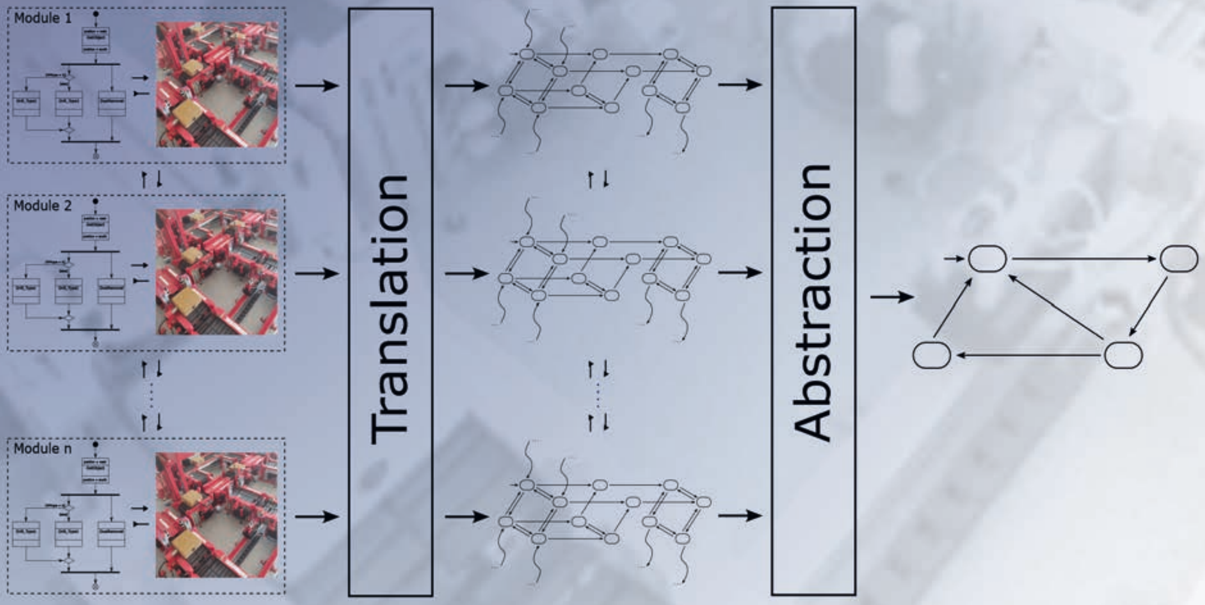

FAU Studien aus der Elektrotechnik 25

Yiheng Tang

Formal Verification in Automated Manufacturing

FAU University Press

Yiheng Tang

Formal Verification in Automated Manufacturing

FAU Studien aus der Elektrotechnik

Band 25

Herausgeber der Reihe:
Prof. Dr.-Ing. Bernhard Schmauß

Yiheng Tang

Formal Verification in Automated Manufacturing

Erlangen
FAU University Press
2024

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Kontakt: Yiheng Tang, Friedrich-Alexander-Universität Erlangen-Nürnberg (ROR https://ror.org/00f7hpc57), https://orcid.org/0009-0003-0396-2619

Bitte zitieren als
Tang, Yiheng. 2024. *Formal Verification in Automated Manufacturing.* FAU Studien aus der Elektrotechnik Band 25. Erlangen. FAU University Press. DOI: 10.25593/978-3-96147-744-9.

Verlag und Auslieferung:
FAU University Press, Universitätsstraße 4, 91054 Erlangen

Druck: docupoint GmbH

ISBN: 978-3-96147-743-2 (Druckausgabe)
eISBN: 978-3-96147-744-9 (Online-Ausgabe)
ISSN: 2363-8699
DOI: 10.25593/978-3-96147-744-9

Formal Verification in Automated Manufacturing

Formale Verifikation in der Fertigungsautomatisierung

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg

zur
Erlangung des Doktorgrades Dr.-Ing.

vorgelegt von

Yiheng Tang

Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung:	14. November 2023
Gutachter:	Prof. Dr.-Ing. Thomas Moor
	Prof. Dr.-Ing. Jörg Raisch

Acknowledgement

This dissertation was completed during my work as a research assistant at the chair of automatic control of Friedrich-Alexander University Erlangen-Nuremberg. I would like to express my appreciation to all the people who had helped me to finish this dissertation.

I would like to start by expressing my deepest gratitude to my advisor, Prof. Dr.-Ing. Thomas Moor, for his patient guidance, inspiring ideas and constant encouragement, which lay the foundation of this dissertation. Furthermore, I would like to sincerely thank Dr. rer. nat. Martin Witte from our industrial partner Siemens AG, who has provided various interesting concepts and use-cases from practical industry scenarios.

I would like to extend my gratitude to the chair holder, Prof. Dr.-Ing. Knut Graichen, for taking the chairmanship of the examining committee. I would also like to express my gratitude to Prof. Dr.-Ing. Jörg Raisch for taking the revision of my dissertation, as well as Prof. Dr. Christoph Pflaum for the participation in the examining committee.

I gratefully acknowledge all my colleagues at the chair of automatic control for the open research atmosphere. Special thanks go to the research group of discrete-event systems for all the fruitful discussions.

I would like to sincerely express my gratitude to my parents for the constant love and encouragement to me. Many thanks also to my friends Jingyuan Li, Danwei Shao and Dr. Hongzhe Zhou for their mental support.

Erlangen, November 2022 Yiheng Tang

Abstract

In recent decades, discrete-event modelling has been widely utilised to address control engineering problems (Preuße et al., 2012; Ramadge and W. Wonham, 1987). Comparing with conventional dynamic system modelling where physical behaviour is explicitly to describe, discrete-event modelling focuses on a more abstract level where logical behaviour is of interest. In this dissertation, we focus on the formal verification of the logical closed-loop behaviour of control systems. To satisfy safety and/or liveness requirements according to given technical specifications, we exploit the formal semantics of control programmes to represent the entire closed-loop behaviour in a discrete-event model, from which the properties of interest can be formally verified.

There are two major challenges to conquer in the current dissertation. The first one is the lack of formal semantics of control programmes. In practice, various modelling languages and programming languages have been developed for control programme design, e.g. Unified Modelling Language (Object Management Group, 2017b), Interdisciplinary Modelling Language (Brecher, Obdenbusch, Özdemir et al., 2016), Grafcet (Provost, J.-M. Roussel et al., 2011) and various programming languages defined in the IEC-61131 standard. Unfortunately, most of the original documents do not provide sufficiently formalised semantics to support formal verification. In particular, since we focus on the discrete-event dynamics of closed-loop behaviour, careful semantics formalisation based on the logic time axis is essential.

The second challenge to conquer is the computational efficiency of the formal verification for complex systems with modular and/or hierarchical structure. Typically, such systems are represented by a collection of synchronised automata, each of which has relatively few states (Leduc, 2002a; Schmidt et al., 2007). For such systems, we focus on verifying their non-blockingness in the current dissertation, which can express various safety properties and (weak) liveness properties (Cassandras and Lafortune, 2008). A conventional way to address this verification problem is based on analysing the monolithic representation of the entire system, which is usually infeasible due to the exponential growth of the state space, a.k.a. the state explosion problem. One approach that mitigates this issue is the compositional verification (Flordal and Malik, 2009). Basically, the idea of compositional verification is to iteratively (i) abstract each synchronised automaton and (ii) compose a small set of automata to form a subsystem. The iteration terminates when there is only one automaton left. In particular, it is required that the abstraction preserves

the property to verify. This guarantees that verifying the monolithic representation is equivalent to verifying the final automaton after iteration, which in general has fewer states. Specifically for compositional non-blockingness verification, various recent contributions have shown convincing results (Flordal and Malik, 2009; Pilbrow and Malik, 2015; Su et al., 2010; Ware and Malik, 2012), where it is generally assumed that all automata are synchronised through the standard synchronous composition (Cassandras and Lafortune, 2008; Milner, 1989). In this dissertation, we address the compositional non-blockingness verification problem where events are prioritised. More precisely, we envisage that each event in the entire system has a priority value. In any state, events with lower priority can never be executed if any event with higher priority is active in this state. This feature can result from e.g. the formal semantics of the control programme and indeed changes the way of synchronisation. Thus, for modular/hierarchical systems with prioritised events, existing frameworks and results w.r.t. compositional non-blockingness verification need to be carefully reviewed and adjusted.

Kurzzusammenfassung

In den letzten Jahrzehnten wurde die ereignisdiskrete Modellierung immer öfter angewandt, um regelungstechnische Probleme zu behandeln (Preuße et al., 2012; Ramadge and W. Wonham, 1987). Im Vergleich zur konventionellen Modellierung von dynamischen Systemen, wobei physikalisches Verhalten explizit zu beschreiben ist, konzentriert sich die ereignisdiskrete Modellierung auf eine abstraktere Ebene, auf der logisches Verhalten von Interesse ist. In dieser Dissertation konzentrieren wir uns auf die formale Verifikation des logischen Verhaltens von Regelkreisen. Um Sicherheits- und/oder Lebendigkeitsanforderungen anhand gegebener technischer Spezifikationen zu gewährleisten, verwenden wir die formale Semantik von Steuerprogrammen, um den gesamten geschlossenen Regelkreis von einem ereignisdiskreten System darzustellen, so dass die interessierenden Eigenschaften formal verifiziert werden können.

In dieser Dissertation sind zwei wesentliche Herausforderungen zu bewältigen. Der erste ist die fehlende formale Semantik von Steuerungsprogrammen. In der Praxis stehen verschiedene Modellierungssprachen und Programmiersprachen für den Entwurf von Steuerungsprogrammen zur Verfügung, z.B. Unified Modelling Language (Object Management Group, 2017b), Interdisciplinary Modelling Langauge (Brecher, Obdenbusch, Özdemir et al., 2016), Grafcet (Provost, J.-M. Roussel et al., 2011) und verschiedene in der Norm IEC-61131 definierte Programmiersprachen. Leider bieten die meisten originalen Dokumente keine ausreichend formalisierte Semantik, um die formale Verifikation zu ermöglichen. Da die ereignisdiskrete Dynamik in geschlossenen Regelkreisen für uns von Interesse ist, ist eine sorgfältige Formalisierung von Semantik auf der logischen Zeitachse notwendig.

Die zweite zu bewältigende Herausforderung ist die rechnerische Effizienz der formalen Verifikation für komplexe Systeme mit modularer und/oder hierarchischer Struktur. Solche Systeme werden typischerweise durch mehrere synchronisierte Automaten repräsentiert, von denen jeder relativ wenige Zustände besitzt (Leduc, 2002a; Schmidt et al., 2007). In dieser Dissertation konzentrieren wir uns für solche Systeme auf die Verifikation von Blockierungsfreiheit, die verschiedene Sicherheits- und (schwache) Lebendigkeitseigenschaften (Cassandras and Lafortune, 2008) ausdrücken kann. Eine konventionelle Vorgehensweise von dieser Aufgabe basiert auf der Analyse der monolithischen Darstellung des gesamten Systems, was oft wegen des exponentiellen Wachstums des Zustandsraums, auch bekannt als State Explosion

Problem, nicht durchführbar ist. Ein Ansatz, der dieses Problem mildert, ist die Compositional Verification (Flordal and Malik, 2009). Grundsätzlich besteht die Idee der Compositional Verification darin, iterativ (i) jeden synchronisierten Automaten zu abstrahieren und (ii) eine kleine Menge von Automaten zusammenzusetzen, um ein Subsystem zu formen. Wenn nur ein Automat verbleibend ist, terminiert die Iteration. Insbesondere ist es erforderlich, dass die Abstraktion die zu verifizierende Eigenschaft erhaltet. Dies garantiert, dass das Verifikationsergebnis der monolithischen Darstellung mit dem des verbleibenden Automaten nach der Iteration übereinstimmt, der im Allgemeinen weniger Zustände hat. Speziell für die Compositional Verification der Blockierungsfreiheit haben verschiedene neueste Beiträge überzeugende Ergebnisse geliefert (Flordal and Malik, 2009; Pilbrow and Malik, 2015; Su et al., 2010; Ware and Malik, 2012), wobei allgemein angenommen wird, dass alle Automaten durch die standardmäßige synchrone Komposition (Cassandras and Lafortune, 2008; Milner, 1989) synchronisiert sind. In dieser Dissertation untersuchen wir die Compositional Verification der Blockierungsfreiheit, bei der Ereignisse priorisiert sind. Genauer gesagt können wir uns so vorstellen, dass jedes Ereignis im gesamten System einen Prioritätswert besitzt. In jedem Zustand können die Ereignisse mit niedrigerer Priorität nicht ausgeführt werden, wenn in diesem Zustand irgendein Ereignis mit höherer Priorität aktiv ist. Diese Eigenschaft kann z.B. aus der formalen Semantik des Steuerungsprogramms ergeben und ändert die Art und Weise von Synchronisation. In diesem Zusammenhang muss die existierenden Methoden und Ergebnisse bzgl. Compositional Verification der Blockierungsfreiheit sorgfältig überprüft und angepasst werden.

Contents

1 Introduction

In modern industrial manufacturing, production procedures are highly automated through logical control programmes, while manual operations through workers tend to be less involved. In this context, ensuring that the entire manufacturing system satisfies certain *safety* and *liveness* requirements (Alpern and Schneider, 1987) is of great practical value, i.e.

Safety Does the manufacturing system potentially exhibit any unsafe behaviour? E.g. is there a risk of collision between two robot arms when they share some region within their individual movement?

Liveness Does the manufacturing system always make progress? E.g. can the processing of a workpiece in a machine eventually be terminated?

Instead of performing tedious enumerative tests on real physical systems, which is extremely time-consuming and may also threaten human life and property safety, formally ensuring that the desired properties are fulfilled already in the system design and development phase is obviously preferred. Typically, safety and liveness requirements are considered as *temporal properties* of a dynamic system, which can formally be represented by *finite automata* (Cassandras and Lafortune, 2008; Daniele et al., 1999; M. Y. Vardi, 1996).[1] Typically, an automaton is a directed graph where each vertex is referred to as a *state* and each directed edge connecting two states is considered a *transition*. Besides, each transition is labelled by an *event*.

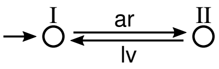

Figure 1: An automaton

As an example, Figure 1 shows an automaton describing the logical behaviour of a binary sensor which detects the presence of workpieces in front of it. Two events ar (for *arrive*) and lv (for *leave*) are labelled on the two transitions connecting both states I and II, indicating that workpieces alternatively arrive and leave the sensor. In particular, by considering state I as the initial state, it is implied that no workpiece is present at the beginning, since event ar, instead of lv, always occurs first.

[1] An automaton is *finite* if it has finitely many states. Throughout the current dissertation, we assume that all automata are finite.

