathbf{q}$ and $ss' \notin L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ (3)

We will now show this implies $\mathbf{C}$ will reject s'.

We will use Proposition 3.1 of the SD setting to show our desired result. To do this, we first need to setup things for the SD setting, and show that the preconditions are satisfied.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 2 and 3. (4)
$\Rightarrow L(\boldsymbol{\mathcal{S}}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ *by Corollary* 8.1 (5)

We note that except for the CS deterministic property, the remaining conditions needed to apply Proposition 3.1 are all language based. Thus, if they apply to $\mathbf{S}\,||_{SD}\,\mathbf{G}$, they also apply to $\boldsymbol{\mathcal{S}}$.

By Corollary 5.1(v), we have: $L(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$
$\Rightarrow s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp}$ *by* (2) (6)

By (1), we have that $\mathbf{S}$ is SD controllable with $||_{SD}$ for $\mathbf{G}$. By (4) and Corollary 8.1, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic. (7)

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$.

Let Λ and Λ' be the state mapping functions (Definition 3.15) for $\mathbf{C}$ and $\boldsymbol{\mathcal{C}}$ respectively.

As $\boldsymbol{\mathcal{S}}$ is CS deterministic by (7) and $s \in L(\boldsymbol{\mathcal{S}}) \cap L_{samp}$ by (6), by Proposition 3.2 we conclude that s will take $\boldsymbol{\mathcal{C}}$ to state $\mathbf{q}' = \Lambda'(\xi'(x'_o, s))$. (8)

As $s \in L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_{samp}$ by (2), by Proposition 5.1 we conclude that $s \in L(\mathbf{S}) \cap L_{samp}$.

We can thus apply Proposition 3.2 and conclude $\mathbf{q} = \Lambda(\xi(x_o, s))$. (9)

As $ss' \notin L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ by (3), this implies $ss' \notin L(\boldsymbol{\mathcal{S}})$ by (5). (10)

We now have $s \in L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G}) \cap L_{samp}$ by (6), $s' \in CB_{\mathbf{G}}(s)$ by (2), $\boldsymbol{\mathcal{S}}$ is CS deterministic by (7), s takes $\boldsymbol{\mathcal{C}}$ to state $\mathbf{q}'$ by (8), and $ss' \notin L(\boldsymbol{\mathcal{S}})$ by (10). Also, by (1), (5) and Corollary 8.1 we have that $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$ and has $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, and $\boldsymbol{\mathcal{S}}$ is SD controllable for $\mathbf{G}$.

We can now apply Proposition 3.1 and conclude that $\boldsymbol{\mathcal{C}}$ will reject s' at state $\mathbf{q}'$. (11)

As all assumptions of Proposition 7.3 are satisfied, we thus conclude:
$\Phi(\Lambda(\xi(x_o, s))) = \Phi'(\Lambda'(\xi'(x'_o, s)))$

$\Rightarrow \Phi(\mathbf{q}) = \Phi'(\mathbf{q}')$ *by* (8) *and* (9)

As $\mathbf{q}$ and $\mathbf{q}'$ have the same output, it follows that if $\boldsymbol{\mathcal{C}}$ rejects s' at state $\mathbf{q}'$ (by (11)), then $\mathbf{C}$ will also reject s' at state $\mathbf{q}$, as required. □

8.2.2 SD Controller and Controllability

In general, a TDES supervisor is more expressive than an SD controller in terms of updating its enablement and forcing information. This is because a supervisor can change this information every time an event occurs. On the other hand, an SD controller is restricted to update its enablement and forcing actions only after a *tick* event, and then it must keep this information constant until the occurrence of the next *tick*.

For our $||_{SD}$ setting, we are interested in showing that despite these differences between the supervisor and the SD controller, the closed-loop behaviour of TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ is exactly the same as the closed-loop behaviour of $\mathbf{G}$ and SD controller $\mathbf{C}$. Please note that the *closed-loop behaviour* of $\mathbf{G}$ and $\mathbf{C}$ is represented as $L(V/\mathbf{G})$.

This notion is proved in our next theorem. We base our result on Theorem 3.1 of the SD setting. Our result is useful as it demonstrates that when we implement our supervisor $\mathbf{S}$ as an SD controller $\mathbf{C}$, we are guaranteed to get the same expected closed-loop behaviour in the physical implementation as our theoretical $||_{SD}$ system, at least with respect to the required enablement and forcing actions of the controller.

Theorem 8.1. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S}\,||_{SD}\,\mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and let V be the map constructed from $\mathbf{C}$ using Algorithm 4. Then:

$$L(V/\mathbf{G}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$$

Proof. Assume initial conditions. (1)

Must show: $L(V/\mathbf{G}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$

In order to use Theorem 3.1 of the SD setting to conclude our desired result, we first need to setup things and show that its preconditions are satisfied.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 2 and 3. (2)

By (1), (2) and Corollary 8.1, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and let $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 1.

Let $L(V/\mathbf{G})$ be the closed behaviour of $V/\mathbf{G}$, and let $L(\mathcal{V}/\mathbf{G})$ be the closed behaviour of $\mathcal{V}/\mathbf{G}$.

Now we will show that $L(V/\mathbf{G}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$.

We first apply Proposition 8.2 and conclude: $V = \mathcal{V}$

By Definition 2.24 of $L(V/\mathbf{G})$ and $L(\mathcal{V}/\mathbf{G})$, this implies: $L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$

By Corollary 5.1(v), we have: $L(\mathbf{S}\,||_{SD}\,\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

Thus, to show that $L(V/\mathbf{G}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$, it is sufficient to show: $L(\mathcal{V}/\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$

By (1), (2), and Corollary 8.1, we have that $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ have finite state spaces, $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, $\mathbf{G}$ has proper time and $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, $\boldsymbol{\mathcal{S}}$ is CS deterministic and SD controllable for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF.

We can now apply Theorem 3.1 and conclude $L(\mathcal{V}/\mathbf{G}) = L(\boldsymbol{\mathcal{S}}) \cap L(\mathbf{G})$, as required. □

Our next proposition shows that map V constructed from SD controller $\mathbf{C}$ using Algorithm 4 is indeed a TDES supervisory control for TDES plant $\mathbf{G}$. We will base our result on Proposition 3.3 of the SD setting which shows similar result for map $\mathcal{V}$ that is constructed from SD controller $\boldsymbol{\mathcal{C}}$ using Algorithm 1.

Proposition 8.4. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S} \,||_{SD}\, \mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and let V be the map constructed from $\mathbf{C}$ using Algorithm 4. Then, map V is a TDES supervisory control for $\mathbf{G}$.

Proof. Assume initial conditions. (1)

In order to apply Proposition 3.3 of the SD setting, we first need to setup things and show that its preconditions are satisfied.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S} ||_{SD} \mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 2 and 3. (2)

By (1), (2) and Corollary 8.1, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and let $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 1.

By (1), (2) and Corollary 8.1, we have that $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ have finite state spaces, $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, $\mathbf{G}$ has proper time and $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, $\boldsymbol{\mathcal{S}}$ is CS deterministic and SD controllable for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}} \,||\, \mathbf{G}$ is ALF.

We can now apply Proposition 3.3 and conclude that map $\mathcal{V}$ is a TDES supervisory control for $\mathbf{G}$. (3)

We next apply Proposition 8.2 and conclude: $V = \mathcal{V}$

As $\mathcal{V}$ is a TDES supervisory control for $\mathbf{G}$ by (3) and $V = \mathcal{V}$, it follows immediately that V is also a TDES supervisory control for $\mathbf{G}$. □

8.2.3 SD Controller and Event Generation

In a typical system, prohibitable events are often part of a supervisor's implementation and they completely depend on the supervisor's discretion for their occurrence. In our $||_{SD}$ setting, this means that the resulting SD controller could potentially make these prohibitable events to occur whenever it wants, possibly even when the plant model does not want them to happen. The occurrence of a prohibitable event might correspond to setting an output of the controller to true, executing a software routine, or sending a message.

In the following theorem, we provide sufficient conditions to make sure that the aforementioned undesirable situation does not occur in our $||_{SD}$ setting. Specifically, we formally prove that if the stated conditions are met, then the SD controller $\mathbf{C}$, translated from TDES supervisor $\mathbf{S}$, cannot generate a prohibitable event when TDES plant $\mathbf{G}$ won't accept it.

This result is beneficial as it forbids the occurrence of *illegal transitions* and prevents the implemented system from violating control laws. It also means that plant model will accurately reflect the $||_{SD}$ system's behaviour when controlled by the SD controller $\mathbf{C}$.

Theorem 8.2. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi,$

$x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S}\,||_{SD}\,\mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and let V be the map constructed from $\mathbf{C}$ using Algorithm 4.
$(\forall s \in L(V/\mathbf{G}) \cap L_{samp})\,(\forall s' \in \Sigma^*_{act})\,(\forall \sigma \in \Sigma_{hib})$
If $ss' \in L(V/\mathbf{G})$ and σ then physically occurs after ss' and before any other events can occur, then $ss'\sigma \in L(\mathbf{G})$.

Proof. Assume initial conditions. (1)

Let $s \in L(V/\mathbf{G}) \cap L_{samp}$, $s' \in \Sigma^*_{act}$, and $\sigma \in \Sigma_{hib}$. (2)

Assume: $ss' \in L(V/\mathbf{G})$ and that σ physically occurs after ss' and before any other events can occur (3)

Must show: $ss'\sigma \in L(\mathbf{G})$

We will use Theorem 3.2 of the SD setting to show our desired result. To do this, we first need to establish the preconditions of Theorem 3.2 and then the result will follow.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 2 and 3. (4)

By (1), (4) and Corollary 8.1, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and let $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 1.

Let $L(\mathcal{V}/\mathbf{G})$ be the closed-loop behaviour of $\mathcal{V}/\mathbf{G}$.

We can first apply Proposition 8.2 and conclude: $V = \mathcal{V}$
$\Rightarrow L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$
$\Rightarrow s \in L(\mathcal{V}/\mathbf{G}) \cap L_{samp}, s' \in \Sigma^*_{act}$, and $\sigma \in \Sigma_{hib}$ *by* (2)

We also have that $ss' \in L(\mathcal{V}/\mathbf{G})$ and that σ physically occurs after ss' and before any other events can occur by (3).

By (1), (4) and Corollary 8.1, we have that $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ have finite state spaces, $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, $\mathbf{G}$ has proper time and $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, $\boldsymbol{\mathcal{S}}$ is CS deterministic and SD controllable for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF.

We can now apply Theorem 3.2 and conclude $ss' \in L(\mathbf{G})$. □

8.2.4 SD Controller and Nonblocking

One of the fundamental properties that a TDES is required to satisfy is nonblocking. In the $||_{SD}$ setting, we wish to guarantee that if our theoretical $||_{SD}$ system is nonblocking, then the physical system implemented under the control action of SD controller will retain this property. This is the main focus of our next proof.

The following proposition proves that if specified conditions are satisfied in the $||_{SD}$ setting, then the closed-loop behaviour of TDES plant $\mathbf{G}$ and SD controller $\mathbf{C}$ is nonblocking if and only if the closed-loop behaviour of $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ is nonblocking.

Proposition 8.5. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S}\,||_{SD}\,\mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the

SD controller constructed from $\mathbf{S}$, and let V be the map constructed from $\mathbf{C}$ using Algorithm 4. Then V is nonblocking for $\mathbf{G}$ if and only if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is nonblocking.

Proof. Assume initial conditions. (1)

Must show: V is nonblocking for $\mathbf{G}$ if and only if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is nonblocking

To show this, it is sufficient to show: $L(V/\mathbf{G}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ and $L_m(V/\mathbf{G})_{||_{SD}} = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$

Applying Theorem 8.1 (by (1)), we conclude: $L(V/\mathbf{G}) = L(\mathbf{S}\,||_{SD}\,\mathbf{G})$ (2)

Now all that remains is to show: $L_m(V/\mathbf{G})_{||_{SD}} = L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})$

By Definition 8.1 of $L_m(V/\mathbf{G})_{||_{SD}}$, we have:

$$\begin{aligned} L_m(V/\mathbf{G})_{||_{SD}} &= L(V/\mathbf{G}) \cap L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \\ &= L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \cap L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \quad by\ (2) \\ &= L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \quad as\ L_m(\mathbf{S}\,||_{SD}\,\mathbf{G}) \subseteq L(\mathbf{S}\,||_{SD}\,\mathbf{G}) \end{aligned}$$

We thus conclude that V is nonblocking for $\mathbf{G}$ if and only if $\mathbf{S}\,||_{SD}\,\mathbf{G}$ is nonblocking. □

In the SD supervisory control theory, the SD setting is proven to be robust with respect to multiple variations of concurrent strings and nonblocking. Specifically, if theoretical TDES system is nonblocking, then TDES plant $\mathbf{G}$ under the control of SD controller $\boldsymbol{\mathcal{C}}$ is shown to be nonblocking, even if multiple concurrent strings with the same occurrence image are possible at a given sampled state in the theoretical SD system and only one of these concurrent strings is actually possible in the physical implementation.

We also wish to demonstrate such robustness with respect to nonblocking for our $||_{SD}$ setting. In order to be able to do that, first we present a supporting proposition that will help us in proving our main result. In the following proposition, we show that for any V' that is a TDES supervisory control for TDES plant $\mathbf{G}$, V' is concurrent supervisory control equivalent (CSCE: Definition 3.24) to TDES supervisory control V of the $||_{SD}$ setting if and only if V' is CSCE to TDES supervisory control $\mathcal{V}$ of the SD setting.

Proposition 8.6. Let TDES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ be a plant to be controlled. Let TDES $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ be a CS deterministic supervisor that is SD controllable with $||_{SD}$ for $\mathbf{G}$, and let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$. Let TDES supervisor $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ constructed using Algorithms 2 and 3 be CS deterministic. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and V be the map constructed from $\mathbf{C}$ using Algorithm 4. Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 1. Then, for all V' that are TDES supervisory controls for $\mathbf{G}$, V' is concurrent supervisory control equivalent to V if and only if V' is concurrent supervisory control equivalent to $\mathcal{V}$.

Proof. Assume initial conditions. (1)

Let V' be a TDES supervisory control for $\mathbf{G}$.

Must show: V' is concurrent supervisory control equivalent (CSCE) to V if and only if V' is CSCE to $\mathcal{V}$

Let $L(V/\mathbf{G})$, $L(\mathcal{V}/\mathbf{G})$ and $L(V'/\mathbf{G})$ be the closed behaviours of $V/\mathbf{G}$, $\mathcal{V}/\mathbf{G}$ and $V'/\mathbf{G}$ respectively.

We can now apply Proposition 8.2 (by (1)) and conclude: $V = \mathcal{V}$ (2)

$\Rightarrow L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$ (3)

We next note that by Definition 3.24, if V' is CSCE to V, this implies:

1) $(\forall s \in L(\mathbf{G}))\, V'(s) \subseteq V(s)$

2) $(\forall s \in L(V'/\mathbf{G}) \cap L_{samp})\, (\forall s' \in L_{conc})\, ss' \in L(V/\mathbf{G}) \Rightarrow$
$(\exists s'' \in L_{conc})\, ss'' \in L(V'/\mathbf{G}) \wedge Occu(s') = Occu(s'')$

However, as $V = \mathcal{V}$ by (2), and $L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$ by (3), we can substitute into the above and get:

1) $(\forall s \in L(\mathbf{G}))\, V'(s) \subseteq \mathcal{V}(s)$

2) $(\forall s \in L(V'/\mathbf{G}) \cap L_{samp})\, (\forall s' \in L_{conc})\, ss' \in L(\mathcal{V}/\mathbf{G}) \Rightarrow$
$(\exists s'' \in L_{conc})\, ss'' \in L(V'/\mathbf{G}) \wedge Occu(s') = Occu(s'')$

This implies that V' is CSCE to $\mathcal{V}$. □

We will now present our final result that proves the robustness of the $||_{SD}$ setting with respect to different variations of concurrent strings and nonblocking. Our next theorem shows that if the theoretical $||_{SD}$ system satisfies the stated conditions and the closed-loop behaviour of TDES plant $\mathbf{G}$ and SD controller $\mathbf{C}$ is nonblocking, then any of its CSCE variations will also be nonblocking. This theorem makes use of Theorem 3.3 of the SD setting to conclude the desired result.

This result is beneficial as it provides liberty to practitioners to choose any specific implementation of $\mathbf{S}\,||_{SD}\,\mathbf{G}$ without having to worry about potential blocking of the physical system. If they fulfill the specified conditions, then they are guaranteed that the physical system under the action of the SD controller $\mathbf{C}$ will be nonblocking, even if their chosen implementation only allows a subset of variations of a concurrent string out of all variations possible in $\mathbf{S}\,||_{SD}\,\mathbf{G}$.

Theorem 8.3. For plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and CS deterministic supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ that is SD controllable with $||_{SD}$ for $\mathbf{G}$, let both TDES have finite state spaces, let $\mathbf{G}$ be complete with $||_{SD}$ for $\mathbf{S}$, have proper time behaviour and $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$, and let $\mathbf{S}\,||_{SD}\,\mathbf{G}$ be ALF. Let $\mathbf{C} = (I, Z, Q, \Omega, \Phi, \mathbf{q}_{res})$ be the SD controller constructed from $\mathbf{S}$, and let V be the map constructed from $\mathbf{C}$ using Algorithm 4. Then, for all V' that are TDES supervisory controls for $\mathbf{G}$, if V is nonblocking for $\mathbf{G}$ and V' is concurrent supervisory control equivalent to V, then V' is also nonblocking for $\mathbf{G}$.

Proof. Assume initial conditions. (1)

Let V' be a TDES supervisory control for $\mathbf{G}$.

Assume: V' is concurrent supervisory control equivalent (CSCE) to V and that V is nonblocking for $\mathbf{G}$. By Definition 8.2, this implies: $\overline{L_m(V/\mathbf{G})_{||_{SD}}} = L(V/\mathbf{G})$ (2)

Must show: V' is nonblocking for $\mathbf{G}$

Let $L(V'/\mathbf{G})$ and $L_m(V'/\mathbf{G})$ be the closed and marked behaviour of $V'/\mathbf{G}$.

Sufficient to show: $\overline{L_m(V'/\mathbf{G})_{||_{SD}}} = L(V'/\mathbf{G})$

We first define our setting and notation for the proof.

Let $\boldsymbol{\mathcal{S}} = min(\mathbf{S}\,||_{SD}\,\mathbf{G}) = (X', \Sigma, \xi', x'_o, X'_m)$ be constructed using Algorithms 2 and 3. (3)

By (1), (3) and Corollary 8.1, we conclude that $\boldsymbol{\mathcal{S}}$ is CS deterministic.

Let $\boldsymbol{\mathcal{C}} = (I', Z', Q', \Omega', \Phi', \mathbf{q}'_{res})$ be the SD controller constructed from $\boldsymbol{\mathcal{S}}$, and let $\mathcal{V}$ be the map constructed from $\boldsymbol{\mathcal{C}}$ using Algorithm 1.

Let $L(V/\mathbf{G})$ and $L(\mathcal{V}/\mathbf{G})$ be the closed behaviours of $V/\mathbf{G}$ and $\mathcal{V}/\mathbf{G}$ respectively.

We now apply Proposition 8.2 and conclude: $V = \mathcal{V}$

$\Rightarrow L(V/\mathbf{G}) = L(\mathcal{V}/\mathbf{G})$ (4)

Now we will show that $\overline{L_m(V'/\mathbf{G})_{||_{SD}}} = L(V'/\mathbf{G})$.

By Definition 8.1 of $L_m(V'/\mathbf{G})_{||_{SD}}$, it is sufficient to show:

$\overline{L(V'/\mathbf{G}) \cap L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})} = L(V'/\mathbf{G})$

$\Rightarrow \overline{L(V'/\mathbf{G}) \cap L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})} = L(V'/\mathbf{G})$ *by Corollary* 5.1(vi)

$\Rightarrow \overline{L_m(V'/\mathbf{G})} = L(V'/\mathbf{G})$ *by Definition* 3.22 *of* $L_m(V'/\mathbf{G})$

This means, in order to show that V' is nonblocking for $\mathbf{G}$ in our $||_{SD}$ setting, it is sufficient to show that V' is nonblocking for $\mathbf{G}$ in the SD setting (Definition 3.23).

We will show this by using Theorem 3.3 of the SD setting. We will first establish the preconditions of Theorem 3.3 and then the result will follow.

By (2), we have: $\overline{L_m(V/\mathbf{G})_{||_{SD}}} = L(V/\mathbf{G})$

$\Rightarrow \overline{L(V/\mathbf{G}) \cap L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})} = L(V/\mathbf{G})$ *by Definition* 8.1 *of* $L_m(V/\mathbf{G})_{||_{SD}}$

$\Rightarrow \overline{L(\mathcal{V}/\mathbf{G}) \cap L_m(\mathbf{S}\,||_{SD}\,\mathbf{G})} = L(\mathcal{V}/\mathbf{G})$ *by* (4)

$\Rightarrow \overline{L(\mathcal{V}/\mathbf{G}) \cap L_m(\boldsymbol{\mathcal{S}}) \cap L_m(\mathbf{G})} = L(\mathcal{V}/\mathbf{G})$ *by Corollary* 5.1(vi)

$\Rightarrow \overline{L_m(\mathcal{V}/\mathbf{G})} = L(\mathcal{V}/\mathbf{G})$ *by Definition* 3.22 *of* $L_m(\mathcal{V}/\mathbf{G})$

By Definition 3.23, this indicates that $\mathcal{V}$ is nonblocking for $\mathbf{G}$.

Applying Proposition 8.6, we note that as V' is CSCE to V by (2), this implies that V' is CSCE to $\mathcal{V}$.

By (1), (3) and Corollary 8.1, we have that $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ have finite state spaces, $\mathbf{G}$ is complete for $\boldsymbol{\mathcal{S}}$, $\mathbf{G}$ has proper time and $\boldsymbol{\mathcal{S}}$-singular prohibitable behaviour, $\boldsymbol{\mathcal{S}}$ is CS deterministic and SD controllable for $\mathbf{G}$, and $\boldsymbol{\mathcal{S}}\,||\,\mathbf{G}$ is ALF.

We now apply Theorem 3.3 and conclude that V' is nonblocking for $\mathbf{G}$ in the SD setting.

By showing that V' is nonblocking for $\mathbf{G}$ in the SD setting, we have thus shown that V' is nonblocking for $\mathbf{G}$ in our $||_{SD}$ setting, as required. □

9 Symbolic Verification in SD Synchronous Product Setting

In this section, we discuss theoretical concepts and predicate-based algorithms to symbolically verify various properties in our $||_{SD}$ setting. This section is based on symbolic verification of the SD supervisory control methodology presented in [42], who in turn built upon the symbolic computation and verification work done by [36] and [31].

We begin this section by introducing the fundamental concepts of predicates and predicate transformers. This is followed by a discussion on how to use logic formulas to represent state subsets and transitions in our $||_{SD}$ setting. After that, we describe the symbolic computation of transitions, inverse transitions and predicate transformers. Finally, we present algorithms that can be used to verify various properties in our $||_{SD}$ setting. All data representations, computations and verifications discussed in this section are based on ordered binary decision diagrams (BDD) [9, 10].

Please note that the algorithms discussed in this section were originally developed as part of the SD supervisory control methodology by [42]. We have tweaked them to match our adapted properties of the $||_{SD}$ setting introduced in Section 4. Since some properties of our $||_{SD}$ setting are logically similar to the SD setting, their corresponding algorithm steps remain unchanged. These unmodified algorithms are included in Appendix B for the sake of completeness.

Note: In this section, we will represent *logical equivalence* between state predicates by '$\equiv$', *logical true* by 'T' and *logical false* by 'F' respectively. Also, we will use $\boldsymbol{\mathcal{S}}$ to refer to the supervisor of the SD setting (Section 3).

9.1 Predicates and Predicate Transformers

This section introduces the concepts of state predicates and predicate transformers from [36].

9.1.1 State Predicates

For the following definitions, let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$.

Definition 9.1. A *predicate* P defined on state set Q is a function $P : Q \to \{T, F\}$ *identified* by the corresponding state subset $Q_P := \{q \in Q \mid P(q) = T\} \subseteq Q$.

If $q \in Q_P$, then $q \models P$ means "q *satisfies* P" or "P *includes* q". Thus, we have $q \models P \iff P(q) = T$.

Definition 9.2. A predicate defined on the state set of a TDES is referred to as a *state predicate*. The state predicate *true* is identified by Q, state predicate *false* by $\emptyset$, and state predicate P_m by Q_m.

We write $Pred(Q)$ to represent the set of all predicates defined on Q. Thus, $Pred(Q)$ is identified by $\mathrm{Pwr}(Q)$. For $P \in Pred(Q)$, $st(P)$ denotes the corresponding state subset $Q_P \subseteq Q$ which identifies P. We use $pr(Q)$ to represent the predicate that is identified by Q.

For $q \in Q$ and $P, P_1, P_2 \in Pred(Q)$, the following predicate operations can be used to build various boolean expressions:

- $(\neg P)(q) = T \iff P(q) = F$
- $(P_1 \wedge P_2)(q) = T \iff P_1(q) = T$ and $P_2(q) = T$
- $(P_1 \vee P_2)(q) = T \iff P_1(q) = T$ or $P_2(q) = T$
- $(P_1 - P_2)(q) = T \iff P_1(q) = T$ and $P_2(q) = F$

Definition 9.3. The partial order relation $\preceq$ over $Pred(Q)$ is defined as:

$$(\forall P_1, P_2 \in Pred(Q))\, P_1 \preceq P_2 \iff (P_1 \wedge P_2) \equiv P_1$$

It is obvious that $Q_{P_1} \subseteq Q_{P_2} \iff P_1 \preceq P_2$. Thus, we have $(\forall q \in Q)\, q \models P_1 \implies q \models P_2$.

Definition 9.4. For some state set Q, let $P_1, P_2 \in Pred(Q)$. P_1 is a *subpredicate* of P_2 if $P_1 \preceq P_2$. We say P_1 is *stronger* than P_2, and P_2 is *weaker* than P_1.

$Sub(P)$ represents the set of all subpredicates of $P \in Pred(Q)$ such that $Sub(P)$ is identified by $\mathrm{Pwr}(Q_P)$.

9.1.2 Predicate Transformers

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ and $P \in Pred(Q)$. A *predicate transformer* is defined as a function $f : Pred(Q) \to Pred(Q)$. In our subsequent sections, we will use the following

predicate transformers from [36].

i) $R(\mathbf{G}, P)$

The *reachability predicate* $R(\mathbf{G}, P)$ holds true for those states in $\mathbf{G}$ that can be reached from q_o by states satisfying P. It is inductively defined as follows:

1. $q_o \models P \implies q_o \models R(\mathbf{G}, P)$.

2. $q \models R(\mathbf{G}, P)\ \&\ \sigma \in \Sigma\ \&\ \delta(q, \sigma)!\ \&\ \delta(q, \sigma) \models P \implies \delta(q, \sigma) \models R(\mathbf{G}, P)$.

3. No other states satisfy $R(\mathbf{G}, P)$.

In simple words, a state $q \models R(\mathbf{G}, P)$ if and only if there exists a path from q_o to q in $\mathbf{G}$ and each state in that path satisfies P. $R(\mathbf{G}, true)$ represents the set of all reachable states in Q.

ii) $CR(\mathbf{G}, P)$

The *coreachability predicate* $CR(\mathbf{G}, P)$ holds true for those states in $\mathbf{G}$ that can reach a marked state by states satisfying P. It is inductively defined as follows:

1. $P_m \wedge P \equiv false \implies CR(\mathbf{G}, P) \equiv false$.

2. $q \models P_m \wedge P \implies q \models CR(\mathbf{G}, P)$.

3. $q \models CR(\mathbf{G}, P)\ \&\ q' \models P\ \&\ \sigma \in \Sigma\ \&\ \delta(q', \sigma)!\ \&\ \delta(q', \sigma) = q \implies q' \models CR(\mathbf{G}, P)$.

4. No other states satisfy $CR(\mathbf{G}, P)$.

In other words, a state $q \models CR(\mathbf{G}, P)$ if and only if there exists a path from q to some marked state in $\mathbf{G}$ and each state in that path satisfies P. $CR(\mathbf{G}, true)$ represents the set of all coreachable states in Q.

iii) $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$

Let $P' \in Pred(Q)$ and $\Sigma' \subseteq \Sigma$. Once $\mathbf{G}$, P' and Σ' are fixed, $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ becomes a predicate transformer. The predicate $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ holds true for those states in $\mathbf{G}$ that can reach a state satisfying P' by states satisfying P and transitions with events in Σ'. It is inductively defined as follows:

1. $P' \wedge P \equiv false \implies \mathcal{CR}(\mathbf{G}, P', \Sigma', P) \equiv false$.

2. $q \models P' \wedge P \implies q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$.

3. $q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)\ \&\ q' \models P\ \&\ \sigma \in \Sigma'\ \&\ \delta(q', \sigma)!\ \&\ \delta(q', \sigma) = q \implies q' \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$.

4. No other states satisfy $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$.

This means that a state $q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ if and only if there exists a path from q to a state satisfying P' in $\mathbf{G}$ and each state in that path satisfies P and each transition event σ is in Σ'.

By comparing the definitions of $\mathcal{CR}$ and CR, we note that $\mathcal{CR}(\mathbf{G}, P_m, \Sigma, P) \equiv CR(\mathbf{G}, P)$.

9.2 Symbolic Representation

In this section, we present the symbolic representation for states and transitions in our $||_{SD}$ setting. Specifically, we discuss how to use logic formulas to represent state subsets and transitions for our $||_{SD}$ system. We have based our work on the symbolic representation of the SD setting given in [42], who in turn borrowed it from [36].

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD} \ldots ||_{SD}\, \mathbf{G}_n$ be constructed by synchronizing component TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ where $i = 1, 2, \ldots, n$, using the SD synchronous product operator. For any state $q \in Q$, by the definition of Q in the $||_{SD}$ operator (Definition 4.1), we have $q = (q_1, q_2, \ldots, q_n)$, where $q_i \in Q_i$.

It is worth pointing out that TDES $\mathbf{G}$ might contain some unreachable states. However, checking for unreachable states while verifying different properties of the TDES is expensive, and does not seem to provide any benefit as these unreachable states do not contribute towards the closed and marked behaviour of $\mathbf{G}$, i.e. $L(\mathbf{G})$ and $L_m(\mathbf{G})$. As a result, the property is first checked (possibly including unreachable states) and then a reachability check is performed over the entire system, and any unreachable states are excluded from the results. This also allows us to do one reachability check, and share the results across several algorithms.

9.2.1 State Subsets

Definition 9.5. For TDES $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD} \ldots ||_{SD}\, \mathbf{G}_n$, let $i = 1, 2, \ldots, n$ and $q_i \in Q_i$. The *state variable* v_i for the i^{th} component TDES $\mathbf{G}_i$ is a variable of domain Q_i. If v_i is assigned the value q_i, then $v_i = q_i$ returns T, otherwise it returns F.

Please note that '=' has been used to test if v_i has been assigned the value q_i.

Definition 9.6. For TDES $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD} \ldots ||_{SD}\, \mathbf{G}_n$, the *state variable vector* $\mathbf{v}$ is a vector $[v_1, v_2, \ldots, v_n]$ of state variables v_i from each component TDES $\mathbf{G}_i$, where $i = 1, 2, \ldots, n$. For state subset $A \subseteq Q$, the predicate P_A for A can be written as:

$$P_A(\mathbf{v}) := \bigvee_{q \in A} (v_1 = q_1 \wedge v_2 = q_2 \wedge \ldots \wedge v_n = q_n)$$

For convenience, instead of $P_A(\mathbf{v})$, we will simply write P_A if $\mathbf{v}$ is understood.

9.2.2 Transitions

Definition 9.7. For TDES $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD} \ldots ||_{SD}\, \mathbf{G}_n$, let $\Sigma_{hib} \subset \Sigma$ and $\sigma \in \Sigma$. A *transition predicate* $N_\sigma : Q \times Q \to \{T, F\}$ is a boolean function that identifies all the transitions for σ in $\mathbf{G}$ and is defined as follows:

$$(\forall q, q' \in Q)\, N_\sigma(q, q') := \begin{cases} T & \text{if } \delta(q,\sigma)! \;\&\; \delta(q,\sigma) = q' \;\&\; \big((\sigma \neq \tau) \text{ OR} \\ & \quad ((\sigma = \tau) \;\&\; (\forall \sigma' \in \Sigma_{hib})\, \neg\, \delta(q, \sigma')!)\big) \\ F & \text{otherwise} \end{cases}$$

For each TDES, two different sets of state variables are needed to distinguish between source and destination states of transitions. These state variables and their corresponding vectors are defined below.

Definition 9.8. For TDES $\mathbf{G} = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD} \ldots ||_{SD}\, \mathbf{G}_n$, let $i = 1, 2, \ldots, n$. For each component TDES $\mathbf{G}_i$, we have the *normal state variable* v_i (source state) and the *prime state variable* v_i' (destination state), both with domain Q_i. For $\mathbf{G}$, we have the *normal state variable vector* $\mathbf{v} = [v_1, v_2, \ldots, v_n]$ and the *prime state variable vector* $\mathbf{v}' = [v_1', v_2', \ldots, v_n']$.

For each $\sigma \in \Sigma$, the transition predicate for σ, N_σ, can be written as follows:

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \begin{cases} \bigwedge\limits_{\{1 \leq i \leq n\}} \left(\bigvee\limits_{\{q_i, q_i' \in Q_i \mid \delta_i(q_i, \sigma) = q_i'\}} (v_i = q_i) \wedge (v_i' = q_i') \right) & \text{if } \mathbf{X} \\ F & \text{otherwise} \end{cases}$$

$$\text{where } \mathbf{X} = (\sigma \neq \tau) \text{ OR } \left(\sigma = \tau \,\&\, (\forall \sigma' \in \Sigma_{hib}) \neg \left(\bigwedge_{\{1 \leq i \leq n\}} \left(\bigvee_{\{q_i \in Q_i | \delta_i(q_i, \sigma')!\}} (v_i = q_i) \right) \right) \right)$$

Essentially, it says that for each $\sigma \in \Sigma - \{\tau\}$, if we set $\mathbf{v} = q$ and $\mathbf{v}' = q'$ such that $\delta(q, \sigma) = q'$, then $N_\sigma(\mathbf{v}, \mathbf{v}')$ will return T. However, for $\sigma = \tau$, $N_\sigma(\mathbf{v}, \mathbf{v}')$ will return T only if we set $\mathbf{v} = q$ and $\mathbf{v}' = q'$ such that $\delta(q, \sigma) = q'$ and for all events $\sigma' \in \Sigma_{hib}$, $\neg\, \delta(q, \sigma')!$.

When a TDES is designed as several smaller component TDES, designers often model these components over different event sets. In order to use the above-mentioned formula for N_σ, selfloops need to be added at every state of the component TDES for events that are missing from their event sets. This makes the transition predicate a lot more complicated and cluttered. In order to resolve this issue, the following version of N_σ has been defined.

Definition 9.9. To represent the transition for a given $\sigma \in \Sigma$, we use the *transition tuple* $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ such that $\mathbf{v}_\sigma := \{v_i \in \mathbf{v} \,|\, \sigma \in \Sigma_i\}, \mathbf{v}'_\sigma := \{v'_i \in \mathbf{v}' \,|\, \sigma \in \Sigma_i\}$ and N_σ is defined as:

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \begin{cases} \bigwedge\limits_{\{1 \leq i \leq n | \sigma \in \Sigma_i\}} \left(\bigvee\limits_{\{q_i, q'_i \in Q_i | \delta_i(q_i, \sigma) = q'_i\}} (v_i = q_i) \wedge (v'_i = q'_i) \right) & \text{if } \mathbf{X} \\ F & \text{otherwise} \end{cases}$$

$$\text{where } \mathbf{X} = (\sigma \neq \tau) \,\text{OR}\, \left(\sigma = \tau \,\&\, (\forall \sigma' \in \Sigma_{hib}) \neg \left(\bigwedge_{\{1 \leq i \leq n | \sigma' \in \Sigma_i\}} \left(\bigvee_{\{q_i \in Q_i | \delta_i(q_i, \sigma')!\}} (v_i = q_i) \right) \right) \right)$$

It is noteworthy that although selflooped transitions are not explicitly specified in the above definition, the tuple still expresses the selfloop information. This implies that this definition can be used to create transition tuples for selflooped components as well.

9.3 Symbolic Computation

By using the logic formula representation for state subsets and transitions of our $||_{SD}$ system defined in the previous section, this section discusses the symbolic computation of transitions, inverse transitions and predicate transformers with respect to our $||_{SD}$ setting. We have built our work on the symbolic computation work done by [36], which was used for the SD setting by [42].

9.3.1 Transitions and Inverse Transitions

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 ||_{SD} \mathbf{G}_2 ||_{SD} \ldots ||_{SD} \mathbf{G}_n$ be constructed by synchronizing component TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ where $i = 1, 2, \ldots, n$, using the SD synchronous product operator.

For any state $q \in Q$ and event $\sigma \in \Sigma$, we want to compute the transition $\delta(q, \sigma)$. An efficient way to compute transitions is to compute the predicate of the set of next states from the predicate of the set of current states.

For $P \in Pred(Q)$, we can directly compute the function $\hat{\delta} : Pred(Q) \times \Sigma \to Pred(Q)$ which is defined as follows:

$$(\forall P \in Pred(Q))\,(\forall \sigma \in \Sigma)\,\hat{\delta}(P, \sigma) := pr(\{q' \in Q \,|\, (\exists q \models P)\, \delta(q, \sigma) = q' \wedge \big((\sigma \neq \tau) \vee ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib})\, \neg\, \delta(q, \sigma')!)\big)\})$$

In order to compute the predicate of the set of source states from the predicate of the set of destination states, we define the inverse function $\hat{\delta}^{-1} : Pred(Q) \times \Sigma \rightarrow Pred(Q)$ as follows:

$$(\forall P \in Pred(Q))\,(\forall \sigma \in \Sigma)\,\hat{\delta}^{-1}(P,\sigma) := pr(\{q \in Q \,|\, \delta(q,\sigma) \models P \wedge \big((\sigma \neq \tau) \vee ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib})\neg\delta(q,\sigma')!)\big)\})$$

As BDD [9, 10] does not support first order logic by itself, [36] has used the *existential quantifier elimination method for finite domain* [2] to compute $\hat{\delta}(P,\sigma)$ and $\hat{\delta}^{-1}(P,\sigma)$. We will use the same method to compute our functions $\hat{\delta}$ and $\hat{\delta}^{-1}$ in the $||_{SD}$ setting.

Definition 9.10. For TDES $\mathbf{G} = \mathbf{G}_1 ||_{SD} \mathbf{G}_2 ||_{SD} \ldots ||_{SD} \mathbf{G}_n$, let $\sigma \in \Sigma$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. For $i = 1, 2, \ldots, n$, if $v_i \in \mathbf{v}_\sigma$ and $v'_i \in \mathbf{v}'_\sigma$, then $\exists v_i N_\sigma$ and $\exists v'_i N_\sigma$ are defined as follows:

$$\exists v_i N_\sigma := \bigvee_{q_i \in Q_i} N_\sigma[q_i/v_i] \qquad\qquad \exists v'_i N_\sigma := \bigvee_{q_i \in Q_i} N_\sigma[q_i/v'_i]$$

Here, $N_\sigma[q_i/v_i]$ is the resulting predicate with each term v_i of N_σ substituted by q_i, and $N_\sigma[q_i/v'_i]$ is the resulting predicate with each term v'_i of N_σ substituted by q_i. In simple words, $\exists v_i$ and $\exists v'_i$ eliminate the variables v_i and v'_i respectively from N_σ.

For $\sigma \in \Sigma$, let $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. For $k \in \{1, 2, \ldots, n\}$, let $\mathbf{v}_\sigma = \{\hat{v_1}, \hat{v_2}, .., \hat{v_k}\}$ and $\mathbf{v}'_\sigma = \{\hat{v'_1}, \hat{v'_2}, \ldots, \hat{v'_k}\}$.

For convenience, we write $\exists \mathbf{v}_\sigma N_\sigma$ to represent $\exists \hat{v_1}(\exists \hat{v_2} \ldots (\exists \hat{v_k} N_\sigma) \ldots)$. The resulting logic formula $\exists \mathbf{v}_\sigma N_\sigma$ contains only the prime variables in $\mathbf{v}'_\sigma$. If we substitute all the prime variables by normal variables, denoted as $\exists \mathbf{v}_\sigma N_\sigma[\mathbf{v}'_\sigma \rightarrow \mathbf{v}_\sigma]$, then the resulting predicate represents the set of *destination states* for σ transitions in $\mathbf{G}$. This means that each state in this set has a σ transition entering it, as defined by our N_σ. This variable substitution is required because normal variables are used to express the logic formula of a state subset predicate.

Likewise, for convenience, we write $\exists \mathbf{v}'_\sigma N_\sigma$ to represent $\exists \hat{v'_1}(\exists \hat{v'_2} \ldots (\exists \hat{v'_k} N_\sigma) \ldots)$. The resulting logic formula $\exists \mathbf{v}'_\sigma N_\sigma$ contains only the normal variables in $\mathbf{v}_\sigma$, therefore no variable substitution is required in this case. $\exists \mathbf{v}'_\sigma N_\sigma$ represents the predicate for the set of *source states* for σ transitions in $\mathbf{G}$. This means that each state in this set has a σ transition leaving it, as defined by our N_σ.

From the above description, it is obvious that $\exists \mathbf{v}_\sigma N_\sigma[\mathbf{v}'_\sigma \rightarrow \mathbf{v}_\sigma]$ computes the predicate representing the set of destination states $\{q' \in Q \,|\, (\exists q \in Q)\,\delta(q,\sigma) = q' \wedge ((\sigma \neq \tau) \vee ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib})\,\neg\delta(q,\sigma')!))\}$. Similarly, $\exists \mathbf{v}'_\sigma N_\sigma$ computes the predicate representing the set of source states $\{q \in Q \,|\, \delta(q,\sigma)! \wedge ((\sigma \neq \tau) \vee ((\sigma = \tau) \wedge (\forall \sigma' \in \Sigma_{hib})\,\neg\delta(q,\sigma')!))\}$.

By using the existential quantifier elimination method, we can now compute $\hat{\delta}$ and $\hat{\delta}^{-1}$ symbolically as follows.

Definition 9.11. For TDES $\mathbf{G} = \mathbf{G}_1 ||_{SD} \mathbf{G}_2 ||_{SD} \ldots ||_{SD} \mathbf{G}_n$, let $\sigma \in \Sigma$, $P \in Pred(Q)$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. Then, $\hat{\delta}(P,\sigma)$ is computed as follows:

$$\hat{\delta}(P,\sigma) := (\exists \mathbf{v}_\sigma (N_\sigma \wedge P))\,[\mathbf{v}'_\sigma \rightarrow \mathbf{v}_\sigma]$$

In the above definition, by first computing $N_\sigma \wedge P$, we are restricting σ transitions to only those source states that satisfy P.

Definition 9.12. For TDES $\mathbf{G} = \mathbf{G}_1 ||_{SD} \mathbf{G}_2 ||_{SD} \ldots ||_{SD} \mathbf{G}_n$, let $\sigma \in \Sigma$, $P \in Pred(Q)$ and $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. Then, $\hat{\delta}^{-1}(P,\sigma)$ is computed as follows:

$$\hat{\delta}^{-1}(P,\sigma) := \exists \mathbf{v}'_\sigma (N_\sigma \wedge (P[\mathbf{v}_\sigma \rightarrow \mathbf{v}'_\sigma]))$$

In this definition, $P[\mathbf{v}_\sigma \rightarrow \mathbf{v}'_\sigma]$ returns predicate P with its normal variables substituted

Algorithm 5 $R(\mathbf{G}, P)$

1: $P_1 \leftarrow P \wedge pr(\{q_o\})$
2: **repeat**
3: $\quad P_2 \leftarrow P_1$
4: $\quad$ **for** $i \leftarrow 1$ to n **do**
5: $\quad\quad P_3 \leftarrow false$
6: $\quad\quad$ **repeat**
7: $\quad\quad\quad P_{new} \leftarrow P_1 - P_3$
8: $\quad\quad\quad P_3 \leftarrow P_1$
9: $\quad\quad\quad P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma_i} (\hat{\delta}(P_{new}, \sigma) \wedge P) \right)$
10: $\quad\quad$ **until** $P_1 \equiv P_3$
11: $\quad$ **end for**
12: **until** $P_1 \equiv P_2$
13: **return** P_1

by prime variables. As prime variables represent destination states, this has the effect of restricting σ transitions to only those destination states that satisfy P.

9.3.2 Predicate Transformers

In order to compute the predicate transformers R and $\mathcal{CR}$ defined in Section 9.1.2, Algorithms 5 and 6 have been taken from [36]. Please refer to [36] for a detailed description of these algorithms.

Let TDES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \,||_{SD}\, \mathbf{G}_2 \,||_{SD} \ldots ||_{SD}\, \mathbf{G}_n$ be constructed by synchronizing component TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ where $i = 1, 2, \ldots, n$, using the SD synchronous product operator. Let $P \in Pred(Q)$.

Reachability Check Algorithm 5[11] computes $R(\mathbf{G}, P)$ by taking two parameters as input, a TDES $\mathbf{G}$ and a predicate P. It then computes and returns a predicate P_1 containing the set of states in $\mathbf{G}$ that are reachable by the initial state q_o via states satisfying P.

It is interesting to note that this algorithm has been used in the SD setting by [42], and we will also use the same algorithm in our $||_{SD}$ setting without any modification. Although the steps of the algorithm are the same, it will most likely give different results in the SD and the $||_{SD}$ setting.

In the SD setting, the input to Algorithm 5 is TDES $\mathbf{G}$ that represents the closed-loop system formed by combining plant and supervisor models defined over the same event set using the synchronous product. For this input, it returns a predicate representing the set of reachable states of this closed-loop system. This predicate is primarily computed at **line 9** by using the definition of $\hat{\delta}$ that is specified for the SD setting.

In the following sections, while discussing symbolic verification of our $||_{SD}$ setting, we will use Algorithm 5 to perform reachability check on the TDES that represents our closed-loop system. In this case, the input to this algorithm will be TDES $\mathbf{G}$ that represents the SD synchronous product of plant and supervisor models, and its output will be the predicate

[11] Readers will find some differences between Algorithm 6.3 given in [36] and our Algorithm 5. This is because of the logical errors that were present in the original algorithm and we have fixed those errors in this version.

containing the set of reachable states of our closed-loop system. As we are using this algorithm in the $||_{SD}$ setting, it is implicit that the algorithm will perform all computations based on the function $\hat{\delta}$ that we have defined for our $||_{SD}$ setting. Please recall that the function $\hat{\delta}(P, \sigma)$ (Definition 9.11) relies on N_σ to compute the predicate, and the definition of N_σ in our $||_{SD}$ setting (Definition 9.9) is different from the SD setting (Definition $B.3$).