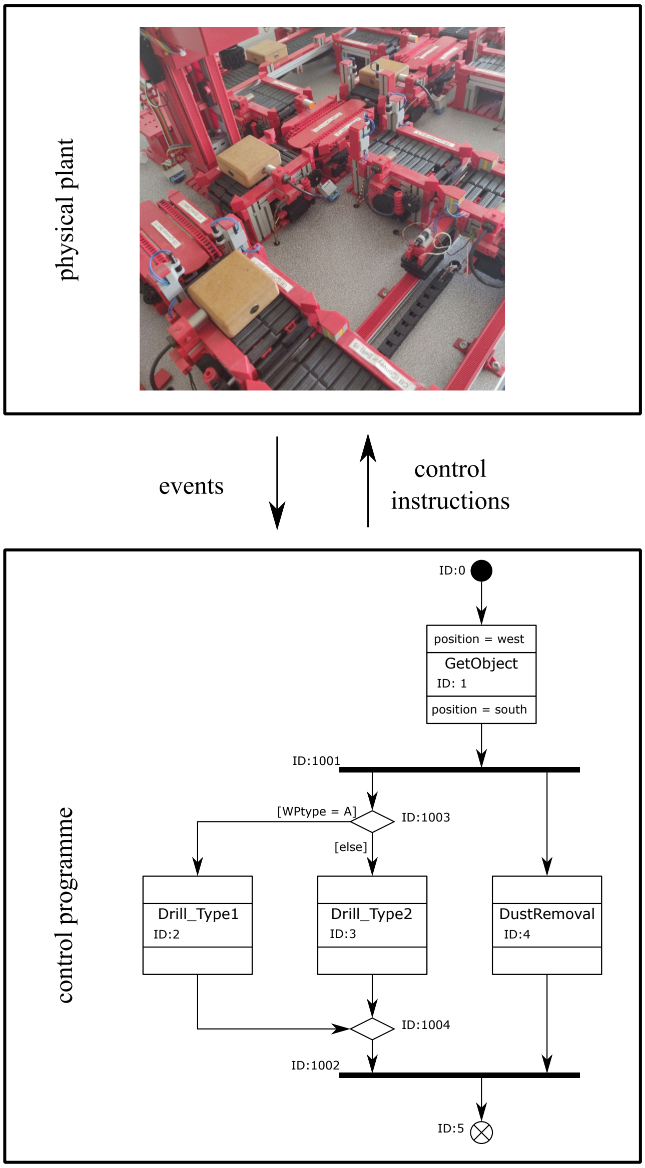

Figure 2: Closed-loop behaviour

With automata as basic model, we now discuss how to ensure desired properties in a manufacturing system. From a control-theory perspective, we consider that the entire system behaviour, which is also referred to as the *closed-loop behaviour*, is represented by a *plant* (i.e. the uncontrolled behaviour) and a feed-back *controller*. By reading event sequences from the plant, the controller sends control instructions back to the plant; see Figure 2. As the uncontrolled plant reflects the “physical nature”, we are interested in the controller, which is the “man-made” counterpart and enforces the desired properties in the closed-loop system. One well-known approach to achieve this goal is the *Supervisory Control Theory (SCT)* (Ramadge and W. Wonham, 1989; Ramadge and W. Wonham, 1987). Given a plant model and a formal *specification* describing the intended system behaviour and desired properties, SCT automatically *synthesises* a controller (which is also referred to as a *supervisor*) that enforces the specification in the closed-loop behaviour. However, although SCT has been actively studied and developed in recent decades, one relative drawback of SCT is that constructing formal discrete-event models, especially formal specifications, is a challenging task that necessarily requires highly advanced mathematical knowledge. This is one of the main reasons why, in practice, SCT is seldom applied by automation engineers to solve real-world control problems.

One alternative to controller synthesis is *formal verification*, which is a well-discussed topic in computer science, e.g. in the theory of *Model Checking* (E. M. Clarke et al., 2001). In this case, we envisage that control programmes are already available and the resulting closed-loop system is algorithmically *verified* (Bauer, Engell et al., 2004; Buzhinsky and Vyatkin, 2017; Gerber et al., 2010; Preuße et al., 2012). One major benefit of applying formal verification is that nominal control sequences are already realised in control programmes (which is usually considered as “specification” as well when applying SCT). Thus, the properties to verify are usually much easier to formulate. If the verification result is positive, the control programme is considered useable for the manufacturing system; otherwise, revision of the codes is necessary (possibly with the help of counterexamples from the verification, i.e. an “evidence” which tells how the desired property can by invalidated).[2]

Describing closed-loop behaviour with automata Recall that close-loop behaviour is represented by combining a plant and a controller. As we utilise automata as our basic model, we purpose that both the plant and the controller are modelled by automata. The closed-loop system is then computed based on the *synchronous composition* of the plant and the controller model, which is a common approach in SCT (Cassandras and Lafortune, 2008). On this basis, we envisage the overall procedure for the closed-loop behaviour verification as shown in Figure 3. In the scope of the current dissertation,

[2] Note that the “trial-and-error” of programme codes is generally unavoidable for controller synthesis approaches as well. As for SCT, it is common that ill-formed specifications result in an overly pessimistic controller that disallows everything in the closed-loop behaviour.

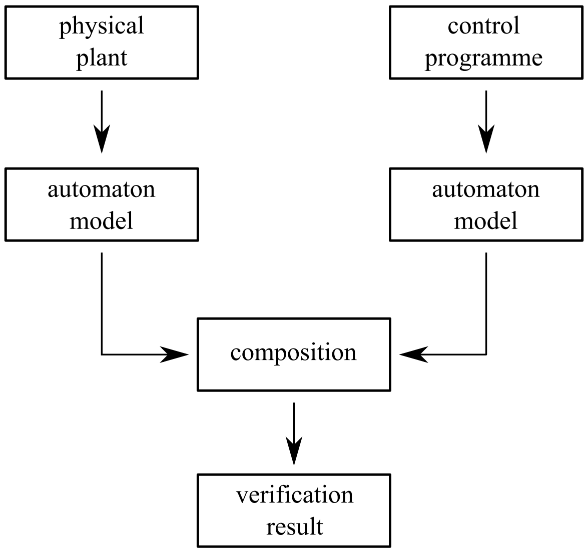

Figure 3: Envisaged verification procedure

we propose that the plant model is directly available, which may have been manually designed (possibly from some component library). From a practical perspective, manually constructing plant models is not a verbose task since plant behaviour is usually "determined", i.e. when programming control codes, it is rather unlikely to redesign and reassemble the machine. In contrast, since programme codes may be edited at any time (due to e.g. new functional requirements or negative verification results), an automatic procedure to translate programme codes into automata is indispensable. To this end, we first focus on the *Sequential Behaviour Diagram (SBD)* defined by *Interdisciplinary Modelling Language (IML)* (Brecher, Obdenbusch, Özdemir et al., 2016; Flender et al., 2019) and utilise SBDs as the formal representation for control programmes. As being derived from the well-known *Unified Modelling Language (UML)* (Object Management Group, 2017b) and *System modelling language (SysML)* which have more general modelling purposes, the recently developed IML has a specific focus on industrial automation systems with compactly three types of diagrams (comparing with 14 diagram types in UML and 9 diagram types in SysML). SBD is a variant of *Activity Diagram (AD)* from UML (and SysML as well) that utilises token propagation on a Petri-net-like structure to illustrate concurrent processes. In fact, a demonstrating example of an SBD has already been given on the right side of Figure 2. Each "square block" in an SBD is referred to as a *process*, which is considered an abstract programme block that can hold a token. In addition, each edge can propagate tokens in its direction. Unfortunately, existing literature does not sufficiently formalise SBD semantics to enable its translation to automata. This problem is addressed in Chapter 4.

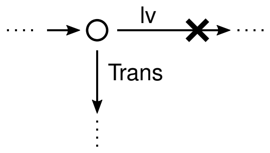

Figure 4: High-priority events preempt low-priority events

At the current stage, a specific feature appearing in the translation result is worth mentioning – all events in the resulting automata are *prioritised*. In any state in the closed-loop behaviour where a high-priority event is active, all events with lower priority cannot be executed. This semantic feature was originally studied in process algebra (Baeten et al., 1986; Cleaveland et al., 2007; Lüttgen, 1998). As for SBDs, we shall stipulate in some cases that token propagation has higher priority over other behaviour. We consider the automaton fragment given in Figure 4 as an example. The event Trans standing for token propagation has higher priority over the event lv, which corresponds to a positive or negative edge in the line level of some sensor. In this case, we should remove the transition labelled by lv as it will always be *preempted* by Trans. This is also the reason why we loosely wrote *composition* instead of the standard *synchronous composition* in Figure 3, i.e. the closed-loop behaviour in this context is represented by removing low-priority transitions (which is also referred to as *shaping*) in the synchronous composition.

Compositional non-blockingness verification with prioritised events In the current dissertation, we pay specific attention to the *non-blockingness* as the property of our interest. Non-blockingness is one of the most common properties required for closed-loop behaviour (Cassandras and Lafortune, 2008; Ramadge and W. Wonham, 1987) which states that in any reachable state, it is always possible to attain desired system configurations in the future. As for the example automaton in Figure 1, one can specify that state II encodes a desired system configuration, which implies that the automaton is non-blocking. The definition of non-blockingness infers some weak[3] liveness properties of the system. Nevertheless, a great variety of safety properties are expressible by non-blockingness. Typically, if reaching certain states causes

[3] Non-blockingness is *weak* since it only requires the system to have the opportunity to reach some good future, while the system does not necessarily need to reach it. See also (Alpern and Schneider, 1985; De Giacomo and M. Vardi, 2013).

safety issues, e.g. collision or overheating, these states are considered unsafe and can be modelled as blocking states in plant models. Since the closed-loop behaviour needs to be non-blocking, the plant behaviour must be restricted by the controller so that unsafe states are rendered unreachable.

For a moderately sized system, its non-blockingness can be simply verified by performing enumerative backward reachability search in the monolithic representation of the system (Cassandras and Lafortune, 2008). However, for large-scale systems with multiple synchronised *modules*, constructing a monolithic representation greatly suffers from the notorious *state explosion problem*, as the entire state space increases exponentially w.r.t. the number of modules. E.g., even each module has only 5 states, 10 synchronised modules can still reach a total state count of $5^{10} \approx 9.8 \times 10^6$ in the monolithic representation. To this end, various contributions in recent decades have attempted to solve the non-blockingness verification problem for modular systems without explicitly constructing the monolithic representation. One well-discussed approach is to utilise *binary decision diagrams (BDDs)* (Akers, 1978) to symbolically encode automata (Kimura and E. Clarke, 1990; Michon and Champarnaud, 1998), which, compared with enumerating the entire transition structure, potentially reduces the memory required for representing the state space.

A well-established alternative to address the non-blockingness verification problem is *compositional verification* (Flordal and Malik, 2009; Pilbrow and Malik, 2015; Su et al., 2010; Ware and Malik, 2012), which is based on applying *abstractions* on individual modules. The underlying idea of compositional verification stems from *compositional reasoning*, which derives the *interface rule* in compositional model checking (E. Clarke et al., 1989). The basic idea of the interface rule is illustrated in Figure 5, where we suppose that the

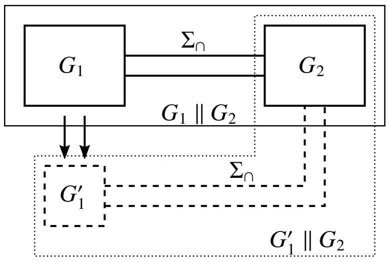

Figure 5: Interface rule

entire system $G_1 \parallel G_2$ consists of two modules G_1 and G_2.[4] Both modules communicate with each other through a set of events $\Sigma_\cap$. Besides, we assume that the property to verify ϕ is expressed by another set of events Σ'. In this context, the idea is to construct a *suitable* abstraction G_1' (which is also referred to as an *interface* in (E. Clarke et al., 1989)) of G_1 by neglecting all events not in $\Sigma_\cap \cup \Sigma'$, i.e. events owned by G_1 *privately* and irrelevant to ϕ. The suitableness of the abstraction then guarantees that verifying ϕ in $G_1 \parallel G_2$ is equivalent to verifying ϕ in $G_1' \parallel G_2$, which typically has fewer states than $G_1 \parallel G_2$. As for non-blockingness verification, the process-algebraic equivalence *conflict equivalence* was proposed in (Malik, Streader et al., 2004) to guarantee the suitableness of an abstraction. Two automata, say G_1 and G_1' where G_1' is an abstraction of G_1, are considered conflict equivalent if for any automaton T, $G_1 \parallel T$ is non-blocking if and only if $G_1' \parallel T$ is non-blocking. As for the situation in Figure 5, G_2 clearly is “any automaton”. At this stage, it is also worth mentioning that, as the composition is generally associative and commutative, the abstraction can be applied to each module in an arbitrary order. E.g. for 10 modules, reducing the state count of each module from 5 to 3 already yields an appreciable overall state space reduction from 9.8×10^6 to 5.9×10^4.

Another key feature of compositional verification is that abstraction can also be applied *iteratively* (Flordal and Malik, 2009; Su et al., 2010). Recall that abstractions make use of *private* events. Suppose that a modular system consists of five modules $G_1, \dots, G_5$ where each module has been abstracted, resulting into $G_1', \dots, G_5'$, respectively. At this stage, strategically choose a small set of automata to compose, e.g. $G_1' \parallel G_2' =: G_{12}$, potentially enables further abstractions since events only being shared by G_1 and G_2 are rendered private. Thus, composition and abstraction can be iteratively applied to the entire system until there is only one module left, whose non-blockingness is identical to that of the monolithic representation. Figure 6 shows a possible procedure to apply compositional verification for five modules. Each edge transforming an automaton $G_{\_}$ into $G_{\_}'$ indicates the application of suitable abstractions, while edges merging multiple automata into a single automaton indicates the composition. In this context, verifying the non-blockingness of $G_1 \parallel G_2 \parallel G_3 \parallel G_4 \parallel G_5$ is equivalent to verifying G_{12345}, which usually has significantly fewer states.

Recently, various abstraction methods have been developed for the compositional non-blockingness verification of (ordinary) automata (Flordal and

[4] In this paragraph, we loosely utilise the operator $\parallel$ to denote some kind of composition which is commutative and associative.

$$\begin{tikzcd}
G_1 \arrow[r] & G'_1 \arrow[dr] & & & \\
& & G_{12} \arrow[r] & G'_{12} \arrow[dddr] & \\
G_2 \arrow[r] & G'_2 \arrow[ur] & & & \\
& & & & \\
G_3 \arrow[r] & G'_3 \arrow[rrr] & & & G_{12345} \\
& & & & \\
G_4 \arrow[r] & G'_4 \arrow[dr] & & & \\
& & G_{45} \arrow[r] & G'_{45} \arrow[uuur] & \\
G_5 \arrow[r] & G'_5 \arrow[ur] & & &
\end{tikzcd}$$

Figure 6: Possible procedure to apply compositional verification for a modular system with five modules

Malik, 2009; Malik, 2015; Pilbrow and Malik, 2015; Su et al., 2010; Ware and Malik, 2012). Besides, compositional verification has been successfully applied to several extended types of automata and/or other properties (Malik and Leduc, 2013; Mohajerani, Malik et al., 2016; Ware and Malik, 2013). Since non-blockingness is also one of the standard properties required for SCT, *compositional synthesis* of supervisors has also been widely discussed and shown convincing results (Malik and Teixeira, 2016; Mohajerani, Malik et al., 2014; Mohajerani, Malik et al., 2017). However, it is challenging to apply existing results to verify the non-blockingness of modular/hierarchical SBDs, since prioritised events influence the synchronisation of modules. As far as the author's knowledge, most contributions addressing compositional verification problems take synchronous composition as the semantics of synchronisation between modules, i.e. the behaviour of the entire system complies with the synchronous composition of all modules. Unfortunately, this is not the case for SBDs due to prioritised events. As we shall see in Chapter 2, the high-priority of token-propagation events has a global effect across all modules. By referring to the case in Figure 6, the behaviour of the entire modular system shall comply with $\mathcal{S}(G_1 \parallel G_2 \parallel G_3 \parallel G_4 \parallel G_5)$, where the *shaping operator* $\mathcal{S}(\cdot)$ removes low-priority transitions in each state in an automaton. In this case, it is conceivable that the ordinary conflict equivalence does not guarantee the suitableness of abstraction any more – new equivalence over automata with new abstraction methods need to be developed for compositional non-blockingness verification with prioritised events. This topic is discussed in Chapter 3 in more detail.

Outline The outline of the current dissertation is as follows. In Chapter 2, we concentrate on translating SBDs into automata. This starts with a rigorous formalisation of the syntax and semantics of SBDs, where we also consider modularly and hierarchically structured SBDs from a practical perspective. In particular, the discrete event dynamic behaviour of SBDs is clarified over the *logic dense time axis*, which is naturally correlated with the semantics of automata. This chapter ends with a practical example, where a set of modular and hierarchical SBDs are constructed to control a production line. The non-blockingness of the entire closed-loop system is then envisaged to be verified by the compositional verification procedure introduced in Chapter 3 – in Chapter 3, we focus on modular systems whose behaviour is represented by synchronised automata with prioritised events. By formally defining the shaping operator $\mathcal{S}(\cdot)$, we purpose a new equivalence over automata, i.e. the *conflict equivalence w.r.t. prioritised events*, as a new characterisation for suitable abstractions in our case. On this basis, new abstraction rules are developed. At the end of this chapter, compositional verification is applied to two different use-cases with prioritised events, including the SBD model previously constructed in Chapter 2. Finally, in Chapter 4, we discuss a graphical programming language, the *Sequential Function Chart (SFC)*, which has been actively used in industry for years. The motivation for adding this final chapter is that the Petri-net-based structure of SFCs is apparently comparable with SBDs. Thus, a question naturally arises is whether the translation procedure for SBDs introduced in Chapter 2 and the compositional verification approach developed in Chapter 3 are applicable for SFC verification. Nevertheless, due to the physical-time-based semantics of SFC as well as the specific execution order of SFC control sequences, careful extensions and assumptions are necessary.

2 Sequential behaviour diagram

The main objective of the current dissertation is to verify the controlled behaviour of manufacturing systems, a.k.a. *closed-loop behaviour* which is composed from a plant model and a controller model. In particular, we focus on the controller part and seek a possible formal representation which, from a practical perspective, is sufficiently intuitive and comprehensive for automation engineers. To this end, we set our sights on the concept of *modelling languages*, which has been intensively discussed recently in various fields, e.g. software engineering, business management and industrial manufacturing. The aim of modelling languages is to standardise the procedures of design and analysis of complex systems. In particular, many of the modelling languages provide various possibilities to model sequential behaviour of a target system, which can be utilised to design control programmes in automated manufacturing. One common choice is the *Activity Diagrams (AD)* from the well-known *Unified Modelling Language (UML)* (Object Management Group, 2017b) and *System Modelling Language (SysML)* (Object Management Group, 2017a); see e.g. (Fanti et al., 2013; Köhler et al., 2000; Y. Liu et al., 2014).

One of the most high-lighted features of UML and SysML is their flexibility and versatility in different modelling domains. However, this may become a burden when it comes to formal verification, which generally requires the formalisation of the massive semantic structure documented in natural language. In fact, many of the recent contributions addressing formal semantics of AD take a subset from the complete language; see e.g. (Daw and Cleaveland, 2015a; R. Eshuis, 2006; Jarraya et al., 2009; Lima et al., 2013). In this context, we focus on the recently developed *Interdisciplinary Modelling Language (IML)* (Brecher, Obdenbusch, Özdemir et al., 2016; Flender et al., 2019; Herfs et al., 2018) which has a specific aim of enabling common and consistent production machine design with interdisciplinary technical requirements. Technically, IML provides three types of diagrams for graphical modelling: *Functional Structure (FS)* represents functions and their sub-functions in a hierarchical fashion and determines the corresponding physical components realising the respective function; *Interaction Structure (IS)* describes the interaction between components through physical links and information flows; *Sequential Behaviour Diagram (SBD)* establishes the abstract structure of control sequences which realises the functions of the machine. Among the three diagram types, SBD is closely related to AD in that they both describe concurrent sequences through Petri-net-like structures. In addition, this intuition is followed by other programming languages for industrial applications as well, e.g. *Sequential Function Charts (SFC)* as defined in the IEC-61131 standard. Thus, in this chapter, we select SBDs as the formal representation of control programmes for manufacturing systems.

The subsequent question is, what property should be formally verified. In this dissertation, we are mostly interested in the *non-blockingness*, which can describe various safety properties as well as weak liveness properties in closed-loop behaviour. To this end, we propose to translate SBDs into automata, which is one of the most common models for non-blockingness analysis (Cassandras and Lafortune, 2008; Ramadge and W. Wonham, 1987). In particular, the translation result can be composed with plant models given in automata as well to form closed-loop behaviour, whose non-blockingness shall be formally verified. To this end, a sufficiently formalised semantics of SBDs is essential for the translation procedure, which, however, is not provided in existing documentations. Thus, in the current chapter, we first focus on the formalisation of SBD semantics by clarifying the discrete event dynamic behaviour of SBDs over *logic dense time axis*, which differs from the ordinary physical time axis in that multiple events can be “stacked” on a same physical time instance. In other words, a sequence of events can be executed without consuming a positive duration of physical time. This time model correlates SBD semantics with automata, which enables a semantically precise translation from SBDs to automata.

This chapter is organised as follows: Section 2.1 introduces the formal syntax and semantics of SBDs, based on which the translation procedure is developed in Section 2.2. In Section 2.3, a typical practical use-case is considered, based on which several extended semantic features are suggested for more precise verification results. Finally, a relatively complicated practical example is modelled in Section 2.4, based on which a brief overview of the non-blockingness of SBDs is presented.

2.1 Syntax and semantics

In this section, we introduce the formal syntax and semantics of SBDs. Being a Petri-net-like graph, the dynamic behaviour characterised by SBDs is basically organised by *token propagation*. Hence, structural components similar to Petri-net places (in which tokens can be held), transitions as well as directed edges (along which tokens are propagated) are conceivable. We first consider a prototypical example of a drill station as depicted in Figure 7 to demonstrate some basic features of SBDs. The drill station comprises a robot arm which

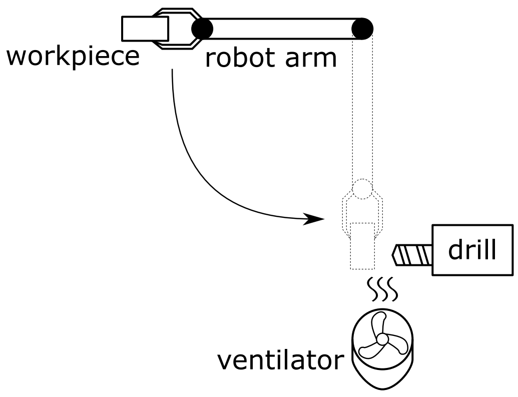

Figure 7: A drill station

can fetch workpieces, a drill which processes workpieces, and a ventilator that removes dust while drilling. The intended usage of the drill station is that, after taking a workpiece to the south position, a whole is drilled into the workpiece, possibly with different depths depending on the workpiece type. While drilling, the ventilator continuously blows off the dust. The above specification can be expressed by the SBDs given in Figure 8, in which we highlight the following features:

Nested SBDs One entire IML project may consist of multiple SBDs. In Figure 8, the SBD T is *nested* to SBD S, which is denoted by "⋔: T" in S. The operation of T is activated whenever S invokes it, while the operation of S is started spontaneously.

Nodes and edges Nodes are basic structural elements in SBDs which are connected by directed edges. In Figure 8, all nodes are numbered with a unique ID. Note that not all types of nodes can hold tokens for a positive duration of time.

Process nodes Process nodes are the core of SBDs. Each process node is a "black box" which can be seen as an abstract representation of a control programme fragment. In Figure 8, nodes with ID 1, 2, 3, 4, 11 are process nodes. Each process node has its pre- and postcondition to denote the prerequisite and the guaranteed result of executing the process, respectively. These are directly represented at the top or bottom of a process node, respectively, and is trivially true if the respective box is empty. For example, the process node with ID = 1 can be executed only when the precondition $\mathsf{position} = \mathsf{west}$ is fulfilled. After execution, it is guaranteed

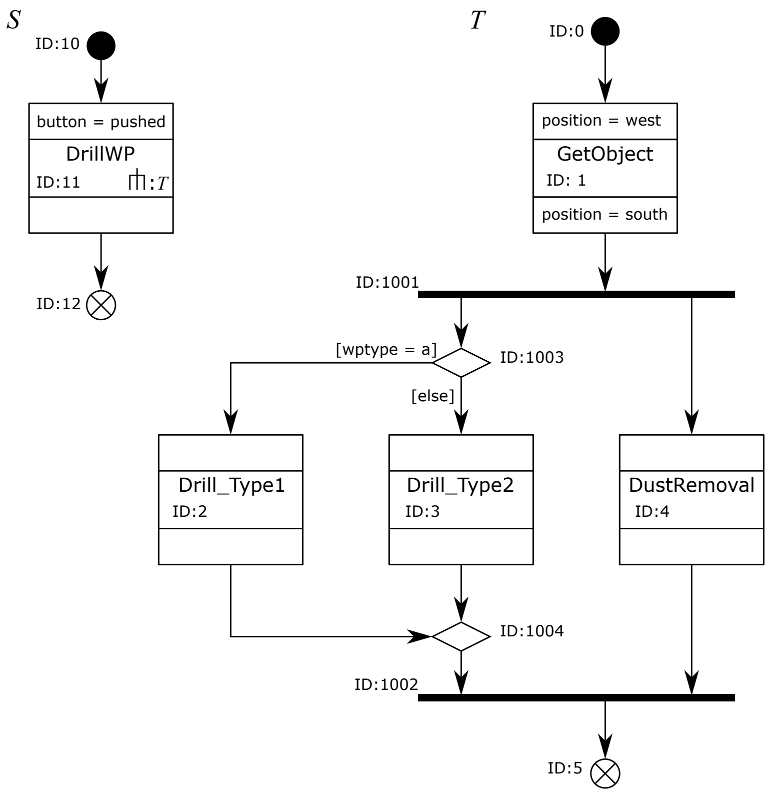

Figure 8: SBDs describing the control sequences of a drill station

that position = east is achieved. Besides,the process node with ID = 4 has trivial precondition and trivial postcondition.

Auxiliary nodes All nodes not being process nodes are referred to as auxiliary nodes, which are adapted from ADs defined in UML. These are initial nodes (ID: 0, 10) which initialise token propagation, terminal nodes (ID: 5, 12) which eliminate tokens, forks/joins (ID: $1001/1002$) which begin/terminate synchronous sequences, and branches/merges (ID: $1003/1004$) which begin/terminate alternative sequences.

2.1.1 Syntax and informal semantics

We first formalise the syntax of SBDs and introduce the intended usage of all types of SBD components. We shall first notice that, as mentioned in the drill station example, a complete IML project may include multiple SBDs, whose executions are related to each other. Thus, we first declare the scope of SBDs that are relevant to the control of a given manufacturing system, namely, an *SBD project*.

Definition 2.1.1. *An* SBD project *is a family* $\mathsf{SBDP} = (S_i)_{1 \leq i \leq k}$ *of* SBDs.

Technically, an SBD $S \in \mathsf{SBDP}$ is a special kind of directed graph in which to each node (or "vertex" from a graph-theoretical perspective), a node-type attribute is assigned. Recall that there are totally seven types of nodes, which motivates us to define the set of node types as

$$\mathsf{NodeTypes} = \{\mathsf{initial}, \mathsf{process}, \mathsf{terminal}, \mathsf{fork}, \mathsf{join}, \mathsf{branch}, \mathsf{merge}\}. \tag{1}$$

We are now prepared to give the definition of an SBD.

Definition 2.1.2. *Given an SBD project* SBDP*, an* SBD $S \in \mathsf{SBDP}$ *is a tuple* $S = \langle \mathsf{Nodes}_S, \mathsf{Edges}_S, \mathsf{type}_S, \mathsf{invoke}_S, \mathsf{Guards}_S \rangle$ *where*

- Nodes_S *is the set of nodes;*
- $\mathsf{Edges}_S \subseteq \mathsf{Nodes}_S \times \mathsf{Nodes}_S$ *is the set of directed edges;*
- $\mathsf{type}_S : \mathsf{Nodes}_S \to \mathsf{NodeTypes}$ *is the node type assignment function;*
- $\mathsf{invoke}_S : \mathsf{Processes}_S \to \mathsf{SBDP} \,\dot{\cup}\, \{\emptyset\}$ *is the invocation function with* $\mathsf{Processes}_S$ *being the set of all nodes with the node type* process *within S;*
- Guards_S *is a substructure for condition assignments.*

In the following, we take the convention that for any $S, T \in \mathsf{SBDP}$ where $S \neq T$, $\mathsf{Nodes}_S \cap \mathsf{Nodes}_T = \emptyset$ must hold. For brevity, the subscript $(\cdot)_S$ of elements of a given SBD $S \in \mathsf{SBDP}$ is often dropped if it is clear from the context that only one SBD is currently discussed, e.g. we may write $n \in \mathsf{Nodes}$ instead of $n \in \mathsf{Nodes}_S$. Besides, for any node $n \in \mathsf{Nodes}$ in some SBD, we define

$$\mathsf{pre}(n) := \{\, n' \in \mathsf{Nodes} \mid (n', n) \in \mathsf{Edges} \,\}; \tag{2}$$
$$\mathsf{suc}(n) := \{\, n' \in \mathsf{Nodes} \mid (n, n') \in \mathsf{Edges} \,\} \tag{3}$$

to conveniently access its *predecessors* and *successors*.

Clearly, randomly connecting two nodes of any types through edges shall not always result in a well-formed SBD with practical meaning. This motivates us to first discuss the intended usage of each type of nodes, which inspires us to stipulate several reasonable syntactical restrictions for a syntactically well-formed SBD. Basically, SBDs describes the sequential behaviour of concurrent processes, which is referred to as *SBD dynamics* in the following. SBD dynamics is organised by *token propagation*, which is a concept originates from Petri-nets. As for SBDs, tokens are propagated in edge directions to model the sequence of process activation (by receiving a token) and deactivation (by sending a token), which is similar to the so-called *control flows* in AD. Since tokens do not carry concrete objects (as opposed to modelling e.g. resources), we stipulate that the weight of each edge is equal to 1. Besides, each node can carry at most one token at the same time, which is an intended restriction especially due to the conceptional meaning of process nodes; see below for a detailed discussion.

Initial nodes A node $n \in \mathsf{Nodes}$ with $\mathsf{type}(n) = \mathsf{initial}$ is an *initial node* and we write

$$\mathsf{InitialNodes} := \{\, n \in \mathsf{Nodes} \mid \mathsf{type}(n) = \mathsf{initial} \,\} \tag{4}$$

for the set of all initial nodes in an SBD. An initial node has no predecessors and exactly one successor. Upon the activation of the current SBD, one token is generated in each initial node and the token is immediately propagated to its successor as soon as the successor is capable of receiving a token.

Processes A node $n \in \mathsf{Nodes}$ with $\mathsf{type}(n) = \mathsf{process}$ is a *process node* (or concisely a *process*) and we write

$$\mathsf{Processes} := \{\, n \in \mathsf{Nodes} \mid \mathsf{type}(n) = \mathsf{process} \,\} \tag{5}$$

for the set of all processes in an SBD. A process has exactly one predecessor as well as exactly one successor and models a programme block which takes a non-negative duration of physical time to execute. To abstractly describe the dynamic behaviour of processes, we utilise the so-called *process state* to describe the cyclic execution of each process; namely,

$$\mathsf{ProcessStates} := \{\mathsf{idle}, \mathsf{busy}, \mathsf{done}\}, \tag{6}$$

without explicitly specifying the concrete content of a process. Each process is initially in the process state idle upon the activation of the current SBD, which is immediately switched to the process state busy when it receives a token. This starts the execution of the programme associated with this process. Afterwards, upon the termination of the programme, the process state is immediately switched to done. In this state, if the successor is ready to take a token, the token is immediately sent to the successor and the process state cycles back to idle. It is worth mentioning that, we have already introduced all types of nodes which can hold a token for a positive duration of physical time, i.e. initial nodes and processes. Finally, we shall stipulate that a process can hold at most one token at any given time, since we do not allow the instantiation of multiple copies of a process at the same time. This may sound overly restrictive especially when compared with ADs, but in the context of automated manufacturing, we should note that a physical plant generally does not provide multiple instances. Recall the drill station example in Figure 8 and consider the process GetObject, which could be relatively complicated involving motion control of a robot arm and other sensor behaviours for workpiece positioning. While GetObject is busy, it is clear that a "second instance" can not be provided if there is no "second copy" of the drill station.

Terminal nodes A node $n \in \mathsf{Nodes}$ with $\mathsf{type}(n) = \mathsf{terminal}$ is a *terminal node* and we write

$$\mathsf{TerminalNodes} := \{\, n \in \mathsf{Nodes} \mid \mathsf{type}(n) = \mathsf{terminal} \,\} \tag{7}$$

for the set of all terminal nodes in an SBD. A terminal node has exactly one predecessor, no successors and eliminates any token it receives immediately. Readers being familiar with UML may have discovered that terminal nodes are comparable with *flow final nodes* in ADs in that eliminating a token does not influence other tokens in the SBD, i.e. the execution of the SBD shall proceed if there are still remaining tokens. When all tokens in one SBD are eliminated, we say that this SBD is finished.