Therefore, due to different ways of constructing the input TDES $\mathbf{G}$ that is passed in to this algorithm, **line 9** uses different underlying definitions of $\hat{\delta}$ in the SD and $||_{SD}$ settings. For this reason, the seemingly same looking Algorithm 5 will potentially generate different results in the two different settings.

Coreachability Check Algorithm 6 computes and returns predicate P_1 containing the set of states of input TDES $\mathbf{G}$ that can reach a state satisfying P' by states satisfying P and transitions with events in Σ'.

Algorithm 6 $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$

1: $P_1 \leftarrow P' \wedge P$
2: **repeat**
3: $\quad P_2 \leftarrow P_1$
4: $\quad$ **for** $i \leftarrow 1$ to n **do**
5: $\quad\quad$ **repeat**
6: $\quad\quad\quad P_3 \leftarrow P_1$
7: $\quad\quad\quad P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma' \cap \Sigma_i} (\hat{\delta}^{-1}(P_1, \sigma) \wedge P) \right)$
8: $\quad\quad$ **until** $P_1 \equiv P_3$
9: $\quad$ **end for**
10: **until** $P_1 \equiv P_2$
11: **return** P_1

Like Algorithm 5, we are using Algorithm 6 of the SD setting unchanged in our $||_{SD}$ setting. As explained above, the only difference is the TDES $\mathbf{G}$ that we provide as an input to this algorithm. Based on how TDES $\mathbf{G}$ has been constructed, Algorithm 6 uses the corresponding definition of function $\hat{\delta}^{-1}$, which in turn relies on N_σ, to compute the required predicate in the SD and the $||_{SD}$ setting, as appropriate.

Please note that we will use this algorithm to compute $CR(\mathbf{G}, P)$ which is equivalent to $\mathcal{CR}(\mathbf{G}, P_m, \Sigma, P)$.

9.4 Construction of Closed-Loop System

In our $||_{SD}$ setting, we construct the closed-loop system by synchronizing TDES plant $\mathbf{G}$ and TDES supervisor $\mathbf{S}$ using the SD synchronous product operator, i.e. $\mathbf{S} \,||_{SD}\, \mathbf{G}$. Instead of designing monolithic TDES, if $\mathbf{G}$ and $\mathbf{S}$ are modelled in a modular fashion, then we assume that these component plant and supervisor models are independently combined using product operator to form $\mathbf{G}$ and $\mathbf{S}$ respectively.

In case, if plant and supervisor components are combined using synchronous product, then we can simply add selfloops at every state of the component TDES for events that are missing from their event sets to obtain our $\mathbf{G}$ and $\mathbf{S}$.

For TDES plant components $\mathbf{G}'_i = (Y_i, \Sigma_i, \delta_i, y_{o,i}, Y_{m,i})$ and $\mathbf{G}' = \mathbf{G}'_1 \,||\, \mathbf{G}'_2 \,||\, \ldots \,||\, \mathbf{G}'_k$, where

$i = 1, 2, \ldots, k$, let $\mathbf{G}_i = \mathbf{selfloop}(\mathbf{G}'_i, \Sigma - \Sigma_i)$. The TDES plant $\mathbf{G}$ is then defined as follows:

$$\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_k = (Y, \Sigma, \delta, y_o, Y_m)$$

For modular TDES supervisors $\mathbf{S}'_j = (X_j, \Sigma_j, \xi_j, x_{o,j}, X_{m,j})$ and $\mathbf{S}' = \mathbf{S}'_1 \,||\, \mathbf{S}'_2 \,||\, \ldots \,||\, \mathbf{S}'_n$, where $j = 1, 2, \ldots, n$, let $\mathbf{S}_j = \mathbf{selfloop}(\mathbf{S}'_j, \Sigma - \Sigma_j)$. The TDES supervisor $\mathbf{S}$ is then defined as follows:

$$\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2 \times \ldots \times \mathbf{S}_n = (X, \Sigma, \xi, x_o, X_m)$$

Using this approach, both $\mathbf{G}$ and $\mathbf{S}$ are now defined over the same event set Σ. Finally, we construct our closed-loop system, $\mathbf{G}_{cl}$, as follows:

$$\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Q, \Sigma, \eta, q_o, Q_m)$$

Here, all five elements of $\mathbf{G}_{cl}$'s tuple are defined as per Definition 4.1 of the SD synchronous product operator. Please note that at this stage, $\mathbf{G}_{cl}$ might contain some unreachable states.

Next, we borrow some definitions of the SD setting from [42]. As our strategy of constructing $\mathbf{G}$ and $\mathbf{S}$ from component TDES is same as the SD setting, these definitions work well in our $||_{SD}$ setting just by changing the way of constructing the closed-loop system.

Definition 9.13. Let $\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Q, \Sigma, \eta, q_o, Q_m)$, where $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_k = (Y, \Sigma, \delta, y_o, Y_m)$ and $\mathbf{S} = \mathbf{S}_1 \times \mathbf{S}_2 \times \ldots \times \mathbf{S}_n = (X, \Sigma, \xi, x_o, X_m)$. For a given event $\sigma \in \Sigma$, the *σ plant transition predicate* $N_{\mathbf{G},\sigma} : Q \times Q \to \{T, F\}$ can be expressed as follows:

$$N_{\mathbf{G},\sigma}(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \le i \le k\}} \left(\bigvee_{\{y_i, y'_i \in Y_i \mid \delta_i(y_i, \sigma) = y'_i\}} (v_i = y_i) \wedge (v'_i = y'_i) \right)$$

Likewise, the *σ supervisor transition predicate* $N_{\mathbf{S},\sigma} : Q \times Q \to \{T, F\}$ can be expressed as follows:

$$N_{\mathbf{S},\sigma}(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \le j \le n\}} \left(\bigvee_{\{x_j, x'_j \in X_j \mid \xi_j(x_j, \sigma) = x'_j\}} (v_{j+k} = x_j) \wedge (v'_{j+k} = x'_j) \right)$$

It is noteworthy that $N_{\mathbf{G},\sigma}$ and $N_{\mathbf{S},\sigma}$ are defined on $Q \times Q$ and use the variables $\mathbf{v}$ and $\mathbf{v}'$ like N_σ. We will use $N_{\mathbf{G},\sigma}$ to determine if there is a σ transition defined at the plant portion of the indicated states. Similarly, $N_{\mathbf{S},\sigma}$ will be used to determine if there is a σ transition defined at the supervisor portion of the indicated states. They must be defined over $Q \times Q$ so that we can compare and combine their results with other state predicates on Q.

Definition 9.14. For TDES plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and some $\sigma \in \Sigma$, let $N_{\mathbf{G},\sigma}$ be the σ plant transition predicate. For $P \in Pred(Q)$, the function $\hat{\delta}_{\mathbf{G}} : Pred(Q) \times \Sigma \to Pred(Q)$ is defined as follows:

$$\hat{\delta}_{\mathbf{G}}(P, \sigma) := (\exists \mathbf{v}(N_{\mathbf{G},\sigma} \wedge P))[\mathbf{v}' \to \mathbf{v}]$$

The inverse function $\hat{\delta}^{-1}_{\mathbf{G}} : Pred(Q) \times \Sigma \to Pred(Q)$ is defined as follows:

$$\hat{\delta}^{-1}_{\mathbf{G}}(P, \sigma) := \exists \mathbf{v}'(N_{\mathbf{G},\sigma} \wedge (P[\mathbf{v} \to \mathbf{v}']))$$

Definition 9.15. For TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$ and some $\sigma \in \Sigma$, let $N_{\mathbf{S},\sigma}$ be the σ supervisor transition predicate. For $P \in Pred(Q)$, the function $\hat{\xi} : Pred(Q) \times \Sigma \to Pred(Q)$ is defined as follows:

$$\hat{\xi}(P, \sigma) := (\exists \mathbf{v}(N_{\mathbf{S},\sigma} \wedge P))[\mathbf{v}' \to \mathbf{v}]$$

The inverse function $\hat{\xi}^{-1} : Pred(Q) \times \Sigma \to Pred(Q)$ is defined as follows:

$$\hat{\xi}^{-1}(P, \sigma) := \exists \mathbf{v}'(N_{\mathbf{S},\sigma} \wedge (P[\mathbf{v} \to \mathbf{v}']))$$

9.5 Symbolic Verification

In this section, we discuss predicate-based algorithms from [42] that we have modified to verify various properties of our $||_{SD}$ system. Please note that due to space limitations, we will not provide a detailed explanation for the unchanged parts of these modified algorithms. Please refer to [42] for the complete logical description of all algorithms. The unmodified algorithms that can be used to check other $||_{SD}$ properties are included in Appendix B for the sake of completeness.

With respect to the unchanged algorithms given in Appendix B, we wish to point out that although the steps of these algorithms remain unaltered, the input TDES that we pass in to these algorithms for verification are certainly different than the ones assumed in the SD setting. Also, the underlying definitions for some variables and functions used by these algorithms have changed with respect to our $||_{SD}$ setting. Therefore, in order to use these algorithms to verify properties in our $||_{SD}$ setting, it is an implicit assumption that these algorithms operate on our input, and use the variable and function definitions that we have specified in this section for our $||_{SD}$ setting.

Precisely, algorithms for the following properties remain unchanged in our $||_{SD}$ setting. Please refer to Sections B.2 and B.3 for further details.

1. Nonblocking (Algorithm 12)

2. Activity-loop-free (ALF) (Algorithm 13)

3. Proper time behaviour (Algorithm 14)

4. **S**-singular prohibitable behaviour with $||_{SD}$ (Algorithm 16: lines 12-16)

5. SD controllability with $||_{SD}$

- **i.** Point ii (Algorithms 15, 16, 17, 18, 19)
- **ii.** Point iii (Algorithm 20)

Now we will discuss our modified algorithms for the $||_{SD}$ setting. With TDES plant $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$ and TDES supervisor $\mathbf{S} = (X, \Sigma, \xi, x_o, X_m)$, our closed-loop system is $\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G} = (Q, \Sigma, \eta, q_o, Q_m)$. As per the definition of state set Q in $||_{SD}$ operator, for every state $q \in Q$, there must exist a state $x \in X$ and $y \in Y$ such that $q = (x, y)$.

The system event set Σ is defined as $\Sigma = \Sigma_{hib} \,\dot\cup\, \Sigma_u \,\dot\cup\, \{\tau\}$, where Σ_{hib} and Σ_u represent the set of prohibitable events and uncontrollable events of $\mathbf{G}_{cl}$ respectively. The set of controllable events is $\Sigma_c = \Sigma_{hib} \,\dot\cup\, \{\tau\}$, and the set of activity events is $\Sigma_{act} = \Sigma_{hib} \,\dot\cup\, \Sigma_u$.

9.5.1 Plant Completeness with $||_{SD}$

According to Definition 4.2 of plant completeness with $||_{SD}$ property, the states of $\mathbf{G}_{cl}$, where a prohibitable event is enabled at the corresponding state in **S** but it is not possible at the corresponding state in **G**, are the *incomplete* states that cause this property to fail. We can express the set of these states, $Q_{incomplete}$, and its corresponding predicate $P_{incomplete} := pr(Q_{incomplete})$ as follows:

$$Q_{incomplete} := \{q = (x, y) \in Q \mid (\exists \sigma \in \Sigma_{hib})\, \xi(x, \sigma)! \;\&\; \neg\, \delta(y, \sigma)!\}$$

$$P_{incomplete} := \bigvee_{\sigma \in \Sigma_{hib}} \left(\hat{\xi}^{-1}(true, \sigma) \wedge \neg\, \hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma) \right)$$

Here, $\hat{\xi}^{-1}$ (Definition 9.15) and $\hat{\delta}_{\mathbf{G}}^{-1}$ (Definition 9.14) are the inverse functions for supervisor **S** and plant **G** respectively.

Algorithm 7 CheckPlantCompleteness($\mathbf{G}$, $\mathbf{S}$)

1: $P_{incomplete} \leftarrow false$
2: **for all** $(\sigma \in \Sigma_{hib})$ **do**
3: $\quad P_{incomplete} \leftarrow P_{incomplete} \vee (\hat{\xi}^{-1}(true, \sigma) \wedge \neg\, \hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma))$
4: **end for**
5: $P_{incomplete} \leftarrow P_{incomplete} \wedge R(\mathbf{S}\,||_{SD}\,\mathbf{G}, true)$
6: **if** $(P_{incomplete} \not\equiv false)$ **then**
7: $\quad$ **return** $False$
8: **end if**
9: **return** $True$

Our $\mathbf{G}$ is considered to be complete with $||_{SD}$ for $\mathbf{S}$ if none of the states of $Q_{incomplete}$ are reachable, i.e. $Q_{incomplete} \cap Q_{reach} = \emptyset$, where Q_{reach} is the set of reachable states of $\mathbf{G}_{cl}$. This implies that $P_{incomplete} \wedge P_{reach} \equiv false$, where $P_{reach} := pr(Q_{reach})$ is the predicate representing the set of states in Q_{reach}. Otherwise, $P_{incomplete} \wedge P_{reach}$ contains the set of states that cause this property to fail.

This is the logic used by Algorithm 7[12] to verify plant completeness with $||_{SD}$ property. It is notable that our plant completeness with $||_{SD}$ definition is similar to the plant completeness property (Definition 2.33) except for the actual supervisor TDES and the way of constructing the closed-loop system. Therefore, we are passing our TDES supervisor $\mathbf{S}$ of the $||_{SD}$ setting as input to Algorithm 7. Also, at **line 5** of Algorithm 7, we are performing reachability check on our closed-loop system "$\mathbf{S}\,||_{SD}\,\mathbf{G}$" instead of the closed-loop system of the SD setting "$\mathbf{G} \times \boldsymbol{\mathcal{S}}$" that was used in the original algorithm. The rest of the algorithm steps are essentially unaltered.

9.5.2 Untimed Controllability with $||_{SD}$

The standard untimed controllability property (Definition 2.20) gets redefined as part of the timed controllability with $||_{SD}$ definition (Definition 4.4). Therefore, its corresponding algorithm needs to be amended for use in the $||_{SD}$ setting.

According to Definition 4.5 for untimed controllability with $||_{SD}$, if an uncontrollable event is possible at a state in $\mathbf{G}$ but it is not possible at the corresponding composite state in $\mathbf{G}_{cl}$, then this composite state of $\mathbf{G}_{cl}$ is considered *bad* as it will make our $\mathbf{S}$ uncontrollable with $||_{SD}$ with respect to our $\mathbf{G}$. The set of these bad states, Q_{bad}, and its corresponding predicate $P_{bad} := pr(Q_{bad})$ can be expressed as follows:

$$Q_{bad} := \{q = (x, y) \in Q \mid (\exists \sigma_u \in \Sigma_u)\, \delta(y, \sigma_u)! \;\&\; \neg\, \eta(q, \sigma_u)!\}$$

$$P_{bad} := \bigvee_{\sigma_u \in \Sigma_u} \left(\hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma_u) \wedge \neg\, \hat{\delta}^{-1}(true, \sigma_u) \right)$$

Here, $\hat{\delta}_{\mathbf{G}}^{-1}$ (Definition 9.14) and $\hat{\delta}^{-1}$ (Definition 9.12) are the inverse functions for $\mathbf{G}$ and $\mathbf{G}_{cl}$ respectively.

In order for $\mathbf{S}$ to be untimed controllable with $||_{SD}$ for $\mathbf{G}$, none of the Q_{bad} states should be reachable, i.e $Q_{bad} \cap Q_{reach} = \emptyset$, where Q_{reach} is the set of reachable states of $\mathbf{G}_{cl}$. This implies that $P_{bad} \wedge P_{reach} \equiv false$, where $P_{reach} := pr(Q_{reach})$ is the predicate representing

[12]The value returned by this algorithm is a boolean, $True$ or $False$, instead of a state predicate.

Algorithm 8 CheckUntimedControllability($\mathbf{G}$, $\mathbf{S}$)

1: $P_{bad} \leftarrow false$
2: **for all** $(\sigma_u \in \Sigma_u)$ **do**
3: $\quad P_{bad} \leftarrow P_{bad} \vee (\hat{\delta}^{-1}_{\mathbf{G}}(true, \sigma_u) \wedge \neg\, \hat{\delta}^{-1}(true, \sigma_u))$
4: **end for**
5: $P_{bad} \leftarrow P_{bad} \wedge R(\mathbf{S}\,||_{SD}\,\mathbf{G}, true)$
6: **if** $(P_{bad} \not\equiv false)$ **then**
7: $\quad$ **return** $False$
8: **end if**
9: **return** $True$

the set of states in Q_{reach}. Otherwise, $P_{bad} \wedge P_{reach}$ holds the set of states where $\mathbf{G}_{cl}$ is not allowing an uncontrollable event that is possible at the corresponding state in $\mathbf{G}$.

Algorithm 8 essentially makes use of the above-mentioned logic to verify the untimed controllability with $||_{SD}$ property in our $||_{SD}$ setting. In addition to passing our $||_{SD}$ supervisor $\mathbf{S}$ as an input, our algorithm differs from the original algorithm of the SD setting at **lines 5** and **3**, where we use our closed-loop system, $\mathbf{S}\,||_{SD}\,\mathbf{G}$, and its inverse function, $\hat{\delta}^{-1}$, respectively.

9.5.3 SD Controllability with $||_{SD}$

All algorithms that contribute in verifying the property of SD controllability with $||_{SD}$ assume that $\mathbf{G}$ has proper time behaviour (Algorithm 14) and $\mathbf{G}_{cl}$ is ALF (Algorithm 13). These algorithms make use of several variables and functions to verify the SD controllability with $||_{SD}$ property. We would like to clarify two points about these variables and functions.

1. $\mathbf{G}_{cl}$ stated in these variables and functions refer to our closed-loop system, i.e. $\mathbf{G}_{cl} = \mathbf{S}\,||_{SD}\,\mathbf{G}$, which is different from the original $\mathbf{G}_{cl}$ of the SD setting that was assumed to be constructed as $\mathbf{G} \times \boldsymbol{\mathcal{S}}$.
2. In our $||_{SD}$ setting, the underlying definitions for some of these variables and functions are different from the original ones that were defined in the SD setting. We will explicitly highlight them in our discussion. Since we are using these algorithms in our $||_{SD}$ setting, it is obvious that all variables and functions will be evaluated using the definitions given in this section for our $||_{SD}$ setting.

The following variables and functions are used in the upcoming algorithms:

- P_{reach}: The predicate of the set of reachable states of $\mathbf{G}_{cl}$.
- P_{SF}: The predicate of the set that contains sampled states of $\mathbf{G}_{cl}$ found by the algorithm.
- Z_{SP}: This set contains the predicates of sampled states in $\mathbf{G}_{cl}$ found and not yet analyzed by the algorithm.
- $N_{\mathbf{G},\sigma}, N_{\mathbf{S},\sigma}$: Transition predicates for σ for $\mathbf{G}$ and $\mathbf{S}$ respectively, as in Definition 9.13.
- N_σ: Transition predicate for σ for $\mathbf{G}_{cl}$, as in Definition 9.9. Please note that our definition for N_σ is different from N_σ of the SD setting (Definition $B.3$).
- $\hat{\delta}$: Transition function for state predicates for $\mathbf{G}_{cl}$, as in Definition 9.11. Please recall that $\hat{\delta}$ relies on N_σ which makes the computation logic of $\hat{\delta}$ different in the SD and the $||_{SD}$ setting.
- $\hat{\delta}_{\mathbf{G}}$: Transition function for state predicates for $\mathbf{G}$ only, as in Definition 9.14.

Algorithm 9 CheckSDControllability($\mathbf{G}, \mathbf{S}$)

1: $\mathbf{G}_{cl} \leftarrow \mathbf{S} ||_{SD} \mathbf{G}$
2: $P_{reach} \leftarrow R(\mathbf{S} ||_{SD} \mathbf{G}, true)$
3: **if** (CheckSDCPointi($\mathbf{G}, \mathbf{S}, P_{reach}$) = $False$) **then**
4: **return** $False$
5: **end if**
6: $SDControllable \leftarrow True$
7: $P_{SF} \leftarrow pr\{q_o\}$
8: $Z_{SP} \leftarrow \{pr\{q_o\}\}$
9: $pNerFail \leftarrow \emptyset$
10: **while** ($Z_{SP} \neq \emptyset$) **do**
11: $P_{ss} \leftarrow$ Pop(Z_{SP})
12: $SDControllable \leftarrow$ AnalyzeSampledState($\mathbf{G}, \mathbf{S}, P_{SF}, Z_{SP}, P_{reach}, P_{ss}, pNerFail$)
13: **if** ($\neg\, SDControllable$) **then**
14: **return** $False$
15: **end if**
16: **end while**
17: **if** ($pNerFail \neq \emptyset$) **then**
18: $SDControllable \leftarrow$ RecheckNerodeCells($pNerFail$)
19: **if** ($\neg\, SDControllable$) **then**
20: **return** $False$
21: **end if**
22: **end if**
23: **if** ($\neg$ CheckSDCPointiii(P_{reach})) **then**
24: **return** $False$
25: **end if**
26: **return** $True$

- $\hat{\xi}$: Transition function for state predicates for $\mathbf{S}$ only, as in Definition 9.15.
- $pNerFail$: This set $pNerFail \subseteq$ Pwr($Pred(Q)$) is a set of sets of predicates that stores information where Point ii.2 of SD controllability with $||_{SD}$ property may have failed.
- $SDControllable$: This flag asserts if $\mathbf{S}$ is SD controllable with $||_{SD}$ with respect to $\mathbf{G}$.

Algorithm 9 serves as the entry point for checking various points of the SD controllability with $||_{SD}$ property. At **line 3**, it calls Algorithm 10 to verify Point i of SD controllability with $||_{SD}$ definition. It is note worthy that Point i essentially represents the timed controllability with $||_{SD}$ property which includes the untimed controllability with $||_{SD}$ check. Since we have already discussed Algorithm 8 for verifying untimed controllability with $||_{SD}$, we will not discuss it again. For the same reason, untimed controllability with $||_{SD}$ check is not showing up in Algorithm 10.

In order to verify Point ii of SD controllability with $||_{SD}$, processing starts with the initial state of $\mathbf{G}_{cl}$ which is always a sampled state (**line 7**). As verification proceeds, a reachability tree is created for a given sampling period, and required checks including the check of **S**-singular prohibitable behaviour with $||_{SD}$ are performed (**line 12**). If any of the desired properties fails, the algorithm terminates, except for Point ii.2 where the algorithm continues after recording the problematic nerode cells. These cells and Point ii.2 is then tested again

Algorithm 10 CheckSDCPointi($\mathbf{G}, \mathbf{S}, P_{reach}$)

1: $P_{q-hib} \leftarrow \bigvee_{\sigma \in \Sigma_{hib}} \exists \mathbf{v}' N_\sigma$
2: $P_{bad} \leftarrow \exists \mathbf{v}' N_{\mathbf{G},tick} \wedge \neg (\exists \mathbf{v}' N_{tick}) \wedge \neg P_{q-hib}$
3: **if** $(P_{bad} \wedge P_{reach} \not\equiv false)$ **then**
4: **return** $False$
5: **end if**
6: **return** $True$

afterwards (**line 18**).

Finally, the algorithm verifies Point iii of SD controllability with $||_{SD}$ at **line 23** by making use of Algorithm 20.