Forks and joins A node $n \in \mathsf{Nodes}$ with $\mathsf{type}(n) = \mathsf{fork}$ is a *fork* while a node $n' \in \mathsf{Nodes}$ with $\mathsf{type}(n') = \mathsf{join}$ is a *join*. We write

$$\mathsf{Forks} := \{\, n \in \mathsf{Nodes} \mid \mathsf{type}(n) = \mathsf{fork} \,\}; \tag{8}$$

$$\mathsf{Joins} := \{\, n \in \mathsf{Nodes} \mid \mathsf{type}(n) = \mathsf{join} \,\} \tag{9}$$

for the set of all forks and joins in an SBD, respectively. A fork has exactly one predecessor as well as at least two successors and describes the simultaneous beginning of concurrent processes. Thus, when taking a token from its predecessor, the received token is duplicated to match the number of successors. Note that a fork is not able to hold a token for a positive duration of physical time, which indicates that the duplicated nodes must be instantaneously propagated to the successors. Therefore, a fork taking a token implicitly requires that *each* successor must be ready to receive a token. On the other hand, a join represents the simultaneous termination of concurrent processes and has at least two predecessors as well as exactly one successor. As the counterpart of forks, a join receives tokens from all its predecessors and assembles them into a single token, which is instantaneously propagated to its successor afterwards. A join cannot hold a token for a positive duration of physical time either. Thus, each predecessor of a join $n \in \mathsf{Joins}$ can send its token only if (i) all other predecessors of n are ready to send a token and (ii) the successor of n can receive a token.

Branches and merges A node $n \in \mathsf{Nodes}$ with $\mathsf{type}(n) = \mathsf{branch}$ is a *branch* while a node $n' \in \mathsf{Nodes}$ with $\mathsf{type}(n') = \mathsf{merge}$ is a *merge*. We write

$$\mathsf{Branches} := \{\, n \in \mathsf{Nodes} \mid \mathsf{type}(n) = \mathsf{branch} \,\}; \tag{10}$$

$$\mathsf{Merges} := \{\, n \in \mathsf{Nodes} \mid \mathsf{type}(n) = \mathsf{merge} \,\} \tag{11}$$

for the set of all forks and joins, respectively. A branch has exactly one predecessor as well as at least two successors and represents the choice of alternative processes. Upon receiving a token from its predecessor, a (possibly non-deterministic) choice of successor is taken, to which the token is instantaneously propagated. The non-determinism can be resolved by assigning disjunct $\mathsf{branchconditions}$ on each outgoing edge, which will be discussed in detailed in Section 2.1.3. Besides, similar to forks, since a branch cannot hold a token for a positive duration of physical time, a branch can only take a token if the chosen successor is ready to take a token. On the other hand, a merge denotes the termination of alternative processes and has at least two predecessors as well as exactly one successor. If some predecessor $n' \in \mathsf{Nodes}$ of a merge $n \in \mathsf{Merges}$ is ready for token propagation and the successor of n is ready to receive a token, the token in n' is instantaneously propagated to the successor of n.

With the intended usage of each type of nodes as discussed above, it is convenient for us to define the *syntactical well-formedness* of an SBD. In the remainder, we assume that all SBDs are syntactically well-formed. For convenience, we say a sequence of nodes $n_0 n_1 \dots n_k$, $k \geq 1$ where $n_{i+1} \in \mathsf{suc}(n_i)$ for all $i \in \{0, \dots k-1\}$ is an *instant node sequence* if none of the nodes in this sequence is an initial node, a process or a terminal node.

Definition 2.1.3. *An SBD $S = \langle \mathsf{Nodes}, \mathsf{Edges}, \mathsf{type} \rangle$ is* syntactically well-formed *if and only if all the following conditions hold:*

(W1) for any node $n \in \mathsf{Nodes}$,

(i) if $\mathsf{type}(n) = \mathsf{initial}$, then $\mathsf{pre}(n) = \emptyset$ and $|\mathsf{suc}(n)| = 1$;

(ii) *if* $\mathsf{type}(n) = \mathsf{process}$, *then* $|\mathsf{pre}(n)| = |\mathsf{suc}(n)| = 1$;

(iii) *if* $\mathsf{type}(n) = \mathsf{terminal}$, *then* $|\mathsf{pre}(n)| = 1$ *and* $\mathsf{suc}(n) = \emptyset$;

(iv) *if* $\mathsf{type}(n) \in \{\mathsf{fork}, \mathsf{branch}\}$, *then* $|\mathsf{pre}(n)| = 1$ *and* $|\mathsf{suc}(n)| > 1$;

(v) *if* $\mathsf{type}(n) \in \{\mathsf{join}, \mathsf{merge}\}$, *then* $|\mathsf{pre}(n)| > 1$ *and* $|\mathsf{suc}(n)| = 1$;

(W2) $|\mathsf{InitialNodes}| \geq 1$;

(W3) *for any instant node sequence* $n_0 n_1 \dots n_k$, *if* $n_0 \in \mathsf{Forks}$, *then* $n_k \notin \mathsf{Joins}$;

(W4) *for any instant node sequence* $n_0 n_1 \dots n_k$, $n_0 \neq n_k$;

(W5) *for any two instant node sequences* $n_0 n_1 \dots n_k$ *and* $n_0 n'_1 \dots n'_{k'}$ *where* $n_0 \in \mathsf{Branches}$ *and* $n_1 \neq n'_1$, *there does not exist any* $n'' \in \mathsf{Joins}$ *so that* n'' *is in both sequences.*

While conditions (W1) and (W2) are relatively straightforward from intuition, we briefly explain (W3)–(W5) which specify the structure of instant node sequences. (W3) prescribes that a join shall never be reached from a fork without visiting any process, otherwise an empty but instantaneous concurrent execution is present, which shall be considered spurious. This condition is inspired by (R. Eshuis, 2006, Transformation rule 2). Furthermore, (W4) disallows any instant node sequence to be cyclic, which prevents indefinite instantaneous cycling of token propagation. Finally, (W5) requires that for any two instant node sequences beginning at the same branch but with different successor choices, they shall not be able to instantaneously reach the same join, as token propagation through such a join can never take place.

2.1.2 Formal semantics

2.1.2.1 Single SBD

In this subsection, we focus on formalising SBD semantics and begin with the case where only one SBD is involved. As SBDs are syntactically comparable with ADs, existing literature addressing the semantics formalisation problem for ADs are great references. One common approach to formalising AD semantics is to consider ADs as Petri-nets with extended semantic features; see e.g. (H. Eshuis, 2002; R. Eshuis and Wieringa, 2003; Störrle, 2004). Recall briefly that a Petri-net is a bipartite graph with two disjunct vertex sets, i.e. the set of *places* and the set of *transitions*, and a set of directed *edges* so that each edge either connects a place to a transition or vice versa. Most prominently, only places are able to hold tokens, while *firing transitions*, i.e. propagate tokens from places via transitions in the edge directions to further places, is instantaneous.[1] In this context, it is worth mentioning that, modelling processes (or *activities* in ADs) as places or transition in Petri-nets are both valid semantic interpretations. The former interpretation which is studied in (H. Eshuis, 2002; R. Eshuis, 2006) follows the UML 1.5 specification where ADs are considered as extended UML State Machines (SMs), which is again derived from statecharts (Harel and Naamad, 1996). However, this is not the case in UML $2.x$ where SM and AD are semantically separated from each other. An activity in UML $2.x$ is loosely considered as the sequencing of instantaneous actions, and thus is naturally considered as a transition in Petri-net; see e.g. (Störrle, 2004). Although the latter one is often considered as closer to Petri-net semantics (R. Eshuis and Wieringa, 2003), the former approach is more preferable for our modelling requirement in that each process represents a programme block and programme execution can consume a positive duration of physical time. Thus, control instructions specified in a process do not need to be instantaneous. Thus, we introduce the notation

$$\mathsf{Places} := \mathsf{InitialNodes} \cup \mathsf{Processes} \subseteq \mathsf{Nodes} \tag{12}$$

to denote the set of nodes which correspond to places in Petri-nets.

Conventionally, the dynamic behaviour of a Petri-net is characterised by the change of token distribution over places, which is caused by token propagation. This intuition is generally followed by SBD dynamics as well. To this end, a careful explanation of the time model used by SBD dynamics is demanded for a faithful formalisation. For a great number of physical systems, time is usually described on the non-negative real time axis $\mathbb{R}_0^+$ so that continuous dynamics can be expressed appropriately. However, in this case, it is awkward to express multiple instantaneous transitions which are ordered in a specific sequence. This motivates the application of the two-dimensional time axis $\mathbb{R}_0^+ \times \mathbb{N}_0$ where in addition to the ordinary continuous dynamics (i.e. the "horizontal axis"), instantaneous events can be finitely "vertically" stacked. This is referred to as *dense time* in e.g. (Eker, Janneck, Lee, J. Liu et al., 2003) and the resulting dynamic behaviour is considered *hybrid* (Tabuada, 2009). For SBDs, we assume that continuous dynamics is not considered, which allows us to simplify the dense time model to $\mathbb{N}_0 \times \mathbb{N}_0$. Furthermore, we can in fact utilise $\mathbb{N}_0$ as our time model, which is often referred to as the *logic time*, on which the physical time duration between two points ranges over $\mathbb{R}_0^+$. By

[1] Note that this is true in most timed Petri-nets as well. In such cases, once a transition becomes enabled, it can actually be fired only after a non-negative duration of physical time. While "waiting" for firing, tokens enabling the transition still stay in original places and firing transitions is instantaneous; see e.g. (Cassandras and Lafortune, 2008).

keeping the two-dimensional time axis $\mathbb{N}_0 \times \mathbb{N}_0$ in mind, the terminology *elapse of physical time* is utilised to denote the progress in the horizontal time axis, i.e. in the physical time. Thus, we utilise $\iota \in \mathbb{N}_0$ in the following to denote a concrete "time point" on the logic time axis.

Based on the logic time axis, we define the token distribution over places, i.e. the *configuration* of an SBD, as a *semantic variable*[2] $\mathsf{Marking}$. Recall that each place in an SBD can hold at most one token. Thus, it is convenient to let $\mathsf{Marking}$ directly range over subsets of Places, i.e. utilise

$$\mathsf{Marking}(\iota) \subseteq \mathsf{Places} \tag{13}$$

to map a time instance $\iota \in \mathbb{N}_0$ to a subset of Places.

If token propagation is possible at some time instance ι, the token in $n \in \mathsf{Marking}(\iota)$ is instantaneously propagated to $\mathsf{suc}(n)$ (note that n as a place has only one successor). However, if $\mathsf{suc}(n) \cap \mathsf{Places} = \emptyset$, further instantaneous propagations shall be taken, until there are no tokens left in any non-place node. From (W4), a series of instantaneous propagations from one configuration to the successive one always takes finite steps, i.e. only a finite number of edges will be visited. Token propagation through such a finite edge sequence, which is referred to as a *hyper-edge* as suggested in (R. Eshuis, 2006), does not consume any physical time and is semantically mapped to a Petri-net transition. We write HEs to denote the set of all hyper-edges included in an SBD. For any $h \in \mathsf{HEs}$, it is convenient to define

$$\mathsf{Sources}(h) \subseteq \mathsf{Places} \quad \text{and} \quad \mathsf{Targets}(h) \subseteq \mathsf{Places} \tag{14}$$

to access its *sources*, from which the tokens are propagated by firing h, and *targets*, which obtain tokens after firing h, respectively. Note that an hyper-edge must have at least one source, but may have no outputs due to token elimination at terminal nodes, which is a possible situation in Petri-nets as well. In addition, another type of information a hyper-edge may carry is its *branch choices*, i.e.

$$\mathsf{BranchChoices}(h) \subseteq \{\, (n, n') \mid n \in \mathsf{Branches}, n' \in \mathsf{suc}(n) \,\} =: \mathsf{BranchChoices}, \tag{15}$$

[2] A *semantic* variable is referred to as a variable utilised to formulate SBD semantics. Semantic variables shall not be confused with *system variables* (or concisely *variables*) later on, which are utilised to e.g. formulate conditions.

which record all branches h passes through and the corresponding successor choice at each such branch. Since each hyper-edge only represent deterministic branch choice, it holds that for each $h \in \mathsf{HEs}$, we must have

$$\forall (n, n'), (m, m') \in \mathsf{Branchchoices}(h).\ n = m \quad \Rightarrow \quad n' = m'. \tag{16}$$

Since a hyper-edge may pass through multiple different kinds of non-place nodes, it is not trivial to compute HEs of a given SBD. In the following, we show an algorithm which constructs a single hyper-edge h from a given place $n_0 \in \mathsf{Places}$ so that $n_0 \in \mathsf{Sources}(h)$. The pseudo-codes of the algorithm are given in Algorithm 1. Note that since only one hyper-edge is constructed from n_0 which may pass through branches and merges, consistent choices for each branch to one of its successors as well for each merge back to one of its predecessors are necessary for multi-step searches. These are denoted by $\beta : \mathsf{Branches} \to \mathsf{Nodes}$ and $\gamma : \mathsf{Merges} \to \mathsf{Nodes}$ as global parameters of Algorithm 1, respectively, where we naturally require that

$$\forall n \in \mathsf{Branches}.\ \beta(n) \in \mathsf{suc}(n)\,; \tag{17}$$
$$\forall n' \in \mathsf{Merges}.\ \gamma(n') \in \mathsf{pre}(n') \tag{18}$$

must hold. We now explain Algorithm 1 as follows.

HyperEdgeConstruction(n_0) This function constitutes the main function of the algorithm which consists of a recursive breadth-first forward search ForwardSearch for the given seed node n_0 and a recursive breadth-first backward search BackwardSearch if any join is visited during the forward search. To represent the resulting hyper-edge h where $n_0 \in \mathsf{Sources}(h)$, this function returns all sources and targets of the hyper-edge as well as all involved branch choices.

ForwardSearch(N, N_b) This function recursively search successors from the input node set $N \subseteq \mathsf{Nodes}$. To encode the successor choice induced by β, we introduce a restricted successor map suc_β which is defined by

$$\mathsf{suc}_\beta(n) = \begin{cases} \mathsf{suc}(n) & \text{if } n \in \mathsf{Nodes} - \mathsf{Branches}\,; \\ \{\beta(n)\} & \text{if } n \in \mathsf{Branches}\,. \end{cases} \tag{19}$$

For each $n \in N$, all non-place non-terminal successors are recorded in a set *sucs* $\subseteq \mathsf{Nodes} - \mathsf{Places} - \mathsf{Terminals}$ for the next iteration, while all place successors are recorded in the result *targets* $\subseteq \mathsf{Places}$. Note that if a successor is a join, its unvisited non-place predecessors are recorded in

Algorithm 1 Hyper-edge construction

```
    global: sources          ▷ result to return, initialised to empty set
    global: targets          ▷ result to return, initialised to empty set
    global: branchChoices    ▷ result to return, initialised to empty set
    global: $\beta$          ▷ pre-defined successor choice of each branch
    global: $\gamma$         ▷ pre-defined predecessor choice of each merge
 1: function HyperEdgeConstuction($n_0$)
 2:     sources $\leftarrow \{n_0\}$
 3:     back $\leftarrow$ ForwardSearch($\{n_0\}, \emptyset$)
 4:     BackwardSearch(back)
 5:     return sources, targets, branchChoices
 6: end function

 7: function ForwardSearch($N$, $N_b$)
 8:     sucs $\leftarrow \emptyset$          ▷ (one-step) successors for the next iteration
 9:     for all $n \in N$ do
10:         sucs $\leftarrow$ sucs $\cup$ ($\mathsf{suc}_\beta(n) - (\mathsf{Places} \cup \mathsf{Terminals})$)
11:         targets $\leftarrow$ targets $\cup$ ($\mathsf{suc}_\beta(n) \cap \mathsf{Places}$)
12:         if $n \in \mathsf{Branches}$ then
13:             branchChoices $\leftarrow$ branchChoices $\cup$ $(n, \mathsf{suc}_\beta(n))$      ▷ $suc_\beta(n)$
    has only one element
14:         end if
15:         for all $n_s \in$ sucs $\cap$ Joins do          ▷ record other joined nodes
16:             $N_b \leftarrow N_b \cup (\mathsf{pre}(n_s) - \{n\} - \mathsf{Places})$     ▷ non-place pred. for
    bw. search
17:             sources $\leftarrow$ sources $\cup$ ($\mathsf{pre}(n_s) \cap \mathsf{Places}$)
18:         end for
19:     end for
20:     if sucs $\neq \emptyset$ then
21:         $N_b \leftarrow N_b \cup$ ForwardSearch(sucs, $N_b$) ▷ collect all joined nodes
    recursively
22:     end if
23:     return $N_b$
24: end function

25: function BackwardSearch($N$)
26:     pres $\leftarrow \emptyset$          ▷ (one-step) predecessors for the next iteration
27:     for all $n \in N$ do
28:         pres $\leftarrow$ pres $\cup$ ($\mathsf{pre}_\gamma(n) - \mathsf{Places}$)
29:         sources $\leftarrow$ sources $\cup$ ($\mathsf{pre}_\gamma(n) \cap \mathsf{Places}$)
```

```
30:         if $n \in \mathsf{Branches}$ then
31:             $branchChoices \leftarrow branchChoices \cup (n, \mathsf{pre}(n))$     ▷ a branch has only one pred.
32:         end if
33:     end for
34:     if $pres \neq \emptyset$ then
35:         BackwardSearch($pres$)
36:     end if
37: end function
```

$N_b \subseteq \mathsf{Nodes}$ – Places for backward search. It is worth mentioning that due to (W4), no non-place node can be reached twice and the recursion is guaranteed to terminate.

BACKWARDSEARCH(N) Since the forward search may visit joins, a backward search for non-place predecessors at each join is necessary. Similar to suc_β, a restricted predecessor map pre_γ is defined by

$$\mathsf{pre}_\gamma(n) = \begin{cases} \mathsf{pre}(n) & \text{if } n \in \mathsf{Nodes} - \mathsf{Merges}\,; \\ \{\gamma(n)\} & \text{if } n \in \mathsf{Merges}\,. \end{cases} \tag{20}$$

to encode the predefined merge predecessor choice. Note that whenever a branch is visited during the backward search, its unique predecessor and itself is directly recorded in the branch choice *regardless of* β. Also note that if the backward search is performed, it can never reach a fork without visiting a process beforehand due to (W3). Thus, the forward search does not need to be performed anew. In addition, (W5) guarantees that all branches visited during the backward search shall not have appeared in the forward search. Hence, ambiguous branch choices can always be avoided, as required in (16).

Based on applying Algorithm 1 for each place $n \in \mathsf{Places}$, each possible branch successor configuration β and merge predecessor configuration γ, the set of all hyper-edges of an SBD can be determined. Note that since each place can be a source of only one hyper-edge under given β and γ, places which already belong to sources of some constructed hyper-edges can be skipped. Furthermore, enumerating β and γ can be recursively implemented by forking the computation when any new branch or new merge is detected during recursion, respectively.

Recall that SBD dynamics is organised by *firing* hyper-edges. Basically, a hyper-edge can be fired only if it is *enabled*. Since enabledness of a hyper-edge is related to the process states of its sources and targets, we first define a semantic variable $\mathsf{ProcessState}_n$ for each process $n \in \mathsf{Processes}$ to map each time instance ι to one of the process states, i.e.

$$\mathsf{ProcessState}_n(\iota) \in \mathsf{ProcessStates}\,, \tag{21}$$

based on which the definition of an enabled hyper-edge is given as follows.

Definition 2.1.4. *A hyper-edge $h \in \mathsf{HEs}$ is* enabled at time ι *if and only if all the following conditions hold:*

(E1) $\mathsf{Sources}(h) \subseteq \mathsf{Marking}(\iota)$*;*

(E2) $\forall n \in \mathsf{Sources}(h) \cap \mathsf{Processes}.\ \mathsf{ProcessState}_n(\iota) = \mathsf{done}$*;*

(E3) $\forall n \in \mathsf{Targets}(h).\ n \notin \mathsf{Marking}(\iota) - \mathsf{Sources}(h)$*;*

(E4) the guard condition associated with h evaluates true *at ι.*

In Definition 2.1.4, (E1) requires that all sources of the considered hyper-edge are currently in the configuration. In addition, (E2) further requires that all processes in the sources must be in the process state done. (E3) implies that all targets of the considered hyper-edge must be in the process state idle, except when a target is also a source of the same hyper-edge. Finally, we temporarily assume that (E4) always holds. A detailed discussion of conditions will be presented in Section 2.1.3. For convenience, we define the semantic variable Enabled to denote the set of enabled hyper-edges at some time instance ι, i.e.

$$\mathsf{Enabled}(\iota) \subseteq \mathsf{HEs}\,. \tag{22}$$

In the following, we characterise the SBD dynamics by describing the *individual update* of configuration as well process states of all processes at $\iota + 1$ utilising the information at ι. Note that the complete dynamic behaviour is based on updating SBD from proper *initialisation* of the SBD, which can only be faithfully described considering how SBDs are nested with each other within an SBD project; see Section 2.1.2.2.

Definition 2.1.5. *The* individual update *of an SBD S is defined by the following equations:*

(i) *if* $\mathsf{Enabled}(\iota) \neq \emptyset$*, then by picking any* $h \in \mathsf{Enabled}(\iota)$ *and fire* h*, the configuration is updated by*

$$\mathsf{Marking}(\iota+1) = (\mathsf{Marking}(\iota) - \mathsf{Sources}(h)) \cup \mathsf{Targets}(h), \tag{23}$$

and the process state of each process $n \in \mathsf{Processes}$ *is updated by the following equations:*
if $n \in \mathsf{Sources}(h) \cup \mathsf{Targets}(h)$*, then*

$$\mathsf{ProcessState}_n(\iota+1) = \begin{cases} \mathsf{idle} & \textit{if } n \in \mathsf{Sources}(h) - \mathsf{Targets}(h); \\ \mathsf{busy} & \textit{if } n \in \mathsf{Targets}(h); \end{cases} \tag{24}$$

otherwise,

$$\mathsf{ProcessState}_n(\iota+1) = \begin{cases} \mathsf{idle} & \textit{if } \mathsf{ProcccessState}_n(\iota) = \mathsf{idle}; \\ \mathsf{busy} \textit{ or } \mathsf{done} & \textit{if } \mathsf{ProcccessState}_n(\iota) = \mathsf{busy}; \\ \mathsf{done} & \textit{if } \mathsf{ProcccessState}_n(\iota) = \mathsf{done}; \end{cases} \tag{25}$$

(ii) *otherwise, i.e. if* $\mathsf{Enabled}(\iota) = \emptyset$*, then the configuration is updated by*

$$\mathsf{Marking}(\iota+1) = \mathsf{Marking}(\iota) \tag{26}$$

and the process state of each process is updated according to (25).

Generally, the individual update is defined as such that if some hyper-edges is enabled, one of them *must* be fired, causing a configuration update by removing tokens from all sources and then providing tokens to all targets. Besides, the process state of each process is updated according the idle–busy–done cycle. At the current stage, the following two points w.r.t. Definition 2.1.5 are worth mentioning:

- In the second case of (25), a process $n \in \mathsf{Processes}$ which is irrelevant to the picked enabled hyper-edge and is in the process state busy at ι have two possible process states at $\iota + 1$, namely, either still busy or done. This corresponds to two possible situations: if n does not invoke any other SBD, evolving from busy to done is spontaneous due to the black-box mechanism of processes; otherwise, i.e. n invokes some SBD T, n evolves from busy to done if and only if T becomes finished. The latter case is addressed in detail in Definition 2.1.8.

- Since an enabled hyper-edge h is arbitrarily picked from all currently enabled hyper-edges, non-deterministic behaviour may emerge when multiple simultaneously enabled hyper-edges share some sources. In other words, if two enabled hyper-edges visit the same branch with different branch choices, firing one hyper-edge may disable the other. This is often referred to as *conflict*; see e.g. (R. Eshuis, 2006). Nevertheless, in practice (especially in automated manufacturing), deterministic behaviour of control programmes is often desired. As for SBDs, this can be interpreted as such that different orders of firing simultaneously enabled hyper-edges should lead to the same configuration. This can be achieved by exclusive branch conditions, which will be introduced in detail in Section 2.1.3.

Finally, we recall that a token can not be propagated into a process which currently holds a token, which is also stated in Definition 2.1.4. We encode this semantic restriction into *semantic well-formedness*, or concisely *well-formedness*, as follows, which is assumed for all SBDs in the remainder.

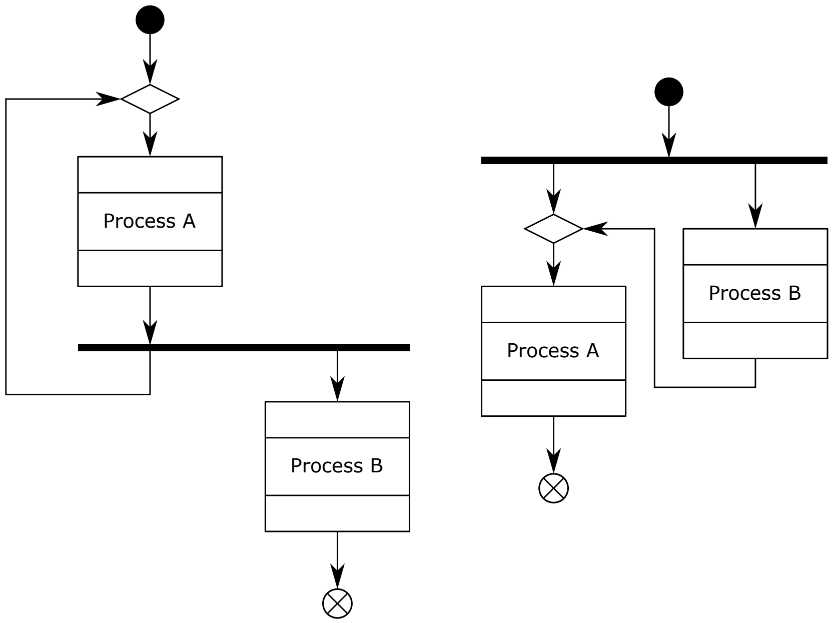

Figure 9: Example SBDs with concurrent processes that are not properly joined

Definition 2.1.6. *A (syntactically well-formed) SBD S is* semantically well-formed *if and only if*

(W6) for all $h \in \mathsf{Enabled}(\iota)$ at any time instance ι, it holds that

$$\mathsf{Sources}(h) \subseteq \mathsf{Marking}(\iota) \quad \rightarrow \quad (\mathsf{Marking}(\iota) - \mathsf{Sources}(h)) \cap \mathsf{Targets}(h) = \emptyset\,. \tag{27}$$

One typical situation for a syntactically well-formed SBD being not (semantically) well-formed is such that multiple tokens generated by a fork or initial nodes are improperly merged, e.g. as depicted in Figure 9. In fact, in other programming languages with similar Petri-net-like structure, e.g. Sequential Function Charts (SFCs) as defined in IEC $61131-3$ standard, structures in Figure 9 are considered illegal as well. In practice, more restrictive syntactic rules are often preferred where concurrent (or alternative) processes initialised by a join (or a branch) must be terminated by a fork (or a merge). This kind of restriction is adopted by e.g. Siemens-GRAPH, which is a programming language derived from SFC.

2.1.2.2 Nested SBDs

Based on the semantics of each individual SBD, the global behaviour of a complete SBD project can be formalised by considering parallel execution (modularity) and invocations (hierarchy) between SBDs. Recall that the tuple of an SBD $S \in \mathsf{SBDP}$ includes the invocation function invoke_S. For any $n \in \mathsf{Processes}_S$, $\mathsf{invoke}_S(n) \in \mathsf{SBDP}$ indicates that n does invoke an SBD, in which case n is referred to as an *invoker*. Contrarily, if $\mathsf{invoke}_S(n) = \emptyset$, then the process n is *atomic* and its behaviour is not specified by any SBD. Correspondingly, we utilise[3]

$$\mathsf{invokedBy}\colon \mathsf{SBDP} \rightarrow 2^{\mathsf{Processes}^{\mathsf{GL}}}\,, \tag{28}$$

with

$$\mathsf{Processes}^{\mathsf{GL}} := \dot{\bigcup}_{S\in\mathsf{SBDP}} \mathsf{Processes}_S \tag{29}$$

to denote the set-valued inverse of invoke, i.e. to match an SBD S to all its invoker processes globally. For any SBD S so that $\mathsf{invokedBy}(S) = \emptyset$, it is not invoked by any process and thus is referred to as a *root*. Note that an SBD project may contain multiple root SBDs, which are operated in parallel in that they are initialised simultaneously; see Definition 2.1.7 below. Also note that henceforth, the convenient notation $(\cdot)^{\mathsf{GL}}$ is utilised to union sets of

[3] 2^X denotes the power set of a set X.

components in all SBDs of the SBD project with uniform component type, which also applies to semantic variables, e.g. we utilise

$$\mathsf{Marking}^{\mathsf{GL}}(\iota) := \dot{\bigcup}_{S\in\mathsf{SBDP}}\mathsf{Marking}_S(\iota)\,. \tag{30}$$

to denote the *global configuration* at time ι.

Based on the individual update of each SBD, the dynamics of an complete SBD project can be represented by the initialisation and update of the global configuration and the process state of each process. Initialisation of an SBD project occurs at $\iota = 0$. Upon initialisation, globally all processes are set to the process state idle, while in each root SBD, each initial node obtains one token.

Definition 2.1.7. *The* initialisation *of an SBD project* SBDP *is defined by the following equations:*
For all $S \in \mathsf{SBDP}$,

$$\mathsf{Marking}_S(0) = \begin{cases} \mathsf{InitialNodes}_S & \textit{if } \mathsf{invokedBy}(S) = \emptyset\,; \\ \emptyset & \textit{if } \mathsf{invokedBy}(S) \neq \emptyset\,. \end{cases} \tag{31}$$

For all $n \in \mathsf{Processes}^{\mathsf{GL}}$,

$$\mathsf{ProcessState}_n(0) = \mathsf{idle}. \tag{32}$$

As for the updates, special care should be taken for each invoker process and the SBD it invokes. Unlike atomic processes, the behaviour of an invoker is specified by an SBD, which implies that the process state cycle is related with its invoked SBD. This motivates us to adapt the *CallBehaviorAction* defined in ADs to interpret the semantics of SBD invocation; namely, a invoker is in the process state busy when the invoked SBD is under execution, and sent to the process state done when the invoked SBD is finished. This design choice is also similar to that of *macro steps* in Grafcet (Provost, J.-M. Roussel et al., 2011).

Definition 2.1.8. *The* update *of an SBD project* SBDP *is defined by the conjunction of individual updates of each SBD* $S \in \mathsf{SBDP}$ *and the following equations:*

(i) *Let* $h \in \mathsf{HEs}^{\mathsf{GL}}$ *be enabled and fired at time instance* ι. *For all* $n \in \mathsf{Targets}(h)$ *so that* $\mathsf{invoke}(n) = T \in \mathsf{SBDP}$, *it holds that*

$$\mathsf{Marking}_T(\iota + 1) = \mathsf{InitialNodes}_T\,; \tag{33}$$

(ii) *At any time instance ι, for all $n \in \mathsf{Processes}^{\mathsf{GL}}$ so that $\mathsf{invoke}(n) = T \in \mathsf{SBDP}$, it holds that*

$$\mathsf{Marking}_T(\iota) \neq \emptyset \;\Rightarrow\; \mathsf{ProcessState}_n(\iota) = \mathsf{busy}\,; \tag{34}$$

$$\mathsf{Marking}_T(\iota) = \emptyset \;\Rightarrow\; \mathsf{ProcessState}_n(\iota) \in \{\mathsf{done}, \mathsf{idle}\}. \tag{35}$$

With Definitions 2.1.5, 2.1.7 and 2.1.8, all possible trajectories of $\mathsf{Marking}^{\mathsf{GL}}(\iota)$ and $\mathsf{ProcessState}_n(\iota)$ for all $n \in \mathsf{Processes}^{\mathsf{GL}}$ of a given SBD project can be described, which can further be represented by automata. At the end of this paragraph, we recall that instantiating multiple copies of any process is considered illegal. As for nested SBDs, this indicates invoking a non-root SBD T is not allowed when T has already been invoked and is still unfinished. In addition, we syntactically disallow cyclic invocation structure. For an SBD S with a invoker process $n \in \mathsf{Processes}_S$, allowing processes in $\mathsf{invoke}(n)$ to invoke S is with little practical value. Hence, we introduce the *well-formedness of an SBD project*. For convenience, we introduce the term *invocation sequence* to denote a finitely concatenated sequence of SBDs $S_0 S_1 \dots S_k$ where $k \geq 1$ and for each S_i and S_{i+1}, there exists some $n \in \mathsf{Processes}_{S_i}$ so that $\mathsf{invoke}(n) = S_{i+1}$.