Point i Algorithm 10 verifies the timed controllability part of Point i of SD controllability with $||_{SD}$. We have derived this algorithm from Algorithm 11 that was developed by [42] to verify Point ii of SD controllability (Definition 3.7) in the SD setting.

From **lines 2-5**, Algorithm 11 checks the forward implication ($\Rightarrow$) of Point ii of SD controllablility. It determines if there exists a reachable state in $\mathbf{G}_{cl} = \mathbf{G} \times \boldsymbol{\mathcal{S}}$ where both *tick* and prohibitable events are enabled. If such a such exists, Point ii ($\Rightarrow$) fails, and the algorithm returns $False$.

Please recall that Point ii ($\Rightarrow$) of SD controllability does not exist in our definition of SD controllability with $||_{SD}$, and we are not required to check this condition explicitly in our $||_{SD}$ setting. We are able to get rid of this explicit check because of the distinct synchronization mechanism of our $||_{SD}$ operator that guarantees to automatically disable a *tick* event in the closed-loop system $\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G}$, if both *tick* and a prohibitable event is possible in $\mathbf{G}$ and enabled by $\mathbf{S}$. This means that our $||_{SD}$ operator will never enable both *tick* and prohibitable event at any state of $\mathbf{G}_{cl}$ while synchronizing $\mathbf{G}$ and $\mathbf{S}$ to form the closed-loop system. Since this condition is automatically satisfied in our $||_{SD}$ setting, we do not need **lines 2-5** of Algorithm 11 and did not include them in our Algorithm 10.

From **lines 6-9**, Algorithm 11 checks the reverse implication ($\Leftarrow$) of Point ii of SD controllability. It determines if there exists a reachable state in $\mathbf{G}_{cl} = \mathbf{G} \times \boldsymbol{\mathcal{S}}$ where no prohibitable event is eligible, and *tick* is possible in $\mathbf{G}$ but disabled by $\boldsymbol{\mathcal{S}}$. If such a such exists, Point ii ($\Leftarrow$) fails, and the algorithm returns $False$. This is essentially the timed part of the timed controllability definition (Definition 2.22) used in the SD setting.

In our Algorithm 10, we have modified this logic to check our corresponding condition by using our closed-loop system $\mathbf{G}_{cl} = \mathbf{S} \,||_{SD}\, \mathbf{G}$ instead of their supervisor $\boldsymbol{\mathcal{S}}$. This change is in line with the timed part of our timed controllability with $||_{SD}$ property (Definition 4.4) that we have defined for our $||_{SD}$ setting.

Line 1 of Algorithm 10 identifies the states of $\mathbf{G}_{cl}$ that have one or more prohibitable events defined. **Line 2** determines if there exists a *bad* state in $\mathbf{G}_{cl}$ where neither *tick* nor a prohibitable event is eligible in $\mathbf{G}_{cl}$, but *tick* is possible at the corresponding state in $\mathbf{G}$. If such a bad state exists and is reachable in $\mathbf{G}_{cl}$ (**line 3**), then the timed check of Point i of our SD controllability with $||_{SD}$ fails, and the algorithm returns $False$ at **line 4**. Otherwise, it returns $True$ at **line 6**.

Point ii In order to verify Point ii of SD controllability with $||_{SD}$, we will reuse several variables from [42]. Please note that we have redefined two variables, Σ_{poss} and B_{conc}, to

Algorithm 11 CheckSDContii($\mathbf{G}, \boldsymbol{\mathcal{S}}, P_{reach}$)

1: $P_{q-hib} \leftarrow \bigvee_{\sigma \in \Sigma_{hib}} \exists \mathbf{v}' N_\sigma$
2: $P_{bad} \leftarrow \exists \mathbf{v}' N_{tick} \wedge P_{q-hib}$
3: **if** $P_{bad} \wedge P_{reach} \not\equiv false$ **then**
4: **return** $False$
5: **end if**
6: $P_{bad} \leftarrow \exists \mathbf{v}' N_{\mathbf{G},tick} \wedge \neg (\exists \mathbf{v}' N_{\boldsymbol{\mathcal{S}},tick}) \wedge \neg P_{q-hib}$
7: **if** $P_{bad} \wedge P_{reach} \not\equiv false$ **then**
8: **return** $False$
9: **end if**
10: **return** $True$

make the corresponding algorithms compatible with our $||_{SD}$ setting.

- Σ_{Elig}: The set of prohibitable events eligible in both $\mathbf{G}$ and $\mathbf{S}$ at q_{ss}, where q_{ss} is the sampled state in $\mathbf{G}_{cl}$ that we are processing.
- P_q: The predicate of current state in $\mathbf{G}_{cl}$.
- Σ_{poss}: [42] defines this variable to be the set of events eligible in both $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ at predicate P_q of current state in $\mathbf{G}_{cl} = \mathbf{G} \times \boldsymbol{\mathcal{S}}$. This is because every event that is possible in $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$ will be enabled in $\mathbf{G}_{cl}$ by the product operator. However, this might not be true for our $||_{SD}$ operator with respect to the *tick* event.

  In our $||_{SD}$ setting, we define Σ_{poss} to be the set of events that are eligible in $\mathbf{G}_{cl} = \mathbf{S}||_{SD}\,\mathbf{G}$ at predicate P_q of current state in $\mathbf{G}_{cl}$. Therefore, this set contains activity events that are eligible in both $\mathbf{G}$ and $\mathbf{S}$ at predicate P_q of current state in $\mathbf{G}_{cl}$. Additionally, it also contains a *tick* event if it is eligible in $\mathbf{G}$ and $\mathbf{S}$, and no prohibitable event is possible at predicate P_q of current state in $\mathbf{G}_{cl}$ to preempt the *tick*.
- $\Sigma_{\mathbf{G}poss}$: The set of prohibitable events eligible in $\mathbf{G}$ at predicate P_q of current state in $\mathbf{G}_{cl}$.
- *nextLabel*: This number represents the next unused node in B_{map}. It is used to name newly discovered nodes of the reachability tree.
- B_{map}: This partial function $B_{map} : \mathcal{N} \to Pred(Q)$ maps the nodes of the reachability tree to the predicates of the states of $\mathbf{G}_{cl}$ that the nodes represent. This function will sometimes be treated like the set $B_{map} \subseteq \mathcal{N} \times Pred(Q)$. Note that $\mathcal{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers.
- B_p: This is the set of nodes pending to be expanded in the reachability tree.
- B_{conc}: The set $B_{conc} \subseteq \mathcal{N} \times Pred(Q)$ contains the nodes that represent concurrent strings and the sampled states the strings lead to.

  In [42], for $(b, q) \in B_{conc}$, node b is a node at which *tick* is eligible in $\mathbf{G}$ and $\boldsymbol{\mathcal{S}}$, and q is the sampled state of $\mathbf{G}_{cl} = \mathbf{G} \times \boldsymbol{\mathcal{S}}$ that the *tick* leads to.

  In our $||_{SD}$ setting, for $(b, q) \in B_{conc}$, we define b to be a node at which *tick* is eligible in $\mathbf{G}_{cl} = \mathbf{S}||_{SD}\,\mathbf{G}$, and q is the sampled state of $\mathbf{G}_{cl}$ that the *tick* leads to.
- Occu_B: The partial function $\text{Occu}_B : \mathcal{N} \to \text{Pwr}(\Sigma)$ maps the nodes of the reachability tree to the occurrence image of the string that they represent. This function will sometimes be treated like the set $\text{Occu}_B \subseteq \mathcal{N} \times \text{Pwr}(\Sigma)$.

The actual algorithm steps to verify Points ii.1, ii.2 and iii of SD controllability with $||_{SD}$ are

essentially the same as [42]. Please refer to Section B.3 to get an overview of these unmodified algorithms. This includes any function calls in Algorithm 9 that have not been discussed in this section. Please note that in order to verify Points ii and iii of SD controllability with $||_{SD}$ property, all these algorithms make use of the variable and function definitions specified in this section for our $||_{SD}$ setting.

10 Flexible Manufacturing System

In this section, we present an example of a Flexible Manufacturing System (FMS) to demonstrate the application, utilization and benefits of our SD synchronous product operator and the $||_{SD}$ setting. This is the same TDES example that has been discussed in [42, 43] to illustrate the SD supervisory control methodology, who in turn based it on the untimed FMS example given in [21]. We have intentionally selected the same system so that we could clearly compare and discuss the complexity of designing modular TDES supervisors by hand and the size of resultant supervisor models in the SD and $||_{SD}$ setting, i.e. in the absence and presence of our $||_{SD}$ operator.

We begin this section by describing the structure and workflow of the FMS. Then, we provide its various TDES plant components. After that, we analyze the original design of each modular TDES supervisor developed in the SD setting and discuss how it gets simplified in our $||_{SD}$ setting in the presence of the $||_{SD}$ operator. Finally, we close this section by presenting a comprehensive discussion on our software implementation and verification results for the FMS example.

10.1 System Structure

The Flexible Manufacturing System (FMS), shown in Figure 1, consists of six machines and five buffers, where each buffer has the capacity to hold a single part. These buffers are treated as specifications and it is desired that buffers never overflow or underflow.

The basic idea of the FMS is that a part enters the system via conveyor Con2 and passes to a handling Robot via buffer B2. The Robot then passes the part to Lathe via buffer B4. The Lathe can generate two types of parts, A and B. Once the Robot receives the part back from Lathe via B4, it sends the part either to buffer B6 or B7 depending upon the part type. Precisely, type A part goes to B6 while type B part goes to B7. From B7, part B goes to a painting machine PM via conveyor Con3 and buffer B8. After completing its operation, PM returns the part to B7 via the same route. From B6 and B7, the part goes to the finishing machine AM, from where the finished part finally exits the system.

Table 1 shows the mapping of numeric event labels, used in [43], to their meaning. Odd numbered event labels represent prohibitable events, whereas even numbered labels represent unconrollable events. Instead of the numeric labels, we will use the meaningful shorthand event labels in our TDES models for the sake of readability and comprehension. Our shorthand corresponding to each event label in given in Table 1.

It is notable that there are five event labels in Table 1 that are prefixed by "no". These labels do not represent any physical events of the FMS. Rather, they are *prohibitable expansion events* that were introduced by [43] to aid in communication between various modular TDES supervisors in order to satisfy the properties of the SD supervisory control methodology. We will discuss these events further in our subsequent sections.

Table 1: Meaning and Shorthand for Event Labels of FMS

Label	Meaning	Shorthand	Label	Meaning	Shorthand
921	Part enters system	pt_ent_sys	922	Part enters B2	pt_ent_B2
933	Robot takes from B2	R_from_B2	934	Robot to B4	R_to_B4
937	B4 to Robot for B6	B4_to_R_for_B6	938	Robot to B6	R_to_B6
939	B4 to Robot for B7	B4_to_R_for_B7	930	Robot to B7	R_to_B7
951	B4 to Lathe (A)	B4_to_L_A	952	Lathe to B4 (A)	L_to_B4_A
953	B4 to Lathe (B)	B4_to_L_B	954	Lathe to B4 (B)	L_to_B4_B
961	Initialize AM	init_AM	963	B6 to AM	B6_to_AM
964	Finished from B6	fin_from_B6	965	B7 to AM	B7_to_AM
966	Finished from B7	fin_from_B7	971	B7 to Con3	B7_to_C3
no921	No part enters system	no_pt_ent_sys	972	Con3 to B8	C3_to_B8
no963a	No B6 to AM (a)	no_B6_to_AM_a	973	B8 to Con3	B8_to_C3
no963b	No B6 to AM (b)	no_B6_to_AM_b	974	Con3 to B7	C3_to_B7
no965a	No B7 to AM (a)	no_B7_to_AM_a	981	B8 to PM	B8_to_PM
no965b	No B7 to AM (b)	no_B7_to_AM_b	982	PM to B8	PM_to_B8

Please recall from Section 2.3 that in the graphical TDES models, an event name given in italics and preceded by "!" indicates an uncontrollable event, a double circle represents the initial state, and a filled circle shows that the state is marked.

10.2 Plant Components

The FMS consists of six plant components: two conveyors **Con2** and **Con3**, **Robot**, **Lathe**, **PM** and **AM**. Their TDES models are shown in Figures 17-22. One more TDES plant model, **SysDownNup**, is given in Figure 23. This plant component is added to introduce a shutdown mechanism in the FMS. This could correspond to a physical switch to turn off/restart the system. When the *shutdown* event occurs, Con2 stops accepting new parts and all existing parts exit the system after being processed. In the shutdown state, all components of the physical system go idle, i.e. return to their marked states in the corresponding TDES models. The *restart* event brings the system back up again.

As all these plant models represent the actual uncontrolled behaviour of the physical system, they are the same in the SD and $||_{SD}$ setting. This will make it easy for us to compare the design of modular TDES supervisors in the two settings, since in both cases, supervisors needed to be designed for the same TDES plant components and the same specifications.

In order to introduce five prohibitable expansion events to the system, three additional plant components were added by [43] as part of the supervisor design. These plant TDES are shown in Figures 24-26. We will examine them further in the next section while discussing the design of modular supervisors.

10.3 Modular Supervisors

Now we will discuss the design of modular TDES supervisors for FMS in the $||_{SD}$ setting. Our approach is to first present the modular TDES supervisors that were originally designed in the SD setting, and then discuss how they get simplified in the presence of our $||_{SD}$ operator by comparing them with the TDES supervisor models that we have designed for our $||_{SD}$ setting. Essentially, there are two key takeaways from our discussion presented in this section:

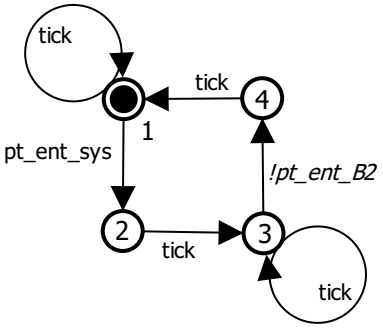


Figure 17: Conveyor **Con2**

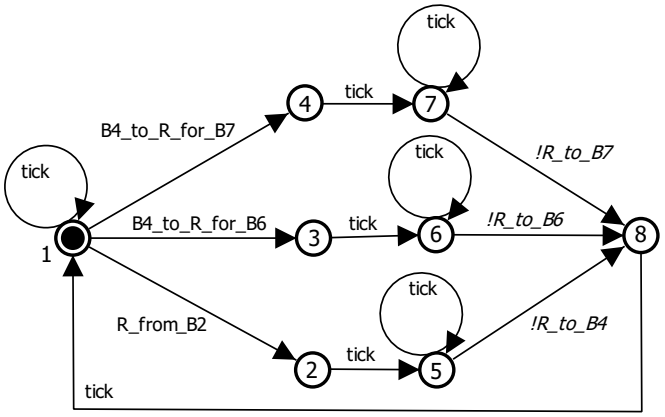


Figure 18: **Robot**

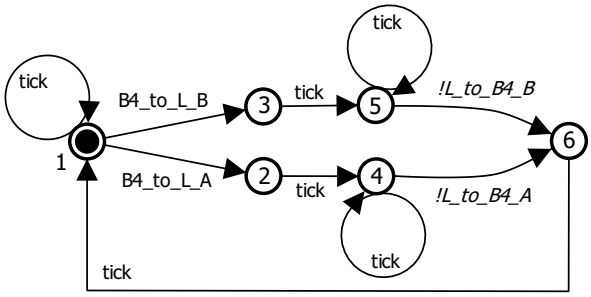


Figure 19: **Lathe**

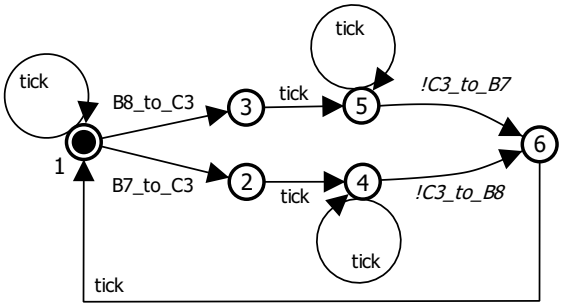


Figure 20: Conveyor **Con3**

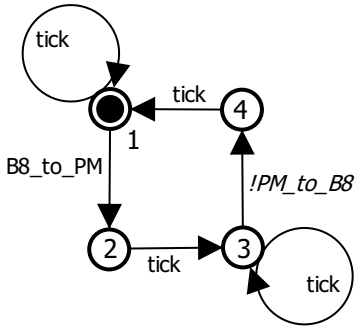


Figure 21: Painting Machine **PM**

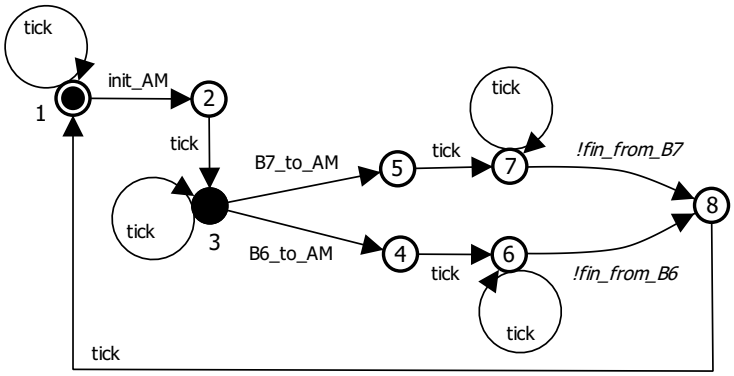


Figure 22: Finishing Machine **AM**

1. It is noticeable how easy it becomes for the designers to design modular TDES supervisors in the presence of our $||_{SD}$ operator and satisfy the same system specifications. This is because they no longer need to manually keep track of the enablement/disablement of *tick* and prohibitable events, nor incorporate this logic explicitly in various supervisor models.

2. It is striking how the size and logical complexity of many of the modular supervisors get reduced in the presence of our $||_{SD}$ operator. The reason is that different supervisor models do not need to communicate with each other and keep track of each other's behaviour about enablement/disablement of *tick* event and forcing of prohibitable events. Also, as

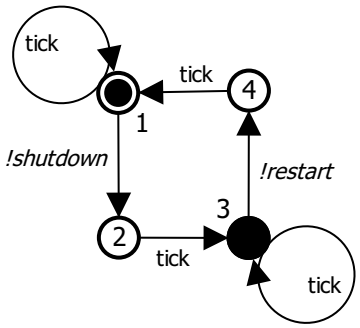


Figure 23: **SysDownNup**

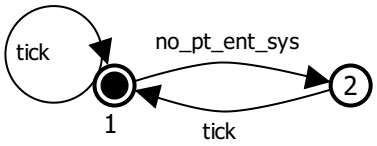


Figure 24:
AddNoPtEntSys

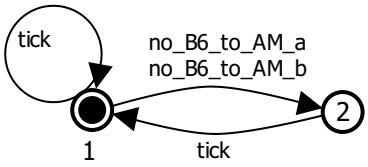


Figure 25:
AddNoB6toAM

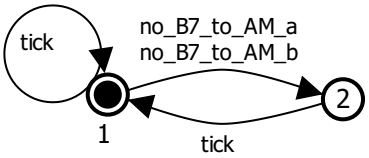


Figure 26:
AddNoB7toAM

we will see in Section 10.4, these simplifications make it easy and efficient to verify different properties of the closed-loop system.

Please note that in order to clearly differentiate between TDES models of the SD supervisory control and our $||_{SD}$ setting, names of TDES plants and supervisors that are used in the SD setting but are removed or modified in the $||_{SD}$ setting, will be stated in ***bold italics***. We will refer to TDES supervisors that appear in the $||_{SD}$ setting using **bold text** only.

10.3.1 Buffer Supervisors

Buffer supervisors control the flow of parts in and out of the buffers. They are primarily responsible for making sure that buffers do not overflow or underflow. Please note that while discussing buffer supervisor **B2**, we will also examine supervisor **HndlSysDwn**, as these two supervisors are closely related with respect to the shutdown/restart mechanism of the FMS.

B2 and HndlSysDwn Supervisor ***B2***, shown in Figure 27, is designed in the SD setting to make sure that buffer B2 does not overflow or underflow. It guarantees this by watching the part's progress once a new part enters the system (*pt_ent_sys*). ***B2*** first waits for the part to enter buffer B2 by keeping track of event *pt_ent_B2*, and then it allows Robot to take the part from B2 by enabling the prohibitable event *R_from_B2*. This prevents the underflow of buffer B2. ***B2*** also ensures that another part does not enter the system (*pt_ent_sys*) until the previous part has been removed from buffer B2 (*R_from_B2*), thus preventing overflow.

Another crucial task performed by ***B2*** is to decide when to force the prohibitable event *pt_ent_sys*. As soon as the system is turned on, ***B2*** causes Con2 to accept a new part into the system by enabling and forcing *pt_ent_sys*. Please recall that in order to force a prohibitable event in the SD supervisory control theory, *tick* event must be explicitly disabled by the supervisor to satisfy Point ii ($\Rightarrow$) of SD controllability. For this reason, *tick* has been disabled at state 0 of ***B2*** to force *pt_ent_sys*.

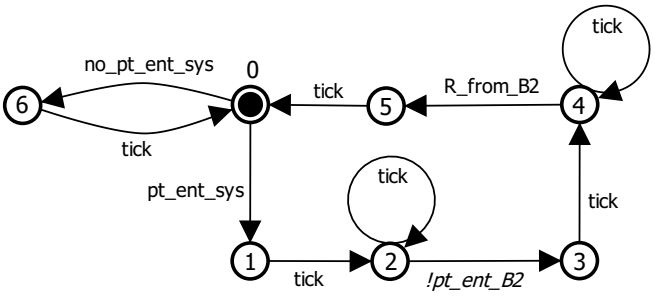


Figure 27: Supervisor ***B2***

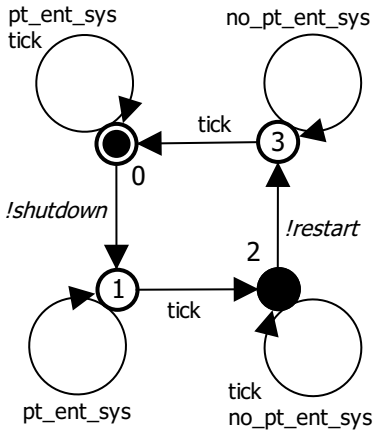


Figure 28: Supervisor ***HndlSysDwn***

By looking at supervisor ***B2***, we observe that as soon as Robot takes the part (*R_from_B2*) and buffer B2 becomes empty, ***B2*** allows a new part to enter the system by forcing *pt_ent_sys*. This behaviour is acceptable as long as the system is up and running. However, once the system is shutdown, then Con2 must stop accepting new parts and system must empty out after processing the existing parts so that all machines can go to their idle (marked) states, as desired by the system specifications.

This means that after shutdown, *pt_ent_sys* needs to be disabled and must not be forced anymore, until the system is restarted. Since *tick* has already been disabled at state 0 to force *pt_ent_sys*, designers must figure out some other way to stop forcing *pt_ent_sys* without duplicating information from other parts of the system. Moreover, if *pt_ent_sys* is not forced at state 0, some other prohibitable event needs to be forced in the absence of an eligible *tick* event. Otherwise, the system becomes uncontrollable.

In order to resolve this issue in the SD setting, a prohibitable expansion event *no_pt_ent_sys* is introduced to the system by designing and including an additional plant TDES, ***AddNoPtEntSys*** (Figure 24). At state 0 of supervisor ***B2***, a loop of concurrent string "*no_pt_ent_sys—tick*" is added that allows ***B2*** to force *no_pt_ent_sys* when *pt_ent_sys* needs to be disabled to achieve the desired behaviour, while keeping the system controllable and not "stopping the clock".