Definition 2.1.9. *An SBD project* SBDP *is* well-formed *if and only if*

(WP1) *for any time instance ι, it holds that*

$$\forall n, n' \in \mathsf{Marking}^{\mathsf{GL}}(\iota).\, n \neq n' \;\wedge\; \mathsf{invoke}(n) \in \mathsf{SBDP} \;\wedge\; \mathsf{invoke}(n') \in \mathsf{SBDP} \;\rightarrow\; \mathsf{invoke}(n) \neq \mathsf{invoke}(n')\,; \tag{36}$$

(WP2) *for any invocation sequence $S_0 S_1 \dots S_k$, it holds that $S_0 \neq S_k$.*

In the remainder, well-formedness is assumed for all SBD projects.

2.1.3 Conditions and variables

In this section, the long awaited definition of the substructure Guards of each SBD tuple is revealed. Recall from Definition 2.1.4 that a hyper-edge is always associated with a *guard condition*, which is defined within Guards. We first provide the formal definition of Guards as follows.

Definition 2.1.10. *The* guards *of an SBD is a tuple* $\mathsf{Guards} := \langle \mathsf{Variables}, \mathsf{precond}, \mathsf{postcond}, \mathsf{branchcond}, \mathsf{Initcond} \rangle$ *where*

- Variables *is a set of* system variables*, or concisely* variables*;*

- $\mathsf{precond} : \mathsf{Processes} \to \mathsf{Conditions}$ *is the precondition assignment function where*
  $\mathsf{Conditions}$ *denotes the set of all legit propositions formulated by* $\mathsf{Variables}$*;*
- $\mathsf{postcond} : \mathsf{Processes} \to \mathsf{Conditions}$ *is the postcondition assignment function;*
- $\mathsf{branchcond} : \mathsf{BranchChoices} \to \mathsf{Conditions}$ *is the branch-condition assignment function;*
- $\mathsf{Initcond} \in \mathsf{Conditions}$ *is the initial condition.*

Each SBD has a set Variables of (system) variables which are manipulated by processes (e.g. for controlling actuator) and/or describe the plant status (e.g. by reading sensor line levels). In additions, variables can be utilised to construct various conditions to guard token propagations. Technically, a condition is a mapping from some expression based on variable evaluation at some *physical time instance* to a boolean value,[4] where we particularly require the trivial condition true must be a valid condition, i.e. $\mathsf{true} \in \mathsf{Condition}$ must hold. Since continuous dynamics is not considered, for each variable $v \in \mathsf{Variables}$, a *finite* set of values is defined which is denoted by $\mathsf{range}(v)$, from which a *value* is taken by v at each discrete *physical* time instance and we write

$$v(\iota) \in \mathsf{range}(v). \tag{37}$$

Note that ι in (37) denotes a time instance on the logic time axis, which encodes the (discrete) physical time axis as well. However, in order to faithfully illustrate the cooperative relation between token propagation and variable evaluation, the two-dimensional dense time axis $\mathbb{N}_0 \times \mathbb{N}_0$ is essential. Semantically, we require that if a hyper-edge is fireable, i.e. enabled and actually chosen if conflict among enabled hyper-edges exists, then it must be fired immediately. This semantic assumption is widely adopted in various Petri-net-like modelling languages; see e.g. (R. Eshuis, 2006; Object Management Group, 2017b; Provost, J.-M. Roussel et al., 2011), since as soon as the guard condition associated with a transition evaluates true, firing this transition shall never be delayed as such that its guard condition is again invalidated. Thus, in the context of SBDs, fireable hyper-edges are stacked vertically on a physical time instance. More importantly, value changes of some explicit variable can

[4] At the current stage, we do not explicitly require the form a condition expression should take. In fact, when translating SBDs to automata in Section 2.2, it is expected that each atomic element forming a condition in Conditions, i.e. an atomic proposition, always takes the form of an equality proposition, e.g. $\mathsf{position} = \mathsf{west}$ as in the drill station example. Extending expressions to more general syntax is beyond the scope of the current dissertation.

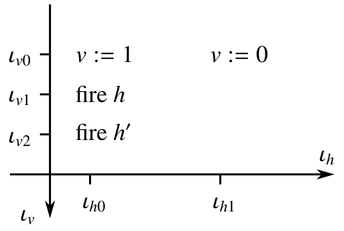

Figure 10: Value change and hyper-edge firing on the two-dimensional logic time axis (the vertical axis is directed from top to bottom to suit the intuition of "the top most event occurs first")

only happen at the top of such stacks, i.e. there must be a minimal positive duration of physical time between value changes. Meanwhile, "inserting" a value change into the middle or bottom of the stack is forbidden.[5] A concise example of this mechanism is illustrated in Figure 10, where we explicitly utilise ι_h and ι_v to separately denote the discrete physical time and vertical instantaneous action stack, respectively. Suppose the value of some variable v changes from 0 to 1 at physical time $\iota_h = \iota_{h0}$. This event is placed at the top of the stack; namely, at $(\iota_h, \iota_v) = (\iota_{h0}, \iota_{v0})$ where ι_{v0} by default denotes the first instance of a stack at any ι_h. Subsequently, hyper-edges h and h' are enabled and instantaneously fired at the same physical time instance. Although the value of v may eventually again change from 1 to 0 at some physical time instance ι_{h1}, it is guaranteed that $\iota_{h1} > \iota_{h0}$.

In the following, we introduce the semantic effect of all types of conditions of an SBD.

Precondition To each process, a precondition is assigned by the function precond. The precondition of a process guards the process state transition from idle to busy, so that it is guaranteed that the process is "correctly started". This implies that a hyper-edge h is enabled at some time instance ι only if the precondition of each $n \in \mathsf{Targets}(h)$ evaluates true at ι.

Postcondition To each process, a postcondition is assigned by the function postcond. The postcondition of a process guards the process state transition from done to idle. Thus, a hyper-edge h is enabled at some time instance ι only if the postcondition of each $n \in \mathsf{Sources} \cap \mathsf{Processes}$ evaluates true at ι. This guarantees that the process is "correctly left". Besides, at any time instance ι so that the process state of some $n \in \mathsf{Processes}$

[5] Note that this holds for all variables, i.e. two variables cannot change their values at exactly the same physical time point.

turns from busy to done, $\mathsf{postcond}(n)$ is guaranteed to evaluate true. Thus, if a process is in the process state done, its postcondition may be invalidated due to e.g. the execution of other processes. In order that the successive hyper-edge of n can be fired, such postcondition invalidation must be temporary. Note that alternatively, we could have also interpreted postconditions as such that the "correct leaving" part is dropped; namely, the process state of a process can evolve from done to idle regardless of its postcondition. Processes with such kind of interpretation can be in fact equivalently modelled by our construct through concatenating a dummy process with trivial pre- and postconditions, which is similar to the so-called *wait node* as suggested in (R. Eshuis, 2006) and the SFC specification in IEC $61131-3$ standard.

Branch condition To each branch choice, a branch condition is assigned by the function branchcond. A hyper-edge h is enabled at some time instance ι only if all branch conditions assigned to branch choices in $\mathsf{BranchChoices}(h)$ evaluate true at ι. In order to guarantee deterministic choice at each branch, branch conditions associated with each branch must be exclusive, which can be verified through syntactical analysis. This can be guaranteed for branches with two successors by simply utilising the keyword else which denotes the complement of the branch condition on the other branch choice.

Initial condition To each SBD, an initial condition Initcond is assigned. An SBD can start execution, i.e. obtain tokens in initial nodes, only if its initial condition evaluates true. In the context of nested SBDs, since the process state of an invoker process n is instantaneously set to busy when the SBD $T = \mathsf{invoke}(n)$ starts operation, a hyper-edge h with $n \in \mathsf{Targets}(h)$ is enabled only if the initial condition of T evaluates true. In fact, the initial condition of an non-root SBD plays the same semantic roll as the precondition of its invoker process. However, when considering translating SBDs into automata, several computational advantages are conceivable when utilising initial conditions since it generally reduces the state space of each SBD containing invokers (since possibly fewer variables are associated with this SBD) and each non-root SBD (since this SBD can only be initialised in restricted cases). Finally, since all root SBDs are directly activated upon the initialisation of the SBD project, it is natural to stipulate that the initial condition of a root SBD must be trivially true.

With all types of conditions explained, the *guard condition* of an hyper-edge which is required in (E4) of Definition 2.1.4 can be formalised as follows.

Definition 2.1.11. *Let* SBDP *be an SBD project. The* guard condition *of an hyper-edge* $h \in \mathsf{HEs}^{\mathsf{GL}}$ *is defined by the condition* $c_h \in \mathsf{Conditions}^{\mathsf{GL}}$ *where*

$$\begin{aligned} c_h \equiv & \left(\bigwedge_{n \in \mathsf{Targets}(h)} \mathsf{precond}(n) \wedge \mathsf{Initcond}_{\mathsf{invoke}(n)} \right) \\ & \wedge \left(\bigwedge_{n \in \mathsf{Sources}(h) \cap \mathsf{Processes}^{\mathsf{GL}}} \mathsf{postcond}(n) \right) \\ & \wedge \left(\bigwedge_{(n,n') \in \mathsf{BranchChoices}(h)} \mathsf{branchcond}(n, n') \right) \end{aligned} \tag{38}$$

where for any $n \in \mathsf{Processes}^{\mathsf{GL}}$ *so that* $\mathsf{invoke}(n) = \emptyset$*, we have* $\mathsf{Initcond}_{\mathsf{invoke}(n)} =$ true.

2.1.4 Operation of the drill station example

With the SBD formal semantics explained, we review the operation of the drill station as given in Figure 8. Recall that each node has a globally unique ID, i.e. $\mathsf{Nodes}^{\mathsf{GL}} \subset \mathbb{N}_0$. In this context, each hyper-edge $h \in \mathsf{HEs}$ is conveniently denoted by a symbolic name which encodes its sources, targets and branch choices, i.e. a hyper-edge $h \in \mathsf{HEs}$ generally takes the form of

$$h \equiv \mathsf{HE}[\mathsf{S}[s_1, \ldots, s_i]\mathsf{C}[b_1 > c_1, \ldots, b_j > c_j]\mathsf{T}[t_1, \ldots t_k]], \tag{39}$$

with $\{s_1, \ldots, s_i\} = \mathsf{Sources}(h)$, $\{(b_1, c_1), \ldots, (b_j, c_j)\} = \mathsf{BranchChoices}(h)$ and $\{t_1, \ldots, t_k\} = \mathsf{Targets}(h)$. The fragment $\mathsf{C}[b_1 > c_1, \ldots, b_j > c_j]$ and/or $\mathsf{T}[t_1, \ldots t_k]$ in (39) is omitted if the involved hyper-edge h does not visit any branch and/or the target set of h is empty, respectively. Considering this nomenclature, the nested SBDs S and T in Figure 8 hold hyper-edges

$$\begin{aligned} \mathsf{HEs}_S \equiv \{ & \mathsf{HE}[\mathsf{S}[10]\mathsf{T}[11]], \\ & \mathsf{HE}[\mathsf{S}[11]] \} \end{aligned} \tag{40}$$

and

$$\begin{aligned} \mathsf{HEs}_T \equiv \{ & \mathsf{HE}[\mathsf{S}[0]\mathsf{T}[1]], \\ & \mathsf{HE}[\mathsf{S}[1]\mathsf{C}[1003>2]\mathsf{T}[2,4]], \\ & \mathsf{HE}[\mathsf{S}[1]\mathsf{C}[1003>3]\mathsf{T}[3,4]], \\ & \mathsf{HE}[\mathsf{S}[2,4]], \\ & \mathsf{HE}[\mathsf{S}[3,4]] \}. \end{aligned} \tag{41}$$

Upon the initialisation of the SBD project $\mathsf{SBDP} = \{S, T\}$, a token is immediately generated in the initial node 10 in S while there is no token in T. By assuming that the initial condition of T is trivial, the hyper-edge $\mathsf{HE}[\mathsf{S}[10]\mathsf{T}[11]]$ becomes enabled as soon as $\mathsf{precond}(11)$, i.e. $\mathsf{button} = \mathsf{pushed}$, evaluates true. Once $\mathsf{HE}[\mathsf{S}[10]\mathsf{T}[11]]$ becomes enabled, it is fired immediately, causing the token in initial node 10 to propagate into process 11 $\mathsf{DrillWP}$. This propagation sends process 11 to the process state busy and since process 11 invokes SBD T (denoted by the symbol $\pitchfork$), a token is generated in the initial node 0 in T at the same time. At this stage, if $\mathsf{precond}(1)$ (i.e. $\mathsf{position} = \mathsf{west}$) evaluates true, the only subsequently fireable hyper-edge $\mathsf{HE}[\mathsf{S}[0]\mathsf{T}[1]]$ is instantaneously fired. This sends process 1 to the process state busy and the programme codes in process 1 are executed. Upon the termination of process 1, it is switched to the process state done and its postcondition $\mathsf{position} = \mathsf{south}$ must evaluate true. At this stage, two hyper-edges are possible to be fired subsequently, i.e. $\mathsf{HE}[\mathsf{S}[1]\mathsf{C}[1003{>}2]\mathsf{T}[2,4]]$ and $\mathsf{HE}[\mathsf{S}[1]\mathsf{C}[1003{>}3]\mathsf{T}[3,4]]$. Firing either hyper-edge requires $\mathsf{postcond}(1)$ to be true. In addition, firing $\mathsf{HE}[\mathsf{S}[1]\mathsf{C}[1003{>}2]\mathsf{T}[2,4]]$ requires that $\mathsf{branchcond}(1003, 2)$, i.e. $\mathsf{wptype} = \mathsf{a}$ evaluates true, while firing $\mathsf{HE}[\mathsf{S}[1]\mathsf{C}[1003{>}3]\mathsf{T}[3,4]]$ requires that $\mathsf{branchcond}$ $(1003, 3)$ evaluates true. For instance, we pick $\mathsf{HE}[\mathsf{S}[1]\mathsf{C}[1003{>}2]\mathsf{T}[2,4]]$ to fire. This sends the configuration of T from $\{1\}$ to $\{2, 4\}$, i.e. drilling starts and the ventilator for dust removal is turned on parallelly. At this stage, the next fireable hyper-edge is $\mathsf{HE}[\mathsf{S}[2,4]]$, which becomes enabled once both processes 2 and 4 are in the process state done. Firing $\mathsf{HE}[\mathsf{S}[2,4]]$ finishes SBD T, which turns process 11 of the SBD S to the process state done. Since the postcondition of process 11 is trivial, its successive hyper-edge $\mathsf{HE}[\mathsf{S}[11]]$ is directly enabled and thus immediately fired afterwards. This eliminates all tokens in SBD S and, since there is no token left in any SBD, the execution of this SBD project is terminated.

2.2 Translating SBDs into automata

Based on the formal semantics introduced in Section 2.1, the current section proposes the procedure to translate SBDs into automata. Technically, we represent the global behaviour of an SBD project by the synchronisation of multiple automata, each of which is translated from one SBD in the project[6] while an explicit construction of the global behaviour is unnecessary.

[6] Note that we do not explicitly refer to as the standard *synchronous composition* at the current stage. See Step 4 below.

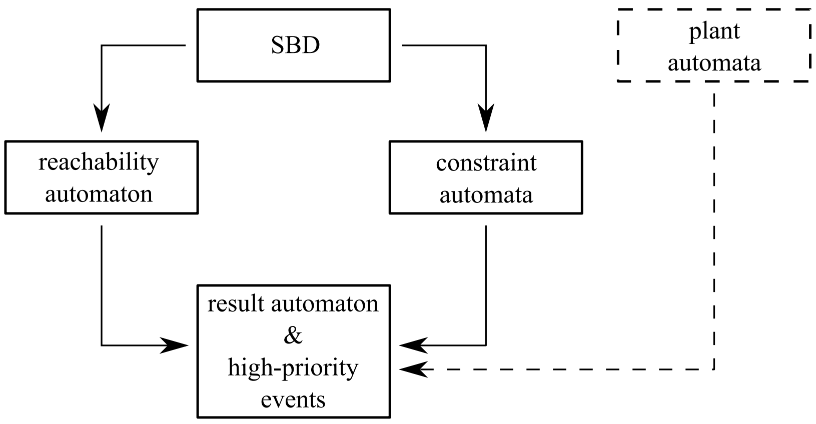

Figure 11: Translation procedure of a single SBD (plant automata are not considered throughout this section)

Moreover, each SBD is translated based on the construction of its corresponding reachability graph as well as various types of constraint automata. By referring to Figure 11, we outline the translation procedure for each SBD as follows:

Step 1 Construct the reachability graph of the SBD and interpret it as an automaton by mapping each reachable configuration of the SBD into a state and mapping hyper-edges to the alphabet of this automaton. In addition, for a non-root SBD in the context of nested SBDs, the reachability graph is extended to describe its cyclic activation and deactivation, since its invokers may be activated multiple times.

Step 2 Construct automata to represent various constraints. In the current section, there are two types of automata involved:

Condition automata Automata of this type handle the guard conditions of hyper-edges. By interpreting value change of variables as events, automata can be constructed where each state represents the evaluation of one or multiple variables. As we only consider conditions formulated based on equality propositions, hyper-edges can be fired only in states where the variable evaluations satisfy the guard condition.

Process state automata Automata of this type organise the process state cycles of processes.

Step 3 All automata constructed in the previous two steps are composed through synchronous composition (Cassandras and Lafortune, 2008). To represent the closed-loop behaviour of the (sub-)system, a pre-constructed plant model is taken into the composition as well. Finally, to represent the high-priority of firing certain hyper-edge over value changes of variables, we collect all events with higher priority as part of the translation result. A typical situation is that for an SBD upon initialisation (i.e. the configuration is $\mathsf{InitialNodes}$), all enabled hyper-edges must be fired *immediately* as soon as the preconditions of all successive processes evaluate true.

From the translation procedure above, translating one SBD results in one single automaton combined with a set of high priority events. The technical details are illustrated in the following subsections. At the end of the current section, we will also clarify how the global behaviour can be represented by the translation results, i.e. how are the automata synchronised considering the high-priority events. Note that for an automaton constructed during the translation of some SBD $S \in \mathsf{SBDP}$, we persist to utilise the subscript $(\cdot)_S$ if multiple SBDs are involved. The superscript $(\cdot)^{\mathsf{GL}}$ is utilised in similar situations where uniform type of elements resulting from each individual SBD translation need to be collected.

2.2.1 Reachability automaton

To represent the dynamics of a Petri-net, its *reachability graph* is commonly utilised which is a directed graph where each vertex denotes one reachable token configuration and each directed edge is associated with one or several Petri-net transitions. Analogously, we represent the individual update of an SBD as defined in (23) and (26) by constructing a reachability graph and interpret it as an automaton. Such an automaton G_{REACH} is a *reachability automaton* with its alphabet Σ_{REACH}. For the drill station example, both reachability automata resulting from SBDs S and T are depicted in Figure 12. To clarify Σ_{REACH}, we utilise the event set Σ_{HEs} in which each event $\sigma_h \in \Sigma_{\mathsf{HEs}}$ is bijectively mapped to a hyper-edge $h \in \mathsf{HEs}$, i.e.

$$\Sigma_{\mathsf{HEs}} = \{\sigma_h \,|\, h \in \mathsf{HEs}\}. \tag{42}$$

Clearly, this event set is identical to the alphabet of the reachability automaton of a root SBD, e.g. for $G_{\mathsf{REACH},S}$ in Figure 12, we have

$$\Sigma_{\mathsf{REACH},S} := \Sigma_{\mathsf{HEs},S}\,. \tag{43}$$

In addition, from the SBD initialisation as defined in (31), the initial state of $G_{\mathsf{REACH},S}$ corresponds to the configuration $\mathsf{InitialNodes}_S$. On the other hand, for any non-root SBD, the alphabet of its reachability automaton shall be

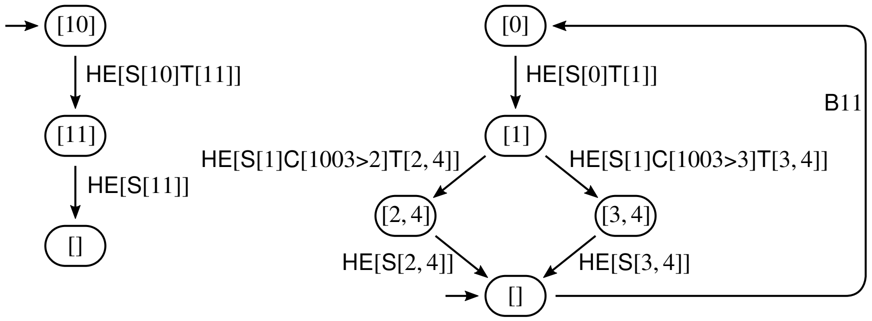

Figure 12: Reachability automata of the drill station example: $G_{\mathsf{REACH},S}$ for S (left) and $G_{\mathsf{REACH},T}$ for T (right)

extended. From (34), an invoker being sent to the process state busy implies that the invoked SBD gets tokens in its initial nodes. As for the non-root SBD T, its corresponding alphabet is given by

$$\Sigma_{\mathsf{REACH},T} := \Sigma_{\mathsf{HEs},T} \dot{\cup} \Sigma_{\mathsf{INV},T} \tag{44}$$

where

$$\Sigma_{\mathsf{INV},T} := \{\mathsf{B}n \mid n \in \mathsf{Processes}^{\mathsf{GL}} \wedge \mathsf{invoke}(n) = T\}. \tag{45}$$

At the current stage, we explain the event set $\Sigma_{\mathsf{INV},T}$ as such that by executing any $\mathsf{B}n \in \Sigma_{\mathsf{INV},T}$, the process state of $n \in \mathsf{Processes}^{\mathsf{GL}}$ (which invokes T) is turned to busy. In fact, events in the form of $\mathsf{B}n$ are referred to as "busy events" which will be discussed in detail in the following subsection. For the current example, we have $\Sigma_{\mathsf{INV},T} = \{\mathsf{B11}\}$. In the subsequent subsections, we shall see that the event B11 will appear in the translation result of S as well. This construction represents the hierarchical structure in a modular fashion, which is similar to the usage of *interface* in (Leduc, 2002a). Furthermore, as (31) suggests, the initial state of $G_{\mathsf{REACH},T}$ corresponds to its empty configuration and for each $\mathsf{B}n \in \Sigma_{\mathsf{INV},T}$, a transition is constructed from the empty configuration to the configuration $\mathsf{IntialNodes}_T$. Finally, from the perspective of automata theory, we point out that state names are only cosmetically illustrated in figures as they do not contribute to the formal language generated by the automaton.

2.2.2 Constraint automata

We now construct automata which guard the transitions in a reachability automaton, namely, condition automata and process state automata.

2.2.2.1 Condition automata

Recall from Definition 2.1.4 that a hyper-edge can be fired only if its corresponding guard condition (38) evaluates true. Since we only consider conditions formulated by equality propositions resulting from variable evaluations, for any condition $c \in \mathsf{Conditions}$, an automaton G_c is constructed based on composing variable automata G_v for each $v \in \mathsf{Variables}$ involved in c. Basically, the state set of G_v is set up as such that each state is bijectively mapped to a value $l \in \mathsf{range}(v)$, from which we define the alphabet of each G_v as

$$\Sigma_v := \{\sigma_{v,l} \,|\, l \in \mathsf{range}(v)\} \tag{46}$$

where each event $\sigma_{v,l} \in \Sigma_v$ is interpreted as "the value of variable v has changed to l". For convenience, we utilise the notation

$$\Sigma_{\mathsf{VAR}} := \cup_{v \in \mathsf{Variables}} \Sigma_v \tag{47}$$

as well to denote all *variable events* of an SBD. Finally, one or several values can optionally be picked from $\mathsf{range}(v)$ to denote possible *initial values* of v in order to restrict the set of initial states. Otherwise, all states of G_v are considered initial.

We show an example G_{light} for a variable light with three possible values $\mathsf{range}(\mathsf{light}) = \{\mathsf{off}, \mathsf{blink}, \mathsf{on}\}$ and initial value off in Figure 13. To match the style of hyper-edge names, we symbolically represent each event in Σ_v for the variable v by

$$\sigma_{v,l} \equiv \mathsf{VE}[v, l] \tag{48}$$

Typically, each value may be changed to any other value freely, which may seem to be overly permissive in some contexts. Consider briefly another situation where a variable depth has three values $\mathsf{range}(\mathsf{depth}) = \{\mathsf{0cm}, \mathsf{1cm}, \mathsf{2cm}\}$. This can e.g. be utilised to denote the drilling depth for the drill station example. Clearly, from $\mathsf{0cm}$, the state $\mathsf{2cm}$ shall not be reachable without first reaching $\mathsf{1cm}$. Nevertheless, such kind of restrictions can always be described in properly constructed plant models.

Let $h \in \mathsf{HEs}$ be some hyper-edge. In order to construct G_{c_h} for the guard condition $c_h \in \mathsf{Conditions}$ of h, the synchronous composition of all G_v where $v \in \mathsf{Variables}$ is involved in c is first to take. Each state in the resulting automaton indicates a possible evaluation of all involved variables. On this basis, self-loops labelled by $\sigma_h \in \Sigma_{\mathsf{REACH}}$ can be appended in states of this automaton where the guard condition c_h of $h \in \mathsf{HEs}$ evaluates true. Finally, for the considered SBD, an automaton G_{COND} which guards all hyper-edges of the

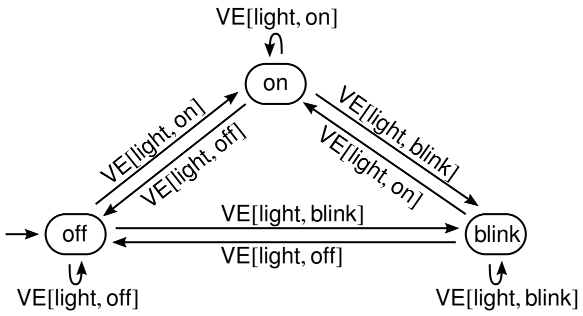

Figure 13: Automaton tracking a variable light with range {off, blink, on}

current SBD can be constructed by computing the synchronous composition of all G_{c_h} for each $h \in \mathsf{HEs}$.

Remark 2.2.1. *Since the guard condition defined in* (38) *is in conjunctive form, we could normally separately construct automata for precondition, postcondition, etc., which are then composed via synchronous composition. Moreover, if* Conditions *only recognises conjunctions of equality propositions, condition automata can be constructed straightforward on a per-variable basis without needing an explicitly construction of G_c. This is achieved by directly constructing G_v for each $v \in \mathsf{Variables}$ and append a self-loop labelled by $\sigma_h \in \Sigma_{\mathsf{REACH}}$ in each specific state of G_v whenever*

- *v is involved in the guard condition of h and*
- *the value corresponding to this state matches the equality proposition utilised in h.*

On this basis, G_{COND} can be computed by the synchronous composition of all such "self-loop augmented" G_v, since G_{COND} overall describes the conjunction of a collection of equality propositions. Note that special care should be taken for branch conditions if this pure conjunctive form of conditions is adopted, especially when utilising the keyword else *for deterministic successor choice. If one of the branch condition is a conjunction of at least two equality propositions, its complement is generally a disjunction, which cannot be implicitly described by synchronous composition. In such cases, we shall separately construct a condition automaton G_c for the branch choice with* else *condition, which is again composed into G_{COND}.*

As for the drill station example, there are globally three involved variables for various conditions, i.e. $\mathsf{Variables}_S = \{\mathsf{button}\}$ and $\mathsf{Variables}_T = \{\mathsf{position}, \mathsf{wptype}\}$. Their respective value ranges and initial values are listed in Table

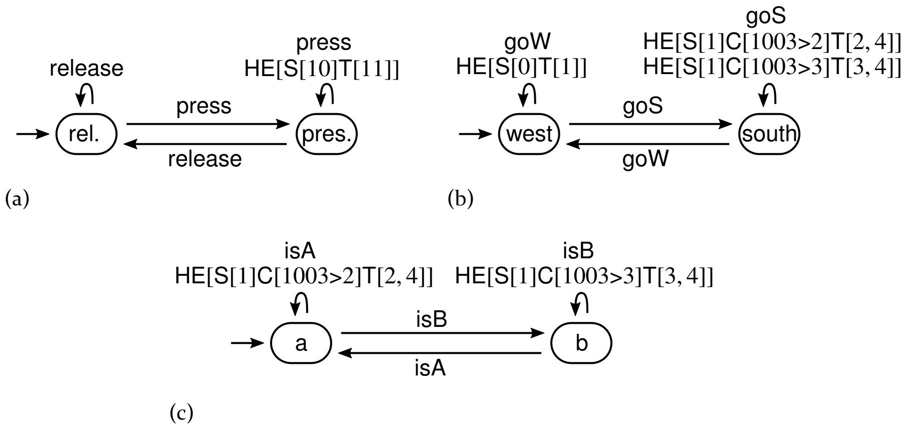

Figure 14: Variable automata for variables button (a), position (b) and wptype (c)

1. With Remark 2.2.1, three variable automata G_{button}, G_{position} and G_{wptype} are constructed with correspondingly augmented self-loops of hyper-edge events. On this basis, $G_{\text{COND},S}$ is identical to G_{button}, while $G_{\text{COND},T}$ can be constructed by computing the synchronous composition of G_{position} and G_{wptype}.

2.2.2.2 Process state automata

In this paragraph, process state automata are constructed in order to represent the process state cycles of processes as defined in (24) and (25). Basically, we utilise Σ_{PROC} to denote the set of events which change the process state of a process, i.e.

$$\Sigma_{\text{B}} := \{\text{B}n \mid n \in \text{Processes}\}; \tag{49}$$

Table 1: Variables involved in the drill station example with their corresponding values (initial values are underlined); push ≡ VE[button, pushed], release ≡ VE[button, released], goW ≡ VE[position, west], goS ≡ VE[position, south], isA ≡ VE[wptype, a] and isB ≡ VE[wptype, b].

variable	values	events
button	{pushed, <u>released</u>}	{push, release}
position	{south, <u>west</u>}	{goW, goS}
wptype	{<u>a</u>, b}	{isA, isB}

$$\Sigma_{\mathsf{D}} := \{\mathsf{D}n \,|\, n \in \mathsf{Processes}\}; \tag{50}$$
$$\Sigma_{\mathsf{I}} := \{\mathsf{I}n \,|\, n \in \mathsf{Places}\}; \tag{51}$$
$$\Sigma_{\mathsf{PROC}} := \Sigma_{\mathsf{B}} \cup \Sigma_{\mathsf{D}} \cup \Sigma_{\mathsf{I}} \tag{52}$$

where each $\mathsf{B}n \in \Sigma_{\mathsf{B}}$, $\mathsf{D}n \in \Sigma_{\mathsf{D}}$ or $\mathsf{I}n \in \Sigma_{\mathsf{I}}$ changes the process state of process $n \in \mathsf{Processes}$ to busy, done or idle, respectively. For convenience, we associate each initial node with an idle event to indicate that the token has left the node. Besides, we recall that firing an enabled hyper-edge causes a series of process state changes of its source and target places. To acknowledge that all process state changes caused by firing some hyper-edge are completed, an additional event ack_S is introduced for each SBD $S \in \mathsf{SBDP}$ which is often paired with $\sigma_h \in \Sigma_{\mathsf{HEs}}$ in process state automata.

For each individual SBD $S \in \mathsf{SBDP}$, we construct one process automaton which is the synchronous composition of five types of automata. For convenience, we utilise regular expressions[7] to represent each automaton to be constructed due to their overall cyclic structure. We shall point out that, to save computational effort, we do not explicitly handle done event $\mathsf{D}n \in \Sigma_{\mathsf{D}}$ if n is not a invoker, since from (25), such $\mathsf{D}n$ may spontaneously happen between an $\mathsf{B}n \in \Sigma_{\mathsf{B}}$ and $\mathsf{I}n \in \Sigma_{\mathsf{I}}$ without any constraints. Nevertheless, we point out in advance that in Section 2.3.1, we may flexibly assign the done event to any process depending on modelling requirements.