Another supervisor ***HndlSysDwn***, shown in Figure 28, is designed in the SD setting to make sure that the two events, *pt_ent_sys* and *no_pt_ent_sys*, are enabled and disabled at the right time. Specifically, when the system is initially turned on or restarted, ***HndlSysDwn*** enables *pt_ent_sys* and disables *no_pt_ent_sys* to allow new parts to enter the system for processing. When the system is shutdown, it enables *no_pt_ent_sys* and disables *pt_ent_sys* to stop Con2 from accepting new parts into the system.

This discussion clearly shows that forcing a prohibitable event by explicitly disabling *tick*, along with making sure that system does not become uncontrollable is not a straightforward and trouble-free task. In this simple example, when prohibitable event *pt_ent_sys* is under the control of only two modular supervisors, designers have to add one extra plant TDES, a prohibitable expansion event and several additional transitions in the supervisor models to specify the correct forcing mechanism.

All this extra design effort is required because the logic for forcing a prohibitable event

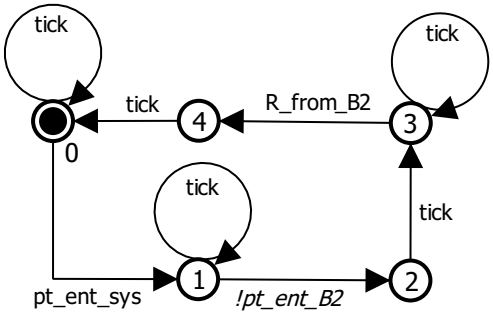


Figure 29: Supervisor **B2**

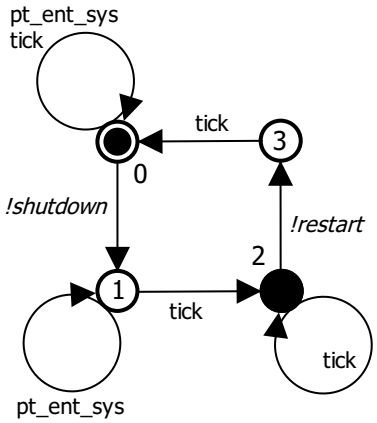


Figure 30: Supervisor **HndlSysDwn**

needed to be manually specified in the supervisor model, and *tick* event was explicitly disabled at one state of the supervisor in order to force the prohibitable event. Certainly, this situation becomes more complicated when the prohibitable event to be forced is under the control of several modular supervisors. Not to mention, the behaviour of the plant model also needs to be considered to make sure that prohibitable event is possible in the plant when supervisor models are collectively trying to force it. We will see a glimpse of this intricate situation later in Section 10.3.4.

Now we will discuss our buffer supervisor **B2**, shown in Figure 29, that we have designed in our $||_{SD}$ setting. Precisely, we have derived **B2** from ***B2*** by trimming away its extra design logic that is not required in our $||_{SD}$ setting.

In the $||_{SD}$ setting, we do not need to manually decide when to force a prohibitable event, nor incorporate this logic explicitly in any of the supervisor models. Rather, we can simply enable a prohibitable event to indicate that we want this event to occur, without disabling the *tick* and the $||_{SD}$ operator will force the event automatically (by deleting the *tick*) as soon as event is enabled by all supervisors, and the event is possible in the plant. That is why, we have enabled both *tick* and prohibitable event *pt_ent_sys* at state 0 of our supervisor **B2**.

Since **B2** is not disabling *tick* at state 0, we do not need to worry about the logic of figuring out how to keep our system controllable with $||_{SD}$ if *pt_ent_sys* cannot or should not be forced. In other words, we are not required to have any alternative expansion event to force in place of *pt_ent_sys* in order to make sure that we do not "stop the clock". This implies that the above-mentioned issue, that designers had to face and resolve in order to explicitly force a prohibitable event while designing supervisors in the SD setting, does not exist in our $||_{SD}$ setting. The development and use of the $||_{SD}$ synchronization operator has completely and permanently resolved this issue in our $||_{SD}$ setting.

As a result, we have altogether removed the expansion event *no_pt_ent_sys* from our FMS plant and supervisor models designed in the $||_{SD}$ setting. Specifically, we are able to remove plant TDES ***AddNoPtEntSys*** from the system. In supervisor **B2**, we have not defined the concurrent string of "$no\_pt\_ent\_sys - tick$" at state 0. Also, our supervisor **HndlSysDwn** shown in Figure 30, that we have developed corresponding to supervisor ***HndlSysDwn*** of the SD setting, does not include any transitions to enable *no_pt_ent_sys* once the system has been shutdown.

Another simplification is that we have removed the state changing *tick* transition between

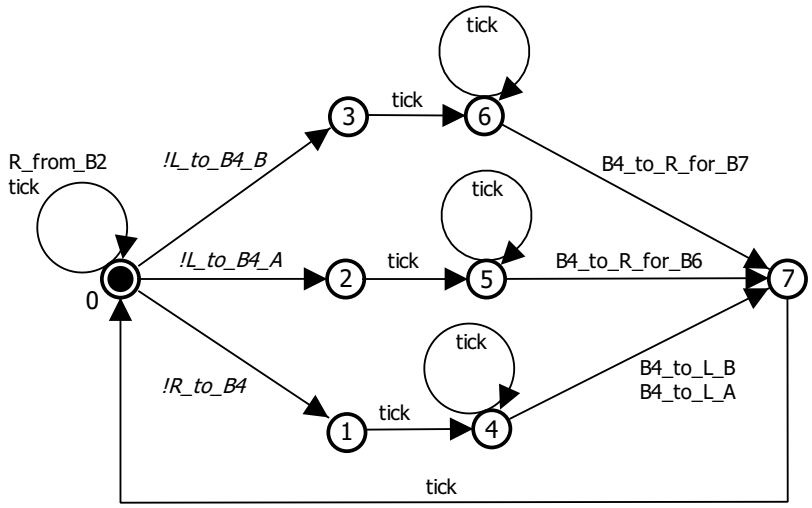


Figure 31: Supervisor **B4**

events *pt_ent_sys* and *pt_ent_B2* of ***B2***, and only included a selfloop of *tick* event at state 1 in our supervisor **B2**. This is because our plant model **Con2** already guarantees that these two events cannot occur in the same sampling period. Therefore, there is no need to replicate this logic in **B2**. Due to the same system specifications, the rest of the logic of our supervisors **B2** and **HndlSysDwn** is the same as their corresponding supervisors ***B2*** and ***HndlSysDwn*** of the SD setting.

B4 Supervisor **B4**, shown in Figure 31, has been designed in the SD setting to fulfill the specification that buffer B4 never overflows or underflows. It ensures this by enabling/disabling related events at the right time. Since this supervisor does not force any prohibitable event, its design remains unchanged in our $||_{SD}$ setting.

An additional role performed by supervisor **B4** is to ensure that once a part enters buffer B4, the correct follow-up action is performed to take it out of B4. To do this, it first makes sure that once a part is moved from buffer B2 to B4, it does go to Lathe for processing. This is ensured by enabling events *B4_to_L_A*/*B4_to_L_B* after *R_to_B4*. Also, supervisor **B4** assures that after being processed by Lathe, the part goes to the correct buffer, B6 or B7, depending upon its type. It guarantees this by enabling event *B4_to_R_for_B6* after a type A part is generated by Lathe and put into buffer B4 (*L_to_B4_A*), and enabling event *B4_to_R_for_B7* after a type B part is produced by Lathe and placed into buffer B4 (*L_to_B4_B*). We will need this information while discussing supervisors in Sections 10.3.2 and 10.3.3.

B6 and B7 In order to prevent the overflow and underflow of buffers B6 and B7, TDES supervisors **B6** (Figure 32) and **B7** (Figure 33) have been designed in the SD setting. These supervisors are strictly responsible for enabling/disabling prohibitable events to manage their respective buffers. Since they do not force any prohibitable event by explicitly disabling the *tick* event, they remain unchanged for our $||_{SD}$ setting.

B8 Figure 34 shows buffer supervisor ***B8*** of the SD setting. ***B8*** not only prevents the overflow and underflow of buffer B8, it also controls the flow of parts once the part arrives at buffer B7 (*R_to_B7*), goes to PM and then comes back to B7. It does this by watching the part's progress and then forcing prohibitable events *B7_to_C3*, *B8_to_PM* and *B8_to_C3*

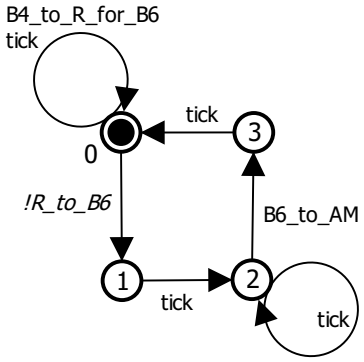


Figure 32: Supervisor **B6**

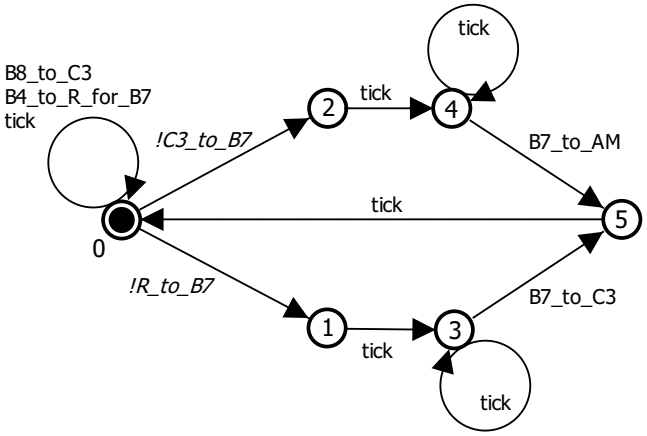


Figure 33: Supervisor **B7**

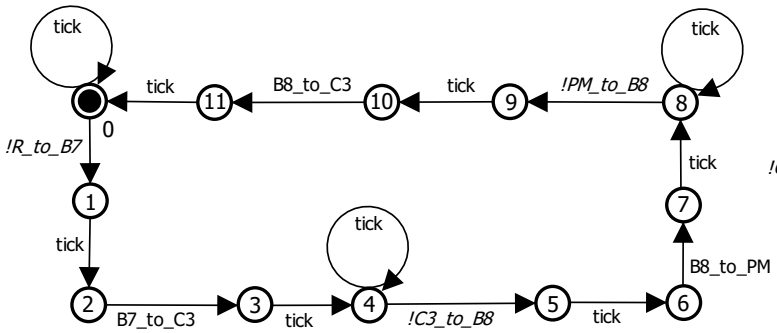


Figure 34: Supervisor ***B8***

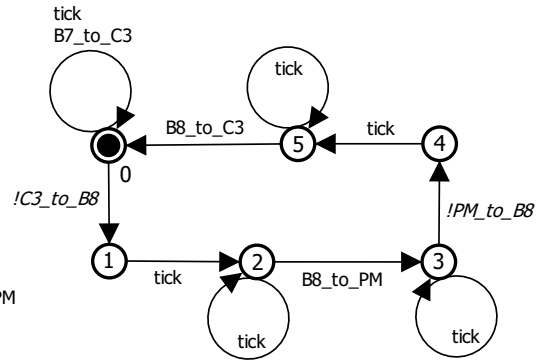


Figure 35: Supervisor **B8**

as needed, by explicitly disabling *tick* at states 2, 6 and 10 respectively to manually satisfy Point ii ($\Rightarrow$) of SD controllability.

It is notable that prohibitable events *B7_to_C3* and *B8_to_C3* are also under the control of supervisor **B7**. This means that ***B8*** needs to make sure that these events must be enabled by **B7** and possible in plant TDES **Con3** before it tries to force them by disabling the *tick*, so that supervisor model does not become uncontrollable with respect to **G**.

In order to force the prohibitable event *B7_to_C3*, ***B8*** needs to know that the part has arrived at buffer B7. This is achieved by replicating the logic of supervisor **B7** into ***B8***, i.e. by repeating the sequence of events "*R_to_B7* − *tick*" in ***B8***. The fact that ***B8*** cannot just enable a prohibitable event without knowing the part's progress and other supervisor's current behaviour, and needs to explicitly disable *tick* at the right time to force the prohibitable event has made things overly complicated and redundant. This point is also highlighted by [43] while discussing the design of their FMS supervisors.

Figure 35 shows buffer supervisor **B8** that we have designed for our $||_{SD}$ setting, with its state size being half (6 states) as compared to the original supervisor ***B8*** (12 states). This is because in the presence of the $||_{SD}$ operator, **B8** can simply enable prohibitable events without explicitly deciding when to force them. Therefore, **B8** does not need to have redundant logic to keep track of the part's progress and supervisor **B7**'s behaviour. Consequently, **B8** gets simplified in two major ways.

First, we have not duplicated the related logic of supervisor **B7** in **B8** by excluding the uncontrollable event *R_to_B7* from **B8**. Second, since **B8** does not need to explicitly decide when to force prohibitable events, we have enabled both *tick* and prohibitable events *B7_to_C3*, *B8_to_PM* and *B8_to_C3* at states 0, 2 and 5 respectively of **B8**. Our $||_{SD}$ operator will automatically disable *tick* and force the appropriate prohibitable event when it is enabled by all concerned supervisors and possible in the plant model **G**, thus keeping our system controllable with $||_{SD}$.

It is worth-mentioning that although we have added a selfloop of prohibitable event *B7_to_C3* at state 0 of supervisor **B8**, this prohibitable event cannot happen more than once in the same sampling period. This is because our **G** is required to have **S**-singular prohibitable behaviour with $||_{SD}$ with respect to our supervisor model **S**. Moreover, once *B7_to_C3* has occurred in the given sampling period, it will be disabled by supervisor **B7** anyway.

The fact that we are able to enable both *tick* and prohibitable event *B7_to_C3* at state 0 of supervisor **B8** due to our $||_{SD}$ operator has also allowed us to remove two explicit state changing *tick* transitions that were present in the original supervisor ***B8***, and include only a selfloop of *tick* event at state 0 in **B8**. First, we have omitted the state changing *tick* transition between events *B7_to_C3* and *C3_to_B8*. The reason being that our plant model **Con3** makes sure that *tick* always happens between these two events, and **B8** is not preventing this *tick* from occurring by explicitly forcing any event. Second, we have eliminated the state changing *tick* transition after event *B8_to_C3*. This is due to the fact that supervisor **B7** and plant component **Con3** already ensure that *B8_to_C3* and *B7_to_C3* do not happen one after another in the same sampling period. **Con3** also guarantees that *B8_to_C3* and *C3_to_B8* occur in different sampling periods. Therefore, there is no need to replicate this logic in **B8**. We have also removed the redundant logic of state changing *tick* transition between *B8_to_PM* and *PM_to_B8* of ***B8*** and replaced it with a selfloop of *tick* event at state 3 in **B8** due to the plant TDES **PM**.

10.3.2 Robot to B4 to Lathe Path

In order to resolve some nonblocking and concurrency issues along the Robot to B4 to Lathe path of the FMS, three supervisors are designed in the SD setting: ***TakeB2***, ***B4Path*** and ***LathePick***. We will discuss them one by one along with the simplifications that we have made while redesigning them for our $||_{SD}$ setting.

TakeB2 Please note that we have already described and compared the design of two supervisors, ***TakeB2*** of the SD setting and **TakeB2** of our $||_{SD}$ setting, with respect to their event forcing logic in Sections 1.2 and 1.3. Below, we only focus on those details and simplifications that we have not discussed before.

In the FMS, the Robot is responsible for serving buffers B2 and B4. Since both buffers cannot be served at the same time, it is essential to dictate the order in which Robot should provide service to these buffers without blocking the system or starving any one of them. This order of service is specified by supervisor ***TakeB2*** of the SD setting given in Figure 4.

TakeB2 forces the Robot to first serve buffer B2, followed by buffer B4, and then alternate between the two. It waits until there is a part in buffer B2 by watching event *pt_ent_B2*, after which it moves the part to buffer B4 by forcing the prohibitable event *R_from_B2* and disabling *tick* at state 2. It does not allow the Robot to serve B2 again, i.e. force another

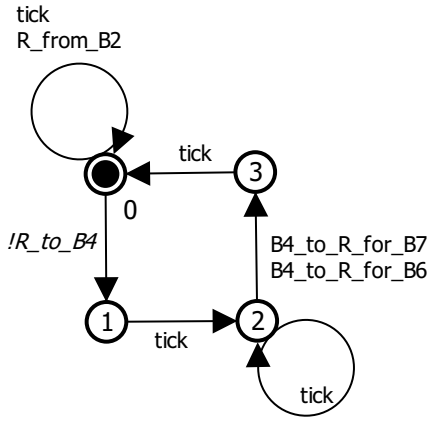


Figure 36: Supervisor ***B4Path***

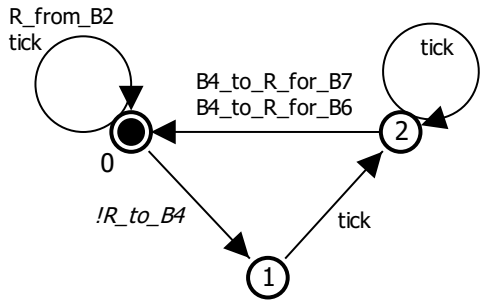


Figure 37: Supervisor **B4Path**

R_from_B2, until Robot has moved the previous part to either buffer B6 (*R_to_B6*) or B7 (*R_to_B7*) from B4 after being processed by Lathe.

This alternate order of serving buffers B2 and B4 also prevents the potential blocking issue that is likely to happen if Robot is allowed to serve buffer B2 two times in a row. In this case, Robot might move second part from B2 to now empty buffer B4 while first part is being processed by Lathe, thus leaving no place for the first part to return to B4.

Figure 5 shows our supervisor **TakeB2** of the $||_{SD}$ setting that specifies the same order for serving buffers B2 and B4 by Robot and addresses the blocking issue but in a much simplified way, i.e. reducing 8-state supervisor ***TakeB2*** to 3-state **TakeB2**.

It is notable that ***TakeB2*** also makes sure that events *pt_ent_B2* and *R_from_B2* do not occur in the same sampling period. This is already ensured by supervisor **B2**, therefore we have not cloned this logic in **TakeB2**. Also, we have replaced the explicit state changing *tick* transition between *R_from_B2* and *R_to_B6*/*R_to_B7* of ***TakeB2*** with a selfloop of *tick* at state 1 in **TakeB2** due to the plant model **Robot**.

B4Path Supervisor ***B4Path***, given in Figure 36, works with buffer supervisor **B4** (Figure 31) to ensure proper behaviour on the Robot−B4−Lathe path. It contributes to the correct behaviour of the system by disabling *R_from_B2* once a part is moved to buffer B4 from B2 (*R_to_B4*). Also, only after moving the part from B2 to B4, it enables *B4_to_R_for_B6* and *B4_to_R_for_B7*.

Figure 37 shows our supervisor **B4Path** that fulfills the same specification as ***B4Path***. The only way in which the two supervisors differ is that unlike ***B4Path***, **B4Path** does not contain an explicit state changing *tick* transition after event *B4_to_R_for_B6*/*B4_to_R_for_B7*. We are able to skip this transition because our plant model **Robot** and buffer supervisor **B4** already ensure that these two events occur in different sampling periods than events *R_from_B2* and *R_to_B4*, as desired.

LathePick Supervisor ***LathePick***, shown in Figure 38, specifies the order for producing two types of parts by Lathe. It forces Lathe to start with type A part, then produce type B part, and then alternate between the two. It does this by forcing prohibitable events *B4_to_L_A* and *B4_to_L_B* at states 2 and 6 respectively.

As supervisor **B4** is also in charge of enabling/disabling events *B4_to_L_A* and *B4_to_L_B*, ***LathePick*** needs to have knowledge about the behaviour of **B4** so that it can make its forcing

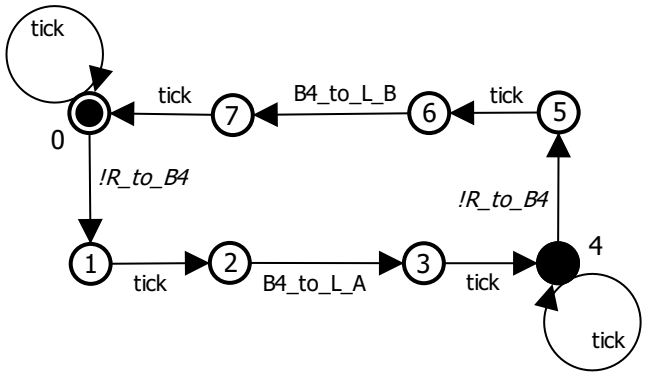


Figure 38: Supervisor ***LathePick***

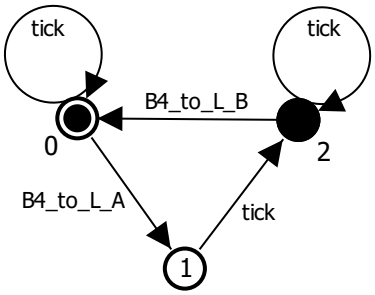


Figure 39: Supervisor **LathePick**

decisions correctly. We note that **B4** enables these two events after the occurrence of event *R_to_B4*. Therefore, ***LathePick*** replicates this logic from **B4** and waits for the occurrence of event *R_to_B4*. Once *R_to_B4* happens, then ***LathePick*** forces *B4_to_L_A*/*B4_to_L_B* to avoid any controllability issues.

Figure 39 illustrates our 3-state supervisor **LathePick** that does the same job as the 8-state ***LathePick*** supervisor. In the $||_{SD}$ setting, since our supervisors are not required to decide precisely when to force a prohibitable event, therefore we have not included event *R_to_B4* in **LathePick**. Also, **LathePick** enables both *tick* and prohibitable events *B4_to_L_A* and *B4_to_L_B* at states 0 and 2 respectively, leaving it up to the $||_{SD}$ operator to make the forcing decision while keeping the supervisor controllable with $||_{SD}$ with respect to **G**.

It is notable that supervisor **B4** and plant model **Lathe** guarantee that events *B4_to_L_B* and *B4_to_L_A* occur in different sampling periods. That is why, we have not added an explicit state changing *tick* transition after *B4_to_L_B* in **LathePick**.

10.3.3 Moving Parts from B4 to B6/B7

In order to resolve some nonblocking and concurrency issues associated with moving parts from buffer B4 to B6 and B7, two supervisors, ***TakeB4PutB6*** and ***TakeB4PutB7***, are designed in the SD setting. Below, we discuss the original design of these supervisors followed by their remodelling for our $||_{SD}$ setting.

TakeB4PutB6 The primary purpose of designing supervisor ***TakeB4PutB6***, shown in Figure 40, in the SD setting is to decide when to force event *B4_to_R_for_B6*. As this prohibitable event is under the control of three other modular supervisors, **B4**, ***B4Path*** and **B6**, ***TakeB4PutB6*** must not try to force *B4_to_R_for_B6* when it is disabled by any of the other supervisors, or not possible in plant TDES **Robot**.

In order to have knowledge about the behaviour of the other models, ***TakeB4PutB6*** duplicates the logic by watching for event *L_to_B4_A*. As soon as type A part enters buffer B4 from Lathe (*L_to_B4_A*), ***TakeB4PutB6*** forces *B4_to_R_for_B6* to initiate the movement of part A from B4 to B6. It then waits for event *B6_to_AM*, signalling that the part has been moved from B6 to AM and now B6 is ready to accept another part A. ***TakeB4PutB6*** also makes sure that *L_to_B4_A* interleaves properly with *B6_to_AM* by specifying the logic for these events to occur in any order.

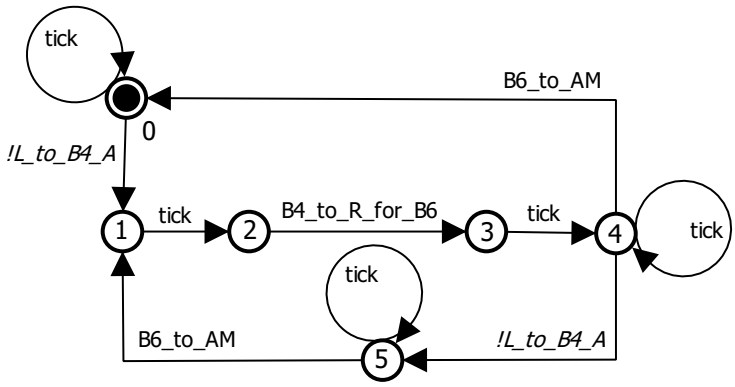


Figure 40: Supervisor ***TakeB4PutB6***

In our $||_{SD}$ setting, we do not need to design and include any supervisor corresponding to ***TakeB4PutB6*** because of the following two reasons: 1) In the presence of the $||_{SD}$ operator, we are not required to explicitly decide and specify when to force *B4_to_R_for_B6* by keeping track of other supervisors' behaviour. In fact, when *B4_to_R_for_B6* is possible in **Robot** and enabled by supervisors **B4**, ***B4Path*** and **B6**, the $||_{SD}$ operator will automatically disable *tick* to force *B4_to_R_for_B6* in the closed-loop system. 2) Our buffer supervisor **B6** already ensures that Robot cannot begin to move type A part from buffer B4 to B6 (*B4_to_R_for_B6*) until the previous part has been taken out of B6 and moved to AM (*B6_to_AM*). **B6** guarantees this by disabling event *B4_to_R_for_B6* once it has happened, and re-enables it only after event *B6_to_AM* has occurred.

As a result, we do not need to specify/replicate any logic and no supervisor exists in our $||_{SD}$ setting corresponding to supervisor ***TakeB4PutB6*** of the SD setting.

TakeB4PutB7 Supervisor ***TakeB4PutB7*** designed in the SD setting is shown in Figure 41. Besides deciding when to force the prohibitable event *B4_to_R_for_B7* to initiate the movement of type B part from buffer B4 to B7, ***TakeB4PutB7*** also handles a potential blocking issue as part B moves along the B7−PM−B7 path.

In order to determine when to force the prohibitable event *B4_to_R_for_B7*, ***TakeB4PutB7*** must take into account the behaviour of supervisors **B4**, ***B4Path*** and **B7**, and plant model **Robot**, as these models are also in charge of enabling/disabling *B4_to_R_for_B7*. Therefore, event *L_to_B4_B* is added to ***TakeB4PutB7*** to replicate the related logic from supervisor **B4** and determine the right time for forcing *B4_to_R_for_B7*. As soon as *L_to_B4_B* occurs, ***TakeB4PutB7*** disables *tick* to force *B4_to_R_for_B7* at state 2.

When part B is placed in buffer B7 from B4, it first goes to PM for processing. It is possible that another part B is put in the now empty buffer B7 by Robot, leaving no place for the returning part, thus blocking the system. Supervisor ***TakeB4PutB7*** prevents this situation from happening by waiting for the part to return to buffer B7 from PM and then moved to AM (*B7_to_AM*), before allowing the Robot to take another part B from B4 (*B4_to_R_for_B7*) to be placed into B7. The design logic for proper interleaving of events *L_to_B4_B* and *B7_to_AM* is also specified in ***TakeB4PutB7***.

To fulfill these specifications, our 3-state supervisor **TakeB4PutB7** designed for the $||_{SD}$ setting is given in Figure 42. Since we do not need to manually decide when to force *B4_to_R_for_B7*, we have neither added the logic for keeping track of other supervisors' behaviour and the plant model, nor forcing of event *B4_to_R_for_B7* in **TakeB4PutB7**.

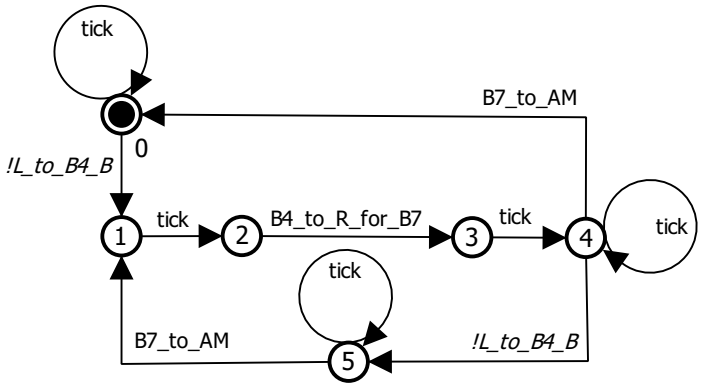


Figure 41: Supervisor ***TakeB4PutB7***

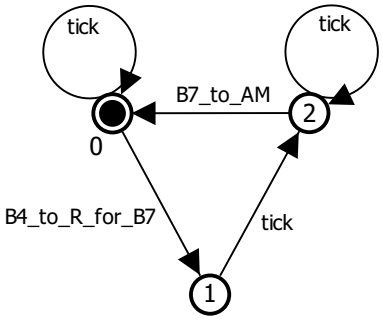


Figure 42: Supervisor **TakeB4PutB7**

As a result, our supervisor does not contain the waiting event $L\_to\_B4\_B$, and enables both *tick* and prohibitable event $B4\_to\_R\_for\_B7$ at state 0.

Once $B4\_to\_R\_for\_B7$ has occurred, **TakeB4PutB7** disables this event to avoid the above-mentioned blocking issue. This event is re-enabled after the occurrence of $B7\_to\_AM$, i.e. when part B returning from PM is moved from buffer B7 to AM. Also, we note that buffer supervisor **B7** already guarantees that events $B7\_to\_AM$ and $B4\_to\_R\_for\_B7$ always occur in different sampling periods. For this reason, we have not added an explicit state changing *tick* transition after $B7\_to\_AM$ in **TakeB4PutB7**, as present in supervisor ***TakeB4PutB7*** of the SD setting.

10.3.4 B6/B7 to AM to Exit Path

Now we will discuss the movement of parts from buffers B6 and B7 to finishing machine AM, from where finished parts finally exit the system. In order to resolve several concurrency issues along this path, supervisors ***ForceB6toAM***, ***ForceB7toAM***, ***ForceInitAM*** and ***AMChooser*** have been designed in the SD setting. These supervisors are heavily dependent upon one another and work closely together to make several decisions. We will analyze them one by one, and then discuss how their design and logic get simplified in the presence of our $||_{SD}$ operator.

Parts are moved from buffers B6 and B7 to AM using prohibitable events $B6\_to\_AM$ and $B7\_to\_AM$ respectively. Before accepting and processing any part, AM needs to initialize, which is indicated by prohibitable event $init\_AM$. This means the first thing that needs to determined and specified along this path is when to force these three prohibitable events by explicitly disabling *tick* in order to satisfy Point ii ($\Rightarrow$) of the SD controllability property. We must also make sure that when one modular supervisor is trying to force a prohibitable event, it must be enabled by all the concerned supervisors and possible in plant TDES **AM**, to keep the system controllable.

This is further complicated by the fact that parts might be waiting in both buffers B6 and B7 to go to AM for processing. This implies that another decision that needs to be made is to determine which buffer to service first. Ideally, these decisions should be reflected in the supervisor models without significant reuse of logic which seemed non-obvious, as stated in [43].

The solution devised in the SD setting to address the above-mentioned issues is to in-

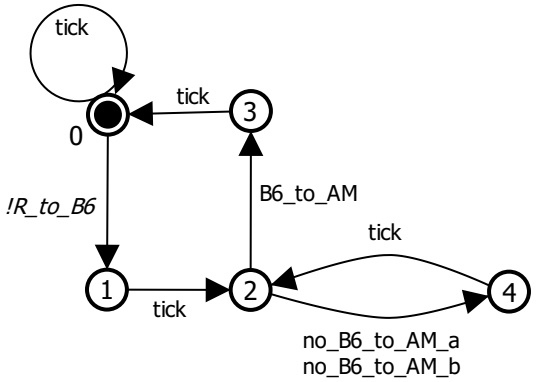


Figure 43: Supervisor ***ForceB6toAM***

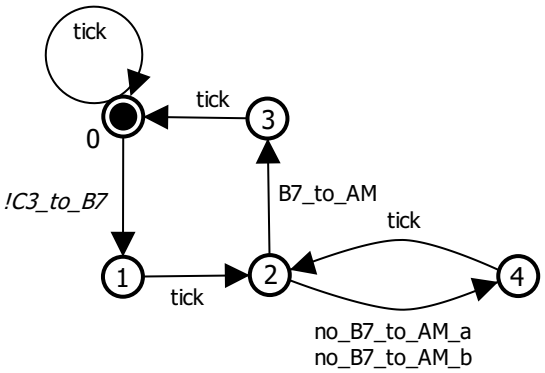


Figure 44: Supervisor ***ForceB7toAM***

troduce four new prohibitable expansion events that provide communication between the modular supervisors. These expansion events are *no_B6_to_AM_a*, *no_B6_to_AM_b*, *no_B7_to_AM_a* and *no_B7_to_AM_b*. They are introduced to the system by designing two additional plant TDES, ***AddNoB6toAM*** (Figure 25) and ***AddNoB7toAM*** (Figure 26).

ForceB6toAM and ForceB7toAM Supervisor ***ForceB6toAM***, shown in Figure 43, is designed in the SD setting to force the prohibitable event *B6_to_AM*. As *B6_to_AM* is under the control of supervisors **B6** and ***TakeB4PutB6***, ***ForceB6toAM*** replicates the design logic from **B6** by adding the watch event *R_to_B6*. ***ForceB6toAM*** waits for the occurrence of *R_to_B6*, signalling that there is a part in buffer B6 waiting to go to AM. It then forces *B6_to_AM* by explicitly disabling *tick* at state 2 in accordance with Point ii ($\Rightarrow$) of SD controllability.

However, if *B6_to_AM* is currently not possible in the plant TDES **AM** (Figure 22) or disabled by other supervisors, ***ForceInitAM*** (Figure 45) and ***AMChooser*** (Figure 47) that are also in control of *B6_to_AM*, then ***ForceB6toAM*** has no way of knowing this. In this case, system will become uncontrollable because *B6_to_AM* could not be forced and *tick* is already disabled by ***ForceB6toAM***.

This issue is handled by adding a loop of concurrent string "*no_B6_to_AM_a*/*no_B6_to_AM_b*−*tick*" at state 2 of supervisor ***ForceB6toAM***. The idea is to use expansion events, *no_B6_to_AM_a* or *no_B6_to_AM_b*, as alternative forcing options when *B6_to_AM* could not be forced. Since enablement information needs to be coordinated between three supervisors, that is why the designers have added two expansion events, '*a*' and '*b*'.

As *no_B6_to_AM_a* and *no_B6_to_AM_b* are meant to be used as substitute forcing options for *B6_to_AM*, they must only be enabled when it is not possible to force *B6_to_AM*. Also, it is important to make sure that only one of these three prohibitable events is possible in the system at a given time. Supervisors ***ForceInitAM*** and ***AMChooser*** contain the logic to fulfill these two requirements.

In order to force the prohibitable event *B7_to_AM*, supervisor ***ForceB7toAM***, shown in Figure 44, has been designed in the SD setting. As *B7_to_AM* is under the control of supervisors **B7** and ***TakeB4PutB7***, therefore ***ForceB7toAM*** duplicates the design logic from supervisor **B7** by including the watch event *C3_to_B7*. The rest of the logic of ***ForceB7toAM*** is same as ***ForceB6toAM***. Also, ***ForceB7toAM*** communicates with plant component **AM**, and supervisors ***ForceInitAM*** and ***AMChooser*** in a similar fashion as ***ForceB6toAM***.

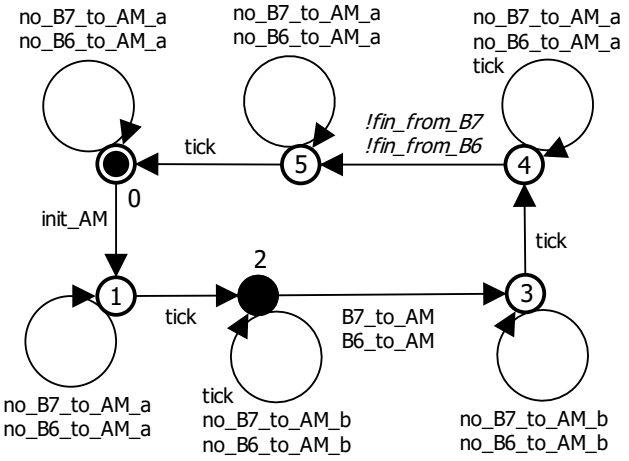


Figure 45: Supervisor ***ForceInitAM***

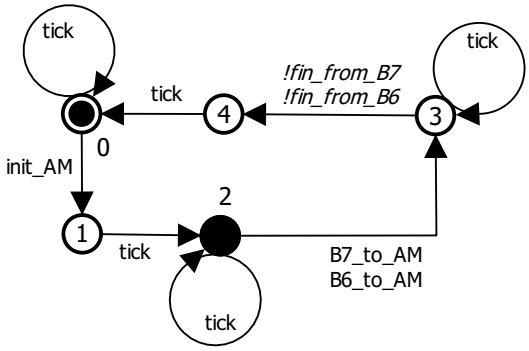


Figure 46: Supervisor **InitAM**

ForceInitAM Figure 45 shows supervisor ***ForceInitAM*** of the SD setting that is primarily responsible for deciding when to force prohibitable event *init_AM*. By disabling *tick* at state 0, it forces *init_AM* right away. After AM has processed the part received from buffer B6 or B7 (*B6_to_AM*/*B7_to_AM*), this supervisor then waits for the finished part to leave the system (*fin_from_B6*/*fin_from_B7*) before forcing another *init_AM*.

Another task performed by ***ForceInitAM*** is to make sure that '*a*' and '*b*' expansion events are never eligible in the system at the same time. It ensures this by enabling '*a*' events when *B6_to_AM*/*B7_to_AM* are not possible in the plant TDES **AM**. When they are possible in **AM**, ***ForceInitAM*** enables '*b*' events instead.

As supervisor ***AMChooser*** (Figure 47) ignores '*a*' events, this guarantees that '*a*' events will never be disabled when ***ForceInitAM*** needs them. As ***ForceInitAM*** never disables '*b*' events when *B6_to_AM*/*B7_to_AM* are possible in **AM**, this ensures that '*b*' events will never be disabled when ***AMChooser*** needs them.

By manually devising and explicitly incorporating this intricate logic in the supervisor models, designers made sure that the two supervisors do not interfere with each other with respect to these expansion events.

AMChooser The primary purpose of the supervisor ***AMChooser***, given in Figure 47, of the SD setting is to dictate the order in which AM accepts the parts from buffers B6 and B7, when both buffers have a part waiting to be processed by AM. If parts A and B arrive in both buffers (*R_to_B6*, *C3_to_B7*) in the same sampling period, then ***AMChooser*** forces AM to first take the part from B7 (*B7_to_AM*), and then from B6 (*B6_to_AM*). The reason is that there are more machines along the B7−PM−B7 path that should be kept busy. If only one buffer has a part waiting, then this part is taken by AM for processing.

In order to enforce this order for processing parts, ***AMChooser*** sometimes has to disable prohibitable events *B6_to_AM* and *B7_to_AM*. In such cases, it enables the appropriate '*b*' expansion events as a forcing alternative. This also guarantees that substitute forcing options of *B6_to_AM* and *no_B6_to_AM_b* are never enabled at the same time. The same is true for *B7_to_AM* and *no_B7_to_AM_b*.

Remodelling of Modular Supervisors for the $||_{SD}$ Setting In the SD setting, the only reason for introducing four prohibitable expansion events was to aid in communication between

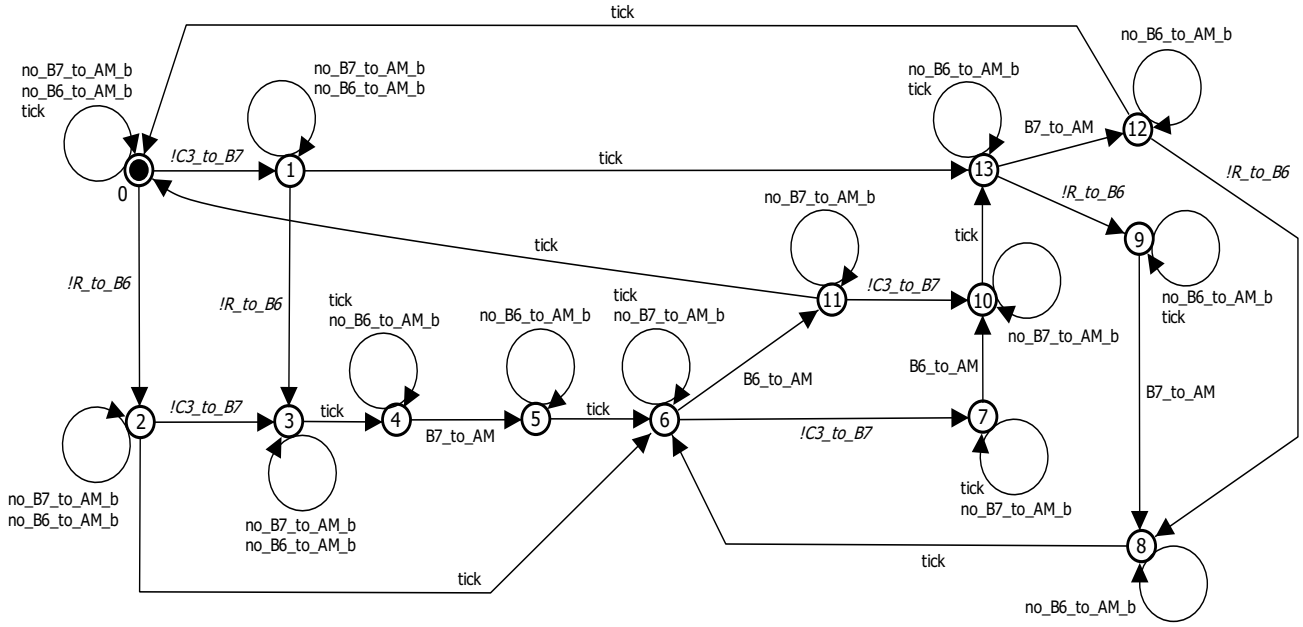


Figure 47: Supervisor ***AMChooser***

various modular supervisors in order to specify the explicit forcing decisions, while making sure that all desired properties and system specifications are satisfied. In the $||_{SD}$ setting, since modular supervisors are only concerned about their own behaviour, this eradicates the need to add expansion events to our $||_{SD}$ system. As a result, we can exclude all these expansion events and their corresponding plant models, ***AddNoB6toAM*** and ***AddNoB7toAM***, from our set of FMS plant components for the $||_{SD}$ setting.

In the SD setting, the primary purpose of designing supervisors ***ForceB6toAM*** and ***ForceB7toAM*** was to decide when to force prohibitable events *B6_to_AM* and *B7_to_AM* respectively. Since we have the $||_{SD}$ operator that automatically makes these forcing decisions for us, we will not include these two supervisors in our set of FMS modular supervisors. Please note that the order of occurrence of events enforced by these two supervisors is already present in the corresponding buffer supervisors, **B6** and **B7**.

Our $||_{SD}$ supervisor **InitAM**, shown in Figure 46, manages the initialization of AM and the movement of parts along the B6/B7−AM−exit path. Since we are not required to explicitly force prohibitable event *init_AM*, we have enabled both *tick* and *init_AM* at state 0. Also, plant TDES **AM** already makes sure that events *B6_to_AM*/*B7_to_AM* and *fin_from_B6*/*fin_from_B7* occur in different sampling periods. Therefore, we have not duplicated this logic in **InitAM**, as specified by ***ForceInitAM*** in the SD setting.

It is worth noting how simple this supervisor's design and logic has become in the absence of all expansion events and their transitions, as compared to its corresponding supervisor ***ForceInitAM*** of the SD setting. With no expansion events, we do not need to do any extra design effort to figure out how many other supervisors **InitAM** will communicate with, determine the number of expansion events required for communication, and then enable/disable all expansion events at the right time by keeping track of plant and other supervisors' behaviours.

Figure 48 illustrates our supervisor **AMChooser** of the $||_{SD}$ setting. Essentially, we have derived it from supervisor ***AMChooser*** of the SD setting after removing all expansion event

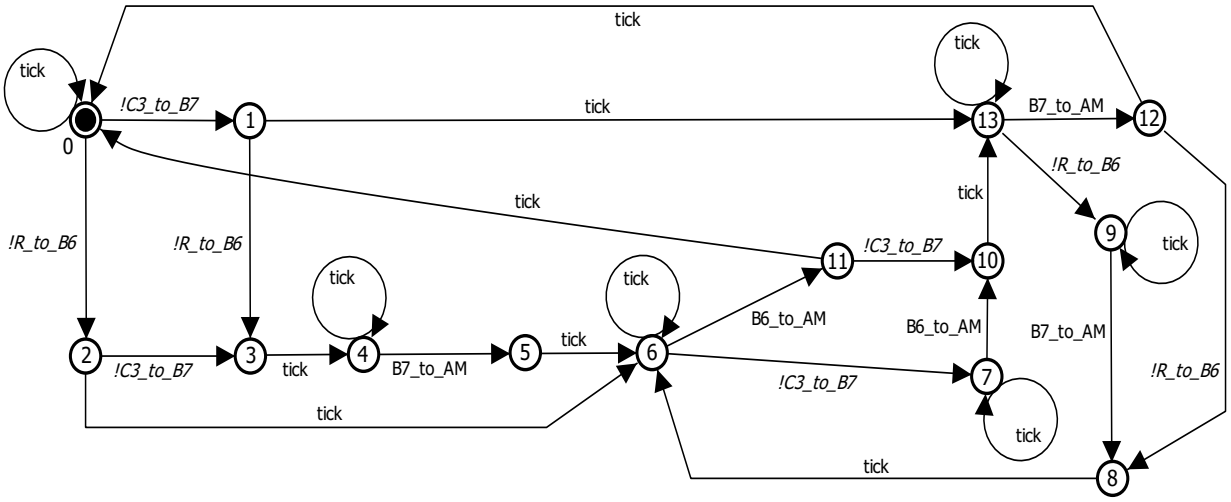


Figure 48: Supervisor **AMChooser**

transitions. **AMChooser** enforces the same order on AM for accepting parts from buffers B6 and B7 as ***AMChooser***. However, unlike ***AMChooser***, our supervisor **AMChooser** does not need to keep track of the plant model and other supervisors' behaviour to enable/disable appropriate expansion events at the right time. This results in greatly reducing the complexity of its design and logic.

10.4 Results and Discussion

Now it is time to present and discuss our results for the FMS example. Our complete results are shown in Table 2. In order to be clear and precise in our discussion, we will refer to the FMS TDES models designed in the SD setting as the "*SD system*", and the simplified FMS TDES models designed for our $||_{SD}$ setting as the "$||_{SD}$ *system*".

10.4.1 Theoretical TDES

By looking at the theoretical TDES models discussed in the previous section, we note that for the same FMS specifications, we are able to model our $||_{SD}$ system by designing fewer plant components and modular supervisors. This is because, unlike the SD system, we did not have to introduce and manage five prohibitable expansion events to aid in communication between different modular supervisors and make explicit forcing decisions in our $||_{SD}$ system. These results are summarized in the topmost section of Table 2.

10.4.2 Verification Results

In order to evaluate the performance of verifying our $||_{SD}$ properties, we implemented the SD synchronous product operator and our tweaked algorithms (presented in Section 9) as part of the DES research tool, DESpot [14]. Our code is based on the source code written by [42] that verifies the SD supervisory control methodology. The code uses the *BuDDy* package [30], a C++ library that implements standard BDD structures and operations.

In the rest of this discussion, we will refer to the algorithms implemented by [42] as the "*SD algorithms*". These SD algorithms check various properties in the SD setting (Section 3)

Table 2: FMS Example Results in the SD and $||_{SD}$ Setting

	SD System		**$||_{SD}$ System**	
Plant Components	10		7	
Modular Supervisors	15		12	
System Events	31		26	
Supervisor Properties	*Verification Time (seconds)*			
CS Deterministic	< 1		< 1	
Non-Selfloop ALF	< 1		< 1	
Closed-Loop System Properties	**SD Algorithms**	**$||_{SD}$ Algorithms**	**SD Algorithms**	**$||_{SD}$ Algorithms**
Nonblocking	< 1	< 1	< 1	< 1
Untimed Controllability	< 1	< 1	< 1	< 1
Timed Controllability	< 1	< 1	< 1	< 1
Proper Time Behaviour	< 1	< 1	< 1	< 1
Plant Completeness	< 1	< 1	< 1	< 1
ALF	2	2	1	1
SD Controllability & **S**-Singular Prohibitable Behaviour	30	26	False	7
Check All	32	28	False	8
State Size	***82,608***	***82,608***	***56,244***	***49,020***

and rely on the standard synchronous product operator to form the closed-loop system. On the other hand, the adapted algorithms that we have implemented for our $||_{SD}$ setting will be referred to as the "$||_{SD}$ *algorithms*". These $||_{SD}$ algorithms verify the $||_{SD}$ version of the properties (introduced in Section 4) and use our SD synchronous product operator to construct the closed-loop system. Please recall that, as mentioned in Section 9, although we are reusing some algorithms of the SD setting in our $||_{SD}$ setting without modifying their steps, still these algorithms are actually different in the two settings because of their way of constructing the closed-loop system in order to verify the desired properties.

In order to analyze and compare the FMS SD and $||_{SD}$ systems in detail, we decided to run SD and $||_{SD}$ algorithms on both systems. In other words, we not only verified our $||_{SD}$ system in our $||_{SD}$ setting by running our $||_{SD}$ algorithms, but we also tested it in the SD setting by running SD algorithms to find out which properties does it fail to satisfy (algorithms return "$False$") in the SD setting in the absence of our $||_{SD}$ operator. Similarly, besides running SD algorithms on the SD system, we also ran our $||_{SD}$ algorithms on the SD system to evaluate its performance in our $||_{SD}$ setting.

Table 2 shows our verification results for running various SD and $||_{SD}$ algorithms on the SD and $||_{SD}$ systems. These tests are performed on a machine running Windows 10 with 16GB of RAM and 2.6GHz Intel 6-core processor.

Supervisor Properties As the ultimate goal of designing TDES supervisors in the SD supervisory control theory is to generate the corresponding SD controller, we started by verifying two properties that play an important role in this translation process. These two properties are CS deterministic and non-selfloop ALF supervisors. As shown in Table 2, TDES supervisors of both SD and $||_{SD}$ systems passed each of these checks in less than 1 second.

Our next step is to verify various properties of the SD and $||_{SD}$ closed-loop systems by

running the SD and $||_{SD}$ algorithms. Before we analyze the results of these tests in detail, first we wish to highlight some important points about state sizes of the two systems that are constructed by the SD and $||_{SD}$ algorithms.

State Space Size of Closed-Loop System By looking at the state size of SD system given in Table 2, we observe that although SD and $||_{SD}$ algorithms use different synchronization operators to construct the closed-loop system, state size of the SD system is same in both cases, i.e. 82,608. The reason is that the SD system was originally designed for the SD setting, where designers are responsible for manually satisfying Point ii ($\Rightarrow$) of SD controllability. In this case, the $||_{SD}$ operator does not find any states where it has to disable *tick* in the presence of an enabled prohibitable event while constructing the closed-loop system. Consequently, the synchronization mechanism of the $||_{SD}$ operator essentially becomes equivalent to the standard synchronous product operator. This results in having the same state space for the SD system in both cases.

On the contrary, SD and $||_{SD}$ algorithms specify different state sizes for our $||_{SD}$ system. Specifically, our $||_{SD}$ closed-loop system constructed by the SD algorithms has 56,244 states, whereas $||_{SD}$ algorithms construct the state space of 49,020 states. This is because, keeping in view the synchronization mechanism of the $||_{SD}$ operator, we have enabled both *tick* and prohibitable events at various states of the TDES supervisors while modelling our $||_{SD}$ system. As the synchronous product operator is not capable of automatically disabling *tick* event in the presence of enabled prohibitable events while forming the closed-loop system, this is why SD algorithms construct a bigger state space for our $||_{SD}$ system than our $||_{SD}$ algorithms.

In essence, the key point to note is that state size for the FMS example has reduced from 82,608 states in the SD setting to 49,020 states in our $||_{SD}$ setting. This represents a reduction of 40% in the overall state space of the FMS closed-loop system.

This decrease in the state space is because of two reasons. First, in the presence of our $||_{SD}$ operator, less number of TDES plant and supervisor components are required to model the same system specifications. Second, as evident in Section 10.3, the size and logical design complexity of most of the modular supervisors of the SD system have been greatly reduced for our $||_{SD}$ system.

Next, we discuss our results of verifying various properties of the SD and $||_{SD}$ systems by running SD and $||_{SD}$ algorithms.

Closed-Loop System Properties As shown in Table 2, both SD and $||_{SD}$ systems satisfy the properties of nonblocking, untimed controllability, timed controllability, proper time behaviour and plant completeness in both settings. Each of these checks is completed individually in less than 1 second.

By running the ALF test, both SD and $||_{SD}$ systems are found to be ALF. However, this check was completed for the SD system in 2 seconds, whereas our $||_{SD}$ system took only 1 second to pass this test in the SD and $||_{SD}$ settings. Given the fact that we have reused ALF algorithm of the SD setting in our $||_{SD}$ setting without changing its steps, this difference in verification time can be attributed to different state sizes of the SD and $||_{SD}$ systems. As our $||_{SD}$ system has a reduced state space as compared to the SD system, this property gets verified more efficiently for our $||_{SD}$ system in both settings.

Currently, in DESpot, the check for **S**-singular prohibitable behaviour is implemented as part of the SD controllability test. Therefore, we verified these two properties together for the SD and $||_{SD}$ systems. Using SD algorithms, it took 30 seconds to verify these properties for the SD system. However, when we checked these properties of the SD system using our $||_{SD}$

algorithms, verification time dropped to 26 seconds. This is because our $||_{SD}$ algorithm tests the property of SD controllability with $||_{SD}$, which does not include an explicit check for Point ii ($\Rightarrow$) of SD controllability. This saves time by performing one less check in the presence of our $||_{SD}$ operator as compared to the SD setting.

When we tried to verify the property of SD controllability for our $||_{SD}$ system using SD algorithms, the algorithms returned $False$. The reason is quite obvious. Since we have not explicitly disabled *tick* while enabling prohibitable events at various states of the modular $||_{SD}$ supervisors, our $||_{SD}$ system fails to satisfy the constraint imposed by Point ii ($\Rightarrow$) of SD controllability in the SD setting.

In order to test the properties of SD controllability with $||_{SD}$ and **S**-singular prohibitable behaviour with $||_{SD}$ for the $||_{SD}$ system, we ran our corresponding $||_{SD}$ algorithms. Our $||_{SD}$ system not only passed these checks but the verification process was completed within 7 seconds, as opposed to the SD system that took 30 seconds to pass these tests in the SD setting.

This indicates that for the FMS example, we have verified these two properties of our $||_{SD}$ system 4x faster as compared to the SD system. In other words, we recorded a 76.6% reduction in verification time and more than 300% increase in performance in our $||_{SD}$ setting as compared to the SD setting with respect to the verification of these two properties.

This significant reduction in verification time is primarily due to the smaller state size of our $||_{SD}$ system as compared to the SD system, that we are able to achieve due to the automatic *tick* disablement mechanism of the $||_{SD}$ operator. Moreover, our $||_{SD}$ algorithms check one less condition as part of the SD controllability with $||_{SD}$ property in the presence of the $||_{SD}$ operator.

In DESpot, we also have an option to check the desired system properties all at once (Check All). In the SD setting, it took 32 seconds to verify all properties of the SD system, while our $||_{SD}$ system passed all tests in 8 seconds in our $||_{SD}$ setting. On the whole, our results demonstrate a time reduction of 75% and performance increase of exactly 300% in our $||_{SD}$ setting compared to the SD setting. This shows that we are able to do complete verification of our FMS $||_{SD}$ system 4x faster than its corresponding FMS SD system.

10.4.3 Miscellaneous Discussion

We will close this section by discussing two important points.

1. It is worth-mentioning that, apparently, our $||_{SD}$ operator does more work than the synchronous product while synchronizing plant and supervisor models to construct the closed-loop system. This is because our $||_{SD}$ operator is required to figure out whether to enable/disable *tick* event at every state of the closed-loop system using a more complex logic than synchronous product.

   Nevertheless, we are still able to notice a visible decline in the overall verification time of FMS example in our $||_{SD}$ setting as compared to the SD setting. This suggests that the slightly complicated synchronization logic of our $||_{SD}$ operator has not adversely affected the overall performance of our $||_{SD}$ algorithms.

2. We would like to point out that we ran SD algorithms on our $||_{SD}$ system because we wanted to inspect that other than manually satisfying Point ii ($\Rightarrow$) of SD controllability, is there any other reason/significance which necessitates the design of a complicated SD system instead of modelling a simpler $||_{SD}$ system? As shown in Table 2, our results indicate that

our $||_{SD}$ system satisfies all properties in the SD setting except for Point ii ($\Rightarrow$) of SD controllability.

This makes it evident that the design and verification of the FMS example got a lot more complicated in the SD setting just because this one property needed to be satisfied manually. This clearly shows the significance of our $||_{SD}$ operator that has made the design and verification process of our FMS $||_{SD}$ system relatively simple, easy and efficient, by providing a guarantee to automatically satisfy this intricate property in our $||_{SD}$ setting.

11 Conclusions and Future Work

This section presents our conclusions and gives some directions for further research related to this study.

11.1 Conclusions

In this report, we present an approach to automate the mechanism of disabling the *tick* event and force eligible prohibitable events in the sampled-data (SD) supervisory control framework. The SD supervisory control theory [42, 29] focuses on the implementation of timed discrete event system (TDES) supervisors as SD controllers, and addresses the related implementation and concurrency issues in a formal and well-defined way.

Our study begins by devising a new synchronization operator, called the *SD synchronous product* (or "$||_{SD}$ *operator*," for short), that provides a novel way of constructing closed-loop system in the SD supervisory control framework. The $||_{SD}$ operator is smart enough to automatically disable *tick* event in the closed-loop system, if both *tick* and a prohibitable event is possible in the plant TDES and enabled by all modular TDES supervisors. As a result, designers are no longer required to manually keep track of the enablement/disablement of *tick* and prohibitable events in the plant and supervisor models. This includes explicitly deciding when to force a prohibitable event, and then incorporating this logic into various supervisor models while designing the modular supervisors by hand.

As we have formulated a new way of constructing the closed-loop system, we adapt the existing definitions of various TDES and SD properties from the SD supervisory control setting (or "*SD setting*," to be concise) to make them compatible with our $||_{SD}$ operator. Most importantly, we modify the definition of SD controllability and eliminate the explicit check of the forward implication ($\Rightarrow$) of Point ii. This is because the synchronization logic of our $||_{SD}$ operator ensures that this condition will always be satisfied at every state of the closed-loop system. In fact, guaranteeing the automatic satisfaction of this intricate property and getting rid of this explicit check is the primary purpose of devising the $||_{SD}$ operator.

In order to verify our SD synchronous product setting (or "$||_{SD}$ *setting,*" from now on), we formally prove all existing controllability and nonblocking verification results of the SD setting for our $||_{SD}$ setting. Rather than proving all these results from scratch, we opt for establishing logical equivalence between the two settings. Essentially, we prove that the SD and $||_{SD}$ settings are equivalent with respect to their closed and marked languages, TDES and SD properties, and the SD controllers that are obtained by translating concurrent string deterministic supervisors of the two settings.

By making use of this formal equivalence, we derive and conclude the desired controllability and nonblocking verification results from the SD setting, but now for our $||_{SD}$ setting. First, we discuss the construction of a TDES supervisory control map that captures the enablement and

forcing behaviour of the SD controller **C** translated from TDES supervisor **S** in our $||_{SD}$ setting, and prove its desired properties. Then, we formally prove that the behaviour of TDES plant **G** under the control of **C** is the same as the behaviour of **G** under the supervision of **S**, given that the theoretical $||_{SD}$ system satisfies the properties that we have defined in our $||_{SD}$ setting. We also show that if the theoretical $||_{SD}$ system is nonblocking, then the implemented system will retain this property. This is proven to be true even if only a single concurrent string, out of multiple concurrent strings with the same occurrence image possible in theoretical model at a given sampled state, is actually possible in the physical system.

This is followed by a presentation of predicate-based algorithms that we have adapted from [42] to symbolically verify various TDES and SD properties in our $||_{SD}$ setting. We implement these tweaked algorithms as part of the DES research tool, DESpot [14].

We demonstrate the application and utilization of our $||_{SD}$ operator and $||_{SD}$ setting by discussing an example of a Flexible Manufacturing System (FMS) from [42, 43]. By comparing the modular TDES supervisors developed for FMS in the SD and $||_{SD}$ settings, we show that the design logic of supervisors has greatly simplified in our $||_{SD}$ setting. In the presence of our $||_{SD}$ operator, modular supervisors are only concerned about their own behaviour and do not need to communicate with each other to collectively make the *tick* disablement and event forcing decisions. This reduces the complexity of the TDES modelling process and improves the ease of designing SD controllable supervisors by hand, as designers are no longer required to manually satisfy Point ii ($\Rightarrow$) of the SD controllability definition at every state of the closed-loop system.

We further note that this simplification in the logical design of TDES supervisors also reduces the state size of individual modular supervisors, hence the overall system. We observe a decrease of 40% in the overall state space of the FMS closed-loop system designed in our $||_{SD}$ setting as compared to the FMS system developed in the SD setting. Due to this reduced state space and one less verification check of Point ii ($\Rightarrow$), we are able to do the complete verification of our FMS $||_{SD}$ system 4x faster than the corresponding FMS SD system.

Overall, our FMS example results indicate that the development of the $||_{SD}$ operator and the $||_{SD}$ setting has simplified the formal design and verification process of TDES control systems. Our approach makes the SD supervisory control methodology more accessible to software and hardware designers and practitioners. This should increase the adoption of SD supervisory control methodology in particular, and formal methods in general, in the industry.

11.2 Future Work

We have verified our $||_{SD}$ approach by applying it to the small example of a Flexible Manufacturing System. We noted improvement in the ease of manually designing SD controllable TDES supervisors, reduction in the state space, and faster verification time of the overall system. In future, it would be interesting to check the efficacy of our $||_{SD}$ setting and reaffirm our results by applying it to TDES with larger state spaces.

In this study, we have implemented our $||_{SD}$ operator and related properties by tweaking the predicate-based algorithms implemented by [42] in DESpot [14]. Wang's implemented algorithms perform a monolithic check to verify various TDES and SD properties of the closed-loop system. [19] has implemented a modular method to verify these properties in DESpot. It would be useful to implement our $||_{SD}$ operator and $||_{SD}$ properties in Hamid's modular verification method and gauge the performance gain.

It would also be interesting to extend our $||_{SD}$ approach to fault-tolerant supervisory control [32] and hierarchical interface-based supervisory control [28] in future.

References

[1] J. Arinez, B. Benhabib, K. Smith, and B. Brandin. Design of a PLC-Based Supervisory Control System for a Manufacturing Workcell. In *Proceedings of the Canadian High Technology Show and Conference*, Canada, 1993.

[2] D. S. Arnon. A Bibliography of Quantifier Elimination for Real Closed Fields. *Journal of Symbolic Computation*, 5(1-2):267–274, Feb. 1988.

[3] S. Balemi. Input/Output Discrete Event Processes and Communication Delays. *Discrete Event Dynamic Systems*, 4(1):41–85, Feb. 1994.

[4] W. Bolton. *Programmable Logic Controllers*. Elsevier, 6th edition, 2015.

[5] B. A. Brandin. *Real-Time Supervisory Control of Automated Manufacturing Systems*. Ph.D. Thesis, Department of Computer and Electrical Engineering, University of Toronto, Toronto, ON, Canada, 1993.

[6] B. A. Brandin. The Real-Time Supervisory Control of an Experimental Manufacturing Cell. *IEEE Transactions on Robotics and Automation*, 12(1):1–14, Feb. 1996.

[7] B. A. Brandin and W. M. Wonham. Supervisory Control of Timed Discrete-Event Systems. *IEEE Transactions on Automatic Control*, 39(2):329–342, Feb. 1994.

[8] S. Brown and Z. Vranesic. *Fundamentals of Digital Logic with Verilog Design*. McGraw-Hill Education, New York, 3rd edition, Feb. 2013.

[9] Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[10] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys (CSUR)*, 24:293–318, June 1992.

[11] M. Cantarelli. Control System Design Using Supervisory Control Theory: From Theory to Implementation. Master's Thesis, University of Cagliari, Italy, Dec. 2006.

[12] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer US, New York, NY, 2nd edition, 2008.

[13] M. H. de Queiroz and J. E. R. Cury. Synthesis and Implementation of Local Modular Supervisory Control for a Manufacturing Cell. In *Proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02)*, pages 377–382, Spain, Oct. 2002.

[14] DESpot. www.cas.mcmaster.ca/~leduc/DESpot.html, 2023.

[15] M. Fabian and A. Hellgren. PLC-Based Implementation of Supervisory Control for Discrete Event Systems. In *Proceedings of the 37th IEEE Conference on Decision and Control*, volume 3, pages 3305–3310, USA, Feb. 1998.

[16] K. Fouquet and J. Provost. A Signal-Interpreted Approach to the Supervisory Control Theory Problem. *IFAC-PapersOnLine*, 50(1):12351–12358, July 2017.

[17] G. Gelen, M. Uzam, and R. Dalci. The Concept of Postponed Event in Timed Discrete Event Systems and its PLC Implementation. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC'10)*, pages 2753–2759, Turkey, Oct. 2010.

[18] D. Gouyon, J.-F. Pétin, and A. Gouin. A Pragmatic Approach for Modular Control Synthesis and Implementation. *International Journal of Production Research*, 42(14):2839–2858, July 2004.

[19] A. Hamid. Implementation of Sampled-Data Supervisory Control. Master's Thesis, Department of Computing and Software, McMaster University, Hamilton, ON, Canada, June 2014.

[20] İ. T. Hasdemir, S. Kurtulan, and L. Gören. An Implementation Methodology for Supervisory Control Theory. *The International Journal of Advanced Manufacturing Technology*, 36(3):373–385, Mar. 2008.

[21] R. C. Hill. *Modular Verification and Supervisory Controller Design for Discrete-Event Systems Using Abstraction and Incremental Construction.* Ph.D. Thesis, Department of Mechanical Engineering, University of Michigan, 2008.

[22] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Menlo Park, 1979.

[23] L. D. James, C. A. Teixeira, and A. B. Leal. Formal Design and Implementation of Supervisory Controller for a Didactic Manufacturing Cell. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT'19)*, pages 935–940, Australia, Feb. 2019.

[24] T. Jiao and Y. Gan. On PLC Implementation of Decentralized and Hierarchical Supervisory Control of Discrete-Event System. In *Proceedings of the 15th International Conference on Computer Modelling and Simulation, UKSim*, pages 413–418, UK, Apr. 2013.

[25] S. C. Lauzon, J. K. Mills, and B. Benhabib. An Implementation Methodology for the Supervisory Control of Flexible Manufacturing Workcells. *Journal of Manufacturing Systems*, 16(2):91–101, Jan. 1997.

[26] A. Leal, D. L. L. da Cruz, and M. Hounsell. PLC-Based Implementation of Local Modular Supervisory Control for Manufacturing Systems. In *Manufacturing System*, pages 159–182. May 2012.

[27] R. J. Leduc. PLC Implementation of a DES Supervisor for a Manufacturing Testbed: An Implementation Perspective. Master's Thesis, Department of Computer and Electrical Engineering, University of Toronto, Toronto, ON, Canada, 1996.

[28] R. J. Leduc. *Hierarchical Interface-Based Supervisory Control.* Ph.D. Thesis, Department of Computer and Electrical Engineering, University of Toronto, Toronto, ON, Canada, 2001.

[29] R. J. Leduc, Y. Wang, and F. Ahmed. Sampled-Data Supervisory Control. *Discrete Event Dynamic Systems*, 24(4):541–579, Dec. 2014.

[30] J. Lind-Nielsen. BuDDy: Binary Decision Diagram Package. IT-University of Copenhagen (ITU), 2002.

[31] C. Ma. *Nonblocking Supervisory Control of State Tree Structures*. Ph.D. Thesis, Department of Computer and Electrical Engineering, University of Toronto, Toronto, ON, Canada, 2004.

[32] A. Mulahuwaish. *Fault-Tolerant Supervisory Control*. Ph.D. Thesis, Department of Computing and Software, McMaster University, Hamilton, ON, Canada, May 2019.

[33] L. Prenzel and J. Provost. PLC Implementation of Symbolic, Modular Supervisory Controllers. *IFAC-PapersOnLine*, 51(7):304–309, Jan. 2018.

[34] P. J. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):81–98, Jan. 1989.

[35] F. F. H. Reijnen, T. R. Erens, J. M. van de Mortel-Fronczak, and J. E. Rooda. Supervisory Controller Synthesis and Implementation for Safety PLCs. *Discrete Event Dynamic Systems*, 32(1):115–141, Mar. 2022.

[36] R. Song. Symbolic Hierarchical Interface-based Supervisory Control. Master's Thesis, Department of Computing and Software, McMaster University, Hamilton, ON, Canada, Mar. 2006.

[37] R. Szpak, M. H. de Queiroz, and J. E. R. Cury. Synthesis and Implementation of Supervisory Control for Manufacturing Systems Under Processing Uncertainities and Time Constraints. 53(4):229–234, 2020.

[38] M. Uzam. A General Technique for the PLC-Based Implementation of RW Supervisors with Time Delay Functions. *The International Journal of Advanced Manufacturing Technology*, 62(5):687–704, Sept. 2012.

[39] M. Uzam, G. Gelen, and R. Dalci. A New Approach for the Ladder Logic Implementation of Ramadge-Wonham Supervisors. In *Proceedings of the 22nd International Symposium on Information, Communication and Automation Technologies (ICAT'09)*, pages 113–119, Sarajevo, Bosnia and Herzegovina, Oct. 2009.

[40] A. Vieira, E. Santos, M. Hering de Queiroz, A. Leal, A. Neto, and J. Cury. A Method for PLC Implementation of Supervisory Control of Discrete Event Systems. *IEEE Transactions on Control Systems Technology*, 25(1):175–191, Jan. 2017.

[41] A. D. Vieira, J. E. R. Cury, and M. H. de Queiroz. A Model for PLC Implementation of Supervisory Control of Discrete Event Systems. In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, pages 225–232, Prague, Czech Republic, Sept. 2006.

[42] Y. Wang. Sampled-Data Supervisory Control. Master's Thesis, Department of Computing and Software, McMaster University, Hamilton, ON, Canada, Jan. 2009.

[43] Y. Wang and R. J. Leduc. Sampled-Data Controller Implementation. *International Journal of Control*, 85(9):1343–1360, Sept. 2012.

[44] K. C. Wong and W. M. Wonham. Hierarchical Control of Timed Discrete-Event Systems. *Discrete Event Dynamic Systems*, 6(3):275–306, July 1996.

[45] W. M. Wonham and K. Cai. *Supervisory Control of Discrete-Event Systems*. Springer International Publishing, 2018.

[46] W. M. Wonham and P. J. Ramadge. On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization*, 25(3):637–659, May 1987.

[47] J. Zaytoon and V. Carre-Meneatrier. Synthesis of Control Implementation for Discrete Manufacturing Systems. *International Journal of Production Research*, 39(2):329–345, Jan. 2001.

[48] J. Zaytoon and B. Riera. Synthesis and Implementation of Logic Controllers – A Review. *Annual Reviews in Control*, 43:152–168, Jan. 2017.

A Miscellaneous Definitions

A.1 Equivalence Relation

Definition A.1. Let X be a nonempty set. Let $E \subseteq X \times X$ be a binary relation on X. The relation E is an *equivalence relation* on X if:

1. $(\forall x \in X)\, xEx$ (E is *reflexive*)
2. $(\forall x, x' \in X)\, xEx' \Rightarrow x'Ex$ (E is *symmetric*)
3. $(\forall x, x', x'' \in X)\, xEx' \;\&\; x'Ex'' \Rightarrow xEx''$ (E is *transitive*)

In this definition, we use the standard infix notation xEx' to represent the ordered pair $(x, x') \in E$. Instead of xEx', we shall often write $x \equiv x' (\mathrm{mod}\ E)$.

A.2 Product Operator

Definition A.2. Let $\mathbf{G}_1 = (Q_1, \Sigma, \delta_1, q_{o,1}, Q_{m,1})$ and $\mathbf{G}_2 = (Q_2, \Sigma, \delta_2, q_{o,2}, Q_{m,2})$ be two DES defined over the same event set Σ. The *product* of two DES is defined as:

$$\mathbf{G}_1 \times \mathbf{G}_2 := (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{o,1}, q_{o,2}), Q_{m,1} \times Q_{m,2})$$

where $\delta_1 \times \delta_2 : Q_1 \times Q_2 \times \Sigma \to Q_1 \times Q_2$ is given by $(\delta_1 \times \delta_2)\,((q_1, q_2), \sigma) := (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$ whenever $\delta_1(q_1, \sigma)!$ and $\delta_2(q_2, \sigma)!$.

By this definition, we have:

$$L(\mathbf{G}_1 \times \mathbf{G}_2) = L(\mathbf{G}_1) \cap L(\mathbf{G}_2) \quad \text{and} \quad L_m(\mathbf{G}_1 \times \mathbf{G}_2) = L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2)$$

A.3 Meet Operator

Definition A.3. The *meet* of two DES $\mathbf{G}_1$ and $\mathbf{G}_2$, represented as $\mathbf{G} = \mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)$, is the reachable sub-DES of the product DES $\mathbf{G}_1 \times \mathbf{G}_2$.

A.4 Selfloop Operation

Definition A.4. Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$ be a DES defined over Σ. Let Σ' be another set of events such that $\Sigma \cap \Sigma' = \emptyset$. The *selfloop* operation on $\mathbf{G}$ is used to generate a new DES $\mathbf{G}'$ by selflooping each event in Σ' at every state of $\mathbf{G}$. Formally, this is expressed as:

$$\mathbf{G}' = \mathbf{selfloop}(\mathbf{G}, \Sigma') = (Q, \Sigma \cup \Sigma', \delta', q_o, Q_m)$$

where $\delta' : Q \times (\Sigma \cup \Sigma') \to Q$ is a partial function defined as:

$$\delta'(q, \sigma) := \begin{cases} \delta(q, \sigma) & \sigma \in \Sigma,\ \delta(q, \sigma)! \\ q & \sigma \in \Sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

A.5 Bijective Function

Definition A.5. A function $f : A \to B$ is:

1. **injective** if for every $x, y \in A, x \neq y \Rightarrow f(x) \neq f(y)$;
2. **surjective** if for every $b \in B$ there is an $a \in A$ with $f(a) = b$;
3. **bijective** if f is both injective and surjective.

B Symbolic Verification

In this appendix, we provide some content related to symbolic verification from [42] for the sake of completeness. Specifically, we restate the definition for symbolic representation of transitions in the SD supervisory control theory that we have referred to in Section 9. Also, we present some predicate-based algorithms that we are reusing to verify some properties in our $||_{SD}$ setting. Please see [42] for complete details.

B.1 Symbolic Representation of Transitions

Let $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m) = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_n$ be the product TDES of component TDES $\mathbf{G}_i = (Q_i, \Sigma_i, \delta_i, q_{o,i}, Q_{m,i})$ for $i = 1, 2, \ldots, n$. For any state $q \in Q$, we have $q = (q_1, q_2, \ldots, q_n)$ where $q_i \in Q_i$.

Definition B.1. For $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_n$, let $\sigma \in \Sigma$. A *transition predicate* $N_\sigma : Q \times Q \to \{T, F\}$ identifies all the transitions for σ in $\mathbf{G}$ and is defined as follows:

$$(\forall q, q' \in Q)\, N_\sigma(q, q') := \begin{cases} T & \text{if } \delta(q,\sigma)! \;\&\; \delta(q,\sigma) = q' \\ F & \text{otherwise} \end{cases}$$

In order to distinguish between source states and destination states, two different vectors of state variables are needed, as defined below.

Definition B.2. For $\mathbf{G} = \mathbf{G}_1 \times \mathbf{G}_2 \times \ldots \times \mathbf{G}_n$, let $i = 1, 2, \ldots, n$. For each $\mathbf{G}_i$, we have the *normal state variable* v_i (source state) and the *prime state variable* v_i' (destination state), both with domain Q_i. For $\mathbf{G}$, we have the *normal state variable vector* $\mathbf{v} = [v_1, v_2, \ldots, v_n]$ and the *prime state variable vector* $\mathbf{v}' = [v_1', v_2', \ldots, v_n']$.

For each $\sigma \in \Sigma$, we can write the transition predicate for σ, N_σ, as below. Essentially, if we set $\mathbf{v} = q$ and $\mathbf{v}' = q'$ such that $\delta(q, \sigma) = q'$, then $N_\sigma(\mathbf{v}, \mathbf{v}')$ will return T.

Definition B.3. We use the *transition tuple* $(\mathbf{v}_\sigma, \mathbf{v}'_\sigma, N_\sigma)$ to represent the transition on σ, where $\mathbf{v}_\sigma := \{v_i \in \mathbf{v} \,|\, \sigma \in \Sigma_i\}, \mathbf{v}'_\sigma := \{v_i' \in \mathbf{v}' \,|\, \sigma \in \Sigma_i\}$, and

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{1 \le i \le n \,|\, \sigma \in \Sigma_i\}} \left(\bigvee_{\{q_i, q_i' \in Q_i \,|\, \delta_i(q_i, \sigma) = q_i'\}} (v_i = q_i) \wedge (v_i' = q_i') \right)$$

B.2 Symbolic Verification of $||_{SD}$ Properties

In this section, we present some predicate-based algorithms from [42] that can be reused in our $||_{SD}$ setting without altering the actual algorithm steps. It is notable that we will only mention the underlying modifications while using these algorithms to verify properties in our $||_{SD}$ setting. Please see complete description of these algorithms in [42].

B.2.1 Nonblocking

Algorithm 12[13] checks the property of nonblocking on the input TDES $\mathbf{G}$. As this property has not changed for our $||_{SD}$ setting, we can use this algorithm from the SD setting to check our $||_{SD}$ system.

[13]Line 2 of this algorithm is different from the corresponding line of Algorithm 6.5 given in [42]. This is due to the incorrect number of parameters specified in the original algorithm. We have fixed this error in this version.

Algorithm 12 Nonblocking($\mathbf{G}$)

1: $P_{reach} \leftarrow R(\mathbf{G}, true)$
2: $P_{coreach} \leftarrow \mathcal{CR}(\mathbf{G}, P_m, \Sigma, P_{reach})$
3: **if** $(P_{reach} \wedge \neg P_{coreach} \not\equiv false)$ **then**
4: **return** $False$
5: **end if**
6: **return** $True$

Algorithm 13 ALF($\mathbf{G}$)

1: $P_{chk} \leftarrow R(\mathbf{G}, true)$
2: $P_{tmp} \leftarrow false$
3: **for** $(q \models P_{chk})$ **do**
4: $P_{visit} \leftarrow \left(\bigvee_{\sigma \in \Sigma_{act}} \hat{\delta}(pr(\{q\}), \sigma) \right) \wedge P_{chk}$
5: $overlap \leftarrow False$
6: $P_{next} \leftarrow P_{visit}$
7: **repeat**
8: $P_{next} \leftarrow \left(\bigvee_{\sigma \in \Sigma_{act}} \hat{\delta}(P_{next}, \sigma) \right) \wedge P_{chk}$
9: $P_{tmp} \leftarrow P_{visit}$
10: **if** $(P_{visit} \wedge P_{next} \not\equiv false)$ **then**
11: $overlap \leftarrow True$
12: **end if**
13: $P_{visit} \leftarrow P_{visit} \vee P_{next}$
14: **if** $(q \models P_{visit})$ **then**
15: **return** $False$
16: **end if**
17: **until** $(P_{visit} \equiv P_{tmp})$
18: $P_{chk} \leftarrow P_{chk} - pr(\{q\})$
19: **if** $(\neg overlap)$ **then**
20: $P_{chk} \leftarrow P_{chk} - P_{visit}$
21: **end if**
22: **end for**
23: **return** $True$

As we are interested in making sure that our closed-loop system is nonblocking, we will provide our closed-loop system of the $||_{SD}$ setting as an input to this algorithm, which is different from the closed-loop system used in the SD setting. Also, R (Algorithm 5) and $\mathcal{CR}$ (Algorithm 6) will be computed using our function definitions, as explained in Section 9.3.2.

B.2.2 Activity Loop Free

As the original activity-loop-free (ALF) property remains unchanged in our $||_{SD}$ setting, we can use Algorithm 13 of the SD setting without changing its steps.

The only difference is the TDES $\mathbf{G}$ that we provide as an input to this algorithm. In our $||_{SD}$ setting, parameter $\mathbf{G}$ will represent our closed-loop system that we want to make sure is

Algorithm 14 ProperTimeBehaviour($\mathbf{G}$)

1: $P_1 \leftarrow \bigvee_{\sigma \in \Sigma_u \cup \{\tau\}} \hat{\delta}_{\mathbf{G}}^{-1}(true, \sigma)$
2: $P_2 \leftarrow R(\mathbf{G}, true)$
3: **if** $(P_2 - P_1 \not\equiv false)$ **then**
4: **return** $False$
5: **end if**
6: **return** $True$

ALF. As our input $\mathbf{G}$ is constructed in a different way than the closed-loop system of the SD setting, Algorithm 13 will use our R (Algorithm 5) at **line 1**, and our Definition 9.11 of the function $\hat{\delta}$ at **lines 4** and **8** to compute the required predicates while verifying ALF property in our $||_{SD}$ setting. Please recall that $\hat{\delta}$ relies on N_σ, and the definition of N_σ in our $||_{SD}$ setting (Definition 9.9) is different from the SD setting (Definition $B.3$).

B.2.3 Proper Time Behaviour

The property of proper time behaviour is defined only in terms of TDES plant $\mathbf{G}$. As $\mathbf{G}$ is same in the SD and $||_{SD}$ settings, Algorithm 14 remains unmodified with respect to its steps, underlying functions, and input $\mathbf{G}$.

B.2.4 S-Singular Prohibitable Behaviour with $||_{SD}$

The property of $\mathbf{S}$-singular prohibitable behaviour with $||_{SD}$ is verified at **lines 12-16** of Algorithm 16. Please note that we have redefined the meaning of variables Σ_{poss}, B_{conc}, and node b in our $||_{SD}$ setting, as discussed in Section 9.5.3. Please refer to Section B.3.2 for further details on Algorithm 16.

B.3 Symbolic Verification of SD Controllability with $||_{SD}$

In this section, we present algorithms from [42] that can be used to verify Points ii and Point iii of our SD controllability with $||_{SD}$ property. Please recall that these two points correspond to Points iii and iv of the SD controllability (Definition 3.7) respectively in the SD setting.

It is worth-mentioning that these algorithms can be used in our $||_{SD}$ setting without modifying their steps. However, the input parameters and the underlying definitions for some variables and functions used by these algorithms have changed in our $||_{SD}$ setting. Therefore, it is our implicit assumption that while verifying properties in our $||_{SD}$ setting, these algorithms use the variable and function definitions as specified in Section 9. In particular, from Section 9, we have used functions, $\hat{\delta}$ (Definition 9.11) and $\hat{\delta}_{\mathbf{G}}$ (Definition 9.14), as well as R (Algorithm 5) to calculate P_{reach}. Please refer to Section 9.5.3 for the definition of variables used in the following algorithms.

B.3.1 Point ii.1

In order to verify Point ii.1 of SD controllability with $||_{SD}$, Algorithm 15 analyzes the concurrent behaviour of sampled state q_{ss} represented by predicate P_{ss}. Starting at q_{ss}, it builds a

Algorithm 15 AnalyzeSampledState($\mathbf{G}, \mathbf{S}, P_{SF}, Z_{SP}, P_{reach}, P_{ss}, pNerFail$)

1: $B_{map} \leftarrow \{(0, P_{ss})\}$
2: $B_{conc} \leftarrow \emptyset$
3: $B_p \leftarrow \{0\}$
4: $nextLabel \leftarrow 1$
5: $\text{Occu}_B \leftarrow \{(0, \emptyset)\}$
6: **while** $(B_p \neq \emptyset)$ **do**
7: $\quad b \leftarrow \text{Pop}(B_p)$
8: $\quad P_q \leftarrow B_{map}(b)$
9: $\quad \Sigma_{poss} \leftarrow \emptyset$
10: $\quad \Sigma_{\mathbf{G}poss} \leftarrow \emptyset$
11: $\quad$ **for all** $(\sigma \in \Sigma)$ **do**
12: $\quad\quad$ **if** $(\hat{\delta}(P_q, \sigma) \not\equiv false)$ **then**
13: $\quad\quad\quad \Sigma_{poss} \leftarrow \Sigma_{poss} \cup \{\sigma\}$
14: $\quad\quad$ **end if**
15: $\quad\quad$ **if** $(\hat{\delta}_{\mathbf{G}}(P_q, \sigma) \not\equiv false)$ **then**
16: $\quad\quad\quad \Sigma_{\mathbf{G}poss} \leftarrow \Sigma_{\mathbf{G}poss} \cup (\{\sigma\} \cap \Sigma_{hib})$
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: $\quad$ **if** $(P_q \equiv P_{ss})$ **then**
20: $\quad\quad \Sigma_{Elig} \leftarrow \Sigma_{poss} \cap \Sigma_{hib}$
21: $\quad$ **end if**
22: $\quad$ **if** $((\Sigma_{poss} \cup \text{Occu}_B(b)) \cap \Sigma_{hib} \neq \Sigma_{Elig})$ **then**
23: $\quad\quad$ **return** $False$
24: $\quad$ **end if**
25: $\quad$ **if** $(\neg\,\text{NextState}(b, \Sigma_{poss}, \Sigma_{\mathbf{G}poss}, P_q, nextLabel, B_{map}, B_p, B_{conc}, P_{SF}, Z_{SP},$ $\text{Occu}_B(b)))$ **then**
26: $\quad\quad$ **return** $False$
27: $\quad$ **end if**
28: **end while**
29: $\text{CheckNerodeCells}(B_{conc}, \text{Occu}_B, pNerFail)$
30: **return** $True$

reachability tree until all nodes terminate at a *tick* event or one of the checks fail. As the tree is built, Point ii.1 of SD controllability with $||_{SD}$ property is tested at **line 22**.

This algorithm also calls Algorithms 16 and 17 that contribute in verifying Point ii.2 of SD controllability with $||_{SD}$.

B.3.2 Point ii.2

The following algorithms can be used to verify Point ii.2 of SD controllability with $||_{SD}$ property.

Next State Algorithm 16 is called by Algorithm 15 to determine all the next states that need to be processed for checking Point ii.2 of SD controllability with $||_{SD}$. This algorithm also checks the property of **S**-singular prohibitable behaviour with $||_{SD}$. This check is performed from **lines 12-16**.

Algorithm 16 NextState(...)

1: **if** $(\Sigma_{poss} = \emptyset)$ **then**
2: **return** $True$
3: **end if**
4: **if** $(\tau \in \Sigma_{poss})$ **then**
5: $P_{q'} \leftarrow \hat{\delta}(P_q, tick)$
6: Push$(B_{conc}, (b, P_{q'}))$
7: **if** $(P_{q'} \wedge P_{SF} \equiv false)$ **then**
8: $P_{SF} \leftarrow P_{SF} \vee P_{q'}$
9: Push$(Z_{SP}, P_{q'})$
10: **end if**
11: **end if**
12: **for all** $(\sigma \in \Sigma_{\mathbf{G}poss})$ **do**
13: **if** $(\text{Occu}_B(b) \cap \{\sigma\} \neq \emptyset)$ **then**
14: **return** $False$
15: **end if**
16: **end for**
17: **for all** $(\sigma \in \Sigma_{poss} - \{\tau\})$ **do**
18: $P_{q'} \leftarrow \hat{\delta}(P_q, \sigma)$
19: $b' \leftarrow nextLabel$
20: $nextLabel \leftarrow nextLabel + 1$
21: Push$(B_{map}, (b', P_{q'}))$
22: Push(B_p, b')
23: Push$(\text{Occu}_B, (b', \text{Occu}_B(b) \cup \{\sigma\}))$
24: **end for**
25: **return** $True$

Check Nerode Cells Algorithm 17 is called by Algorithm 15 to determine if there are possible violations for Point ii.2 of SD controllability with $||_{SD}$. These potentially problematic states are recorded in $pNerFail$ to be analyzed later.

Recheck Nerode Cells Algorithm 18 is called by Algorithm 9 to recheck Point ii.2 with respect to the potentially problematic states stored in $pNerFail$. It makes use of Algorithm 19 to conclude its result.

Recheck Nerode Cell Algorithm 19 is called by Algorithm 18 to recheck Point ii.2 with respect to the potentially problematic states stored in $pNerFail$. It determines if the set of states that this algorithm is called with are λ-equivalent to each other. If they are not, the algorithm returns $False$ indicating the violation of Point ii.2 of SD controllability with $||_{SD}$ property.

B.3.3 Point iii

Algorithm 20 verifies Point iii of SD controllability with $||_{SD}$. It determines if there exists a marked state in $\mathbf{G}_{cl} = \mathbf{S} ||_{SD} \mathbf{G}$ with an incoming non-*tick* transition from a reachable state. If such a state exists, Point iii fails and the algorithm returns $False$. Please note that in our $||_{SD}$ setting, $P_{reach} = R(\mathbf{S} ||_{SD} \mathbf{G}, true)$.

Algorithm 17 CheckNerodeCells($B_{conc}, \text{Occu}_B, pNerFail$)

```
1: while (B_conc ≠ ∅) do
2:     (b, P_q) ← Pop(B_conc)
3:     Z_eqv ← ∅
4:     Push(Z_eqv, P_q)
5:     sameCell ← True
6:     for all ((b', P_q') ∈ B_conc) do
7:         if (Occu_B(b) = Occu_B(b')) then
8:             Push(Z_eqv, P_q')
9:             B_conc ← B_conc − {(b', P_q')}
10:            if (P_q ≢ P_q') then
11:                sameCell ← False
12:            end if
13:        end if
14:    end for
15:    if (¬ sameCell) then
16:        Push(pNerFail, Z_eqv)
17:    end if
18: end while
19: return
```

Algorithm 18 RecheckNerodeCells($pNerFail$)

```
1: if (pNerFail = ∅) then
2:     return True
3: end if
4: Visited ← ∅
5: while (pNerFail ≠ ∅) do
6:     Z_eqv ← Pop(pNerFail)
7:     if (¬ RecheckNerodeCell(Z_eqv, Visited)) then
8:         return False
9:     end if
10: end while
11: return True
```

Algorithm 19 RecheckNerodeCell($Z_{eqv}, Visited$)

1: $P_{q_1} \leftarrow \text{Pop}(Z_{eqv})$
2: $Pending \leftarrow \emptyset$
3: **while** $(Z_{eqv} \neq \emptyset)$ **do**
4: $\quad P_{q_2} \leftarrow \text{Pop}(Z_{eqv})$
5: $\quad \text{Push}(Pending, (P_{q_1}, P_{q_2}))$
6: **end while**
7: **while** $(Pending \neq \emptyset)$ **do**
8: $\quad (P_{q_1}, P_{q_2}) \leftarrow \text{Pop}(Pending)$
9: $\quad P \leftarrow P_{q_1} \vee P_{q_2}$
10: $\quad$ **if** $((P \wedge P_m \not\equiv false)$ & $(P \wedge P_m \not\equiv P))$ **then**
11: $\quad\quad$ **return** $False$
12: $\quad$ **end if**
13: $\quad$ **for all** $(\sigma \in \Sigma)$ **do**
14: $\quad\quad P' \leftarrow \hat{\delta}(P, \sigma)$
15: $\quad\quad P'_{q_1} \leftarrow \hat{\delta}(P_{q_1}, \sigma)$
16: $\quad\quad P'_{q_2} \leftarrow \hat{\delta}(P_{q_2}, \sigma)$
17: $\quad\quad$ **if** $(P' \not\equiv false)$ **then**
18: $\quad\quad\quad$ **if** $((P'_{q_1} \wedge P' \not\equiv false)$ & $(P'_{q_2} \wedge P' \not\equiv false))$ **then**
19: $\quad\quad\quad\quad$ **if** $((P'_{q_1} \not\equiv P'_{q_2})$ & $((P'_{q_1}, P'_{q_2}) \notin Visited))$ **then**
20: $\quad\quad\quad\quad\quad \text{Push}(Visited, (P'_{q_1}, P'_{q_2}))$
21: $\quad\quad\quad\quad\quad \text{Push}(Visited, (P'_{q_2}, P'_{q_1}))$
22: $\quad\quad\quad\quad\quad \text{Push}(Pending, (P'_{q_1}, P'_{q_2}))$
23: $\quad\quad\quad\quad$ **end if**
24: $\quad\quad\quad$ **else**
25: $\quad\quad\quad\quad$ **return** $False$
26: $\quad\quad\quad$ **end if**
27: $\quad\quad$ **end if**
28: $\quad$ **end for**
29: **end while**
30: **return** $True$

Algorithm 20 CheckSDCPointiii(P_{reach})

1: $P \leftarrow \bigvee_{\sigma \in \Sigma - \{\tau\}} \hat{\delta}(P_{reach}, \sigma)$
2: **if** $(P \wedge P_m \not\equiv false)$ **then**
3: $\quad$ **return** $False$
4: **end if**
5: **return** $True$