(PA1) For each process $n \in \mathsf{Processes}$, we represent the process state cycle by generating
$$(\,\mathsf{B}n \cdot \mathsf{D}n \cdot \mathsf{I}n\,)^* \tag{53}$$
if n is an invoker or
$$(\,\mathsf{B}n \cdot \mathsf{I}n\,)^* \tag{54}$$
if n is not a invoker, respectively. Note that the generated sequence always begins with a busy event $\mathsf{B}n \in \Sigma_{\mathsf{B}}$ since all processes are initially in the process state idle, as defined in (32). Besides, by considering each initial node as a special type of empty process without predecessors, we generate
$$\mathsf{I}n \tag{55}$$

[7] Automata in the current paragraph are represented by regular expressions where sums $+$ and products $\cdot$ stand for expression union and concatenation, respectively. A superscript asterisk $(\cdot)^*$ denotes the Kleene-closure of a language. The terminology of "by generating some regular expression" is interpreted as such that the generated language of the automaton to construct matches the prefix closure of the given regular expression (Cassandras and Lafortune, 2008).

for each $n \in \mathsf{InitialNodes}$ if S is a root SBD, since once a token has left n, n can never hold a token again; in contrary, if S is not a root SBD,

$$(\,\Sigma_{\mathsf{INV},S}\,\cdot\,\mathsf{I}n\,)^* \tag{56}$$

is generated for n, since S can repeatedly be invoked. By referring to the definition of synchronous composition, we observe that for an invoker process n, an automaton generating (53) and $G_{\mathsf{REACH},T}$ for any $\mathsf{invoke}(n) = T$ are synchronised over the event $\mathsf{B}n$, which indeed represents (34).

(PA2) For each hyper-edge $h \in \mathsf{HEs}$, we represent the deactivation of source places as well as the activation of target places by generating

$$\begin{aligned}((\underbrace{\mathsf{I}n_1 + \cdots + \mathsf{I}n_k + \mathsf{B}m_1 + \cdots + \mathsf{B}m_{k'} + \mathsf{ack}_S}_{(\$)})^* \cdot \sigma_h \\ \cdot \underbrace{\mathsf{I}n_1 \cdots \mathsf{I}n_k \cdot \mathsf{B}m_1 \cdots \mathsf{B}m_{k'}}_{(\$\$)} \cdot \mathsf{ack}_S)^*\end{aligned} \tag{57}$$

to denote that firing h sends places $\{n_1, \ldots, n_k\} = \mathsf{Sources}(h)$ to the process state idle and sends places $\{m_1, \ldots, m_{k'}\} = \mathsf{Targets}(h)$ to the process state busy, respectively. Note that enabling the (\$) part is necessary since all places $n_1, \ldots, n_k, m_1, \ldots, m_{k'}$ in (\$) can also be sources or targets of hyper-edges other than h. Besides, ack_S can also be utilised to acknowledge hyper-edges other than h in HEs_S. Moreover, the order of events in (\$\$) of the above expression is in general inessential.

(PA3) For each process $n \in \mathsf{Processes}$, we restrict the execution of $\mathsf{B}n$ so that it can only occur between a hyper-edge, of which n is a target, and the subsequent ack_S. Thus, we utilise

$$\Sigma_n^{\mathsf{TARGET}} := \{\sigma_h \in \Sigma_{\mathsf{HEs}} \,|\, n \in \mathsf{Targets}(h)\} \tag{58}$$

to denote the set of hyper-edge events which place a token on n and generate

$$(\,\mathsf{ack}_S^* \,\cdot\, \Sigma_n^{\mathsf{TARGET}} \,\cdot\, \mathsf{B}n^* \,\cdot\, \mathsf{ack}_S\,)^* \tag{59}$$

for each $n \in \mathsf{Processes}$. Recall that initial nodes are not associated with busy events. Similarly, we generate

$$(\,\mathsf{ack}_S^* \,\cdot\, \Sigma_n^{\mathsf{SOURCE}} \,\cdot\, \mathsf{I}n^* \,\cdot\, \mathsf{ack}_S\,)^* \tag{60}$$

for each $n \in \mathsf{Places}$ where

$$\Sigma_n^{\mathsf{SOURCE}} := \{\sigma_h \in \Sigma_{\mathsf{HEs}} \,|\, n \in \mathsf{Sources}(h)\} \tag{61}$$

denotes the set of hyper-edge events which take a token from n.

(PA4) For each *invoker* process $n \in \mathsf{Processes}$, we generate

$$(\,\mathsf{D}n \;\cdot\; \Sigma_n^{\mathsf{SOURCE}} \;\cdot\; \mathsf{I}n\,)^* \,. \tag{62}$$

to represent that firing a hyper-edge is possible only if all its source processes are in the process state done.

(PA5) Since firing hyper-edges is instantaneous, it is clear that variable events shall not occur between a hyper-edge event and a subsequent ack_S. Thus, we generate

$$(\,(\Sigma_{\mathsf{VAR}})^* \;\cdot\; \Sigma_{\mathsf{HEs}} \;\cdot\; \mathsf{ack}_S\,)^* \,. \tag{63}$$

for the current SBD S.

Note that the above construction only handles a single SBD $S \in \mathsf{SBDP}$, which indicates that (P5) only addresses local variable events. For an SBD project with multiple SBDs, an overall version of (P5) shall be generated as

$$(\,(\Sigma_{\mathsf{VAR}}^{\mathsf{GL}})^* \;\cdot\; \Sigma_{\mathsf{HEs}}^{\mathsf{GL}} \;\cdot\; \Sigma_{\mathsf{ack}}^{\mathsf{GL}}\,)^* \,. \tag{64}$$

where $\Sigma_{\mathsf{ack}}^{\mathsf{GL}} := \cup_{S \in \mathsf{SBDP}} \{\mathsf{ack}_S\}$ is the union of all hyper-edge acknowledgements.

At this stage, we recall from (35) where we required that the done state transition of a invoker and finishing the invoked SBD are synchronised. To this end, we first take the assumption that

$$\forall S \in \mathsf{SBDP}.\ \mathsf{invoked}(S) \neq \emptyset \;\; \Rightarrow |\mathsf{Terminals}_S| = 1 \,, \tag{65}$$

i.e. any non-root SBD has exactly one terminal node. In this situation, any non-root SBD T is finished if any event in

$$\Sigma_{\mathsf{FIN},T} := \{\sigma_h \in \Sigma_{\mathsf{HEs},T} \,|\, \mathsf{Targets}(h) = \emptyset\} \tag{66}$$

is executed. By recalling (35), it is the event set $\Sigma_{\mathsf{FIN},T}$ that sends the invokers of T to the process state done. Hence, we conveniently define a mapping fin as

$$\mathsf{fin}(\mathsf{D}n) = \begin{cases} \{\mathsf{D}n\} & \text{if } \mathsf{invoke}(n) = \emptyset\,; \\ \Sigma_{\mathsf{FIN},\mathsf{invoke}(n)} & \text{if } \mathsf{invoke}(n) \neq \emptyset \end{cases} \tag{67}$$

for all $n \in \mathsf{Processes}^{\mathsf{GL}}$ and replace $\mathsf{D}n$ in (53) and (62) with $\mathsf{fin}(\mathsf{D}n)$. As for the drill station example, the only explicit done event is D11 (since only the process with ID = 11 is an invoker), which should be replaced by $\mathsf{fin}(\mathsf{D}11) = \{\,\mathsf{HE}[\mathsf{S}[2,4]], \mathsf{HE}[\mathsf{S}[3,4]]\,\}$. At this stage, it is worth mentioning that one consequence of the replacement through fin is that done events will totally disappear in the translation result. Nevertheless, we shall see below in Section 2.3.1 that we could optionally append done events to atomic processes as well.

Remark 2.2.2. *Depending on the verification purpose, it is often desired to only preserve the (replaced) done events* $\mathsf{fin}(\Sigma_{\mathsf{D}})$ *(that is the union of* $\mathsf{fin}(\mathsf{D}n)$ *of all processes) in the model while neglecting* Σ_{I}, Σ_{B} *as well as* ack_S. *In such cases, the tedious construction of process state automata as proposed in (PA1)–(PA5) as well as* (64) *can be circumvented, since the reachability automaton* G_{REACH} *already implicitly encodes whether a process is currently in the process state* idle *or not. In this regard, there are two questions to answer:*

How to synchronise invocation *We recall that a non-root SBD is activated through the synchronisation via busy events of its invokers. For each non-root SBD T, if we remove all busy events in the translation result, we shall simply replace* $\Sigma_{\mathsf{INV},T}$ *as defined in* (45) *by*

$$\Sigma^{\mathsf{EXT}}_{\mathsf{INV},T} := \{\sigma_h \in \Sigma^{\mathsf{GL}}_{\mathsf{HEs}} \mid \exists n \in \mathsf{Targets}(h).\ \mathsf{invoke}(h) = T\}, \tag{68}$$

i.e. the set of hyper-edge events, by executing which an invoker of T receives a token.

How to distinguish busy and done states *To answer this question, we only need to introduce a binary flag for each invoker process, which evaluates* true *if and only if this process in currently in the process state* done. *Upon receiving a token, the flag is initially* false, *i.e. the process is in the process state* busy, *which then becomes* true *by executing any event in* $\mathsf{fin}(\mathsf{D}n)$. *On this basis, instead of composing the process state automaton with the reachability automaton later on, we can more efficiently extend the reachability automaton by enabling* $\mathsf{fin}(\mathsf{D}n)$ *for each process* n *which is currently in the process state* busy, *i.e. those in current configuration but with the additional flag evaluated* false. *Correspondingly, executing any event in* $\mathsf{fin}(\mathsf{D}n)$ *changes the flag to* true. *This kind of state space reduction is also utilised in other semantics formalisation scenarios; see e.g. (Daw and Cleaveland, 2015a) where the author computed the so-called "macro steps" of ADs by abstracting the detailed token propagation steps.*

In summary, if the construction of process state automata is undesired, minor modifications in the reachability automaton are required. The resulting extended reachability automata $G^{\mathsf{EXT}}_{\mathsf{REACH},S}$ *and* $G^{\mathsf{EXT}}_{\mathsf{REACH},T}$ *of SBDs* S *and* T *for the drill station example are illustrated in Figure 15.*

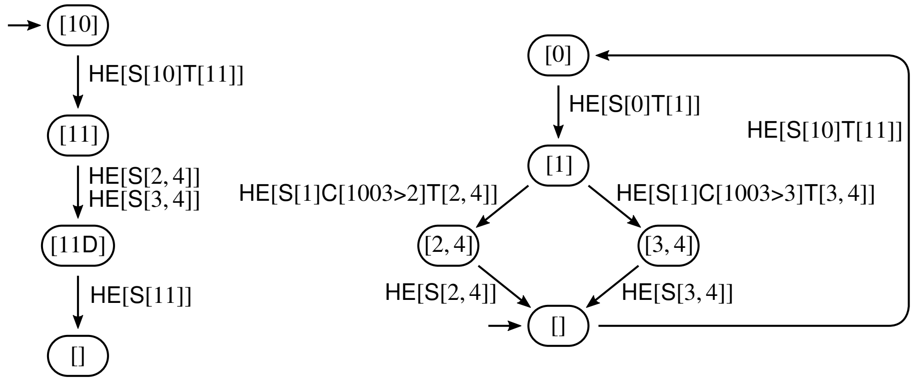

Figure 15: Extended reachability automata for SBDs S and T in the drill station example

2.2.3 Result automaton and high-priority events

All automata constructed in the previous two steps (as well as a plant modelled by automata) are composed through synchronous composition to form the translation result G_{SBD}. At this stage, we again consider Definition 2.1.5. As soon as a hyper-edge becomes enabled (and if deterministic branch choices are guaranteed), it must be fired instantaneously. Combining the explanation of variable value changes in the time model as introduced in Section 2.1.3 and Figure 10, we conclude that at some state in G_{SBD}, if a hyper-edge $h \in \mathsf{HEs}$ is enabled, its corresponding event $\sigma_h \in \Sigma_{\mathsf{HEs}}$ shall take priority over all variable events Σ_{VAR}. However, concluding that a hyper-edge is enabled requires that all its source processes are in the process state done, which currently can only be implied if the considered process is an invoker, i.e. its completion is implied by the completion of the SBD it invokes. Note that although process completion implies that its postcondition evaluates true, the converse generally does not hold. Thus, we collect all hyper-edge events $\sigma_h \in \Sigma_{\mathsf{HEs}}$ so that

$$\forall n \in \mathsf{Sources}(h).\ n \in \mathsf{InitialNodes}\ \vee\ (n \in \mathsf{Processes} \wedge \mathsf{invoke}(n) \neq \emptyset) \quad (69)$$

holds. Such events are related to hyper-edges whose enabledness can be concluded on a per-state basis and thus are with higher priority. As for the drill station example, we shall collect $\{$ HE[S[10]T[11]], HE[S[11]] $\}$ from SBD S and $\{$ HE[S[0]T[1]] $\}$ from SBD T as high-priority events.

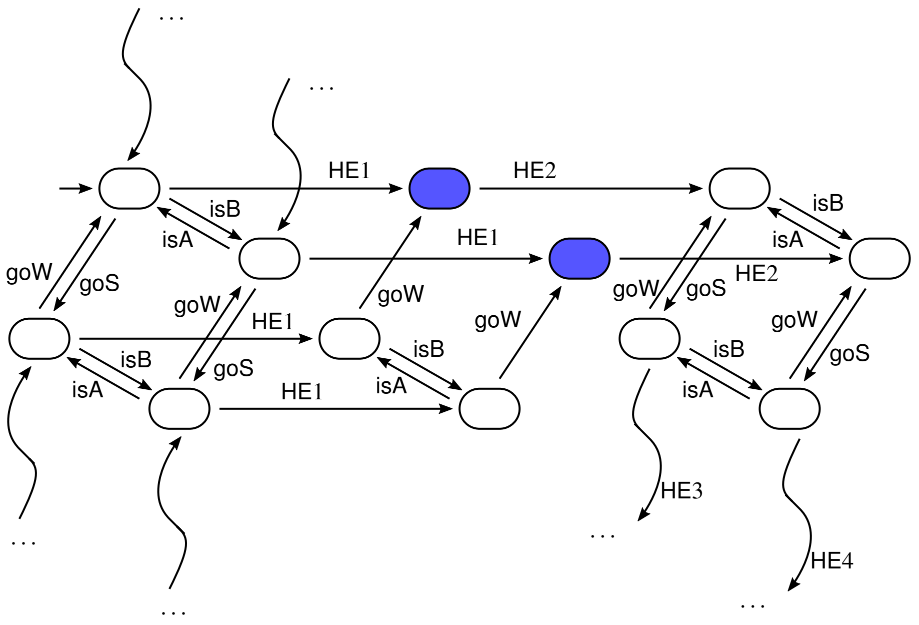

Figure 16: Translation result after local shaping (self-loops of Σ_{VAR} are omitted); HE1 = HE[S[10]T[11]], HE2 = HE[S[0]T[1]], HE3 = HE[S[1]C[1003>2]T[2, 4]], HE4 = HE[S[1]C[1003>3]T[3, 4]]; at both states where HE2 is active (blue), all active variable events are disabled

2.2.4 Representing the global behaviour

The translation procedure introduced hitherto represents each single SBD as one automaton with a set of high priority events, as depicted in Figure 11. For the entire SBD project, the monolithic global behaviour complies with a single automaton which can be constructed by

- translating each SBD following the procedure hitherto and constructing their synchronous composition;
- shaping spurious transitions according to the high priority events, i.e. if any high-priority event is executable in some state, all outgoing transitions labelled by a non-high-priority event must be removed.

The purpose of (ii) is to remove all transitions labelled by variable events whenever a high priority event collected in Section 2.2.3 is active. This is also referred to as *preemption* which represents the fact that if a hyper-edge is enabled, it must be fired immediately before any variable changes its value, i.e. enabled hyper-edges *preempt* variable events.

Remark 2.2.3. *Without influencing the global behaviour, shaping spurious transitions can in fact already partially be applied in the local construction phase. This simplifies each module by reducing its transition count, which possibly reduces its state space since some states may become unreachable. Technically, if a high-priority event is private, i.e. does not appear in other synchronised automata, we can shape the local behaviour directly since the high-priority event will never be disabled by other modules; see Lemma 3.2.2 in Chapter 3 for a more detailed explanation. We show a fragment of the shaped translation result of SBD T in the drill station example in Figure 16, where the only high priority event* HE[S[0]T[1]] *is clearly private. Note that by following Remark 2.2.2, we omit representing* Σ_B *and* Σ_I *explicitly. Similarly, both high priority events* $\{$ HE[S[10]T[11]], HE[S[11]] $\}$ *from SBD S are private as well, thus can locally preempt other events.*

2.3 Extended semantics

As shown in the previous two sections, SBD semantics is generally represented by firing enabled hyper-edges in an extended Petri-net with guard conditions and process state cycles. Technically, the process state cycle is a means of abstraction of process operations. However, this is in some cases a too weak model for verification. We consider the following practical scenario as shown in Figure 17: suppose the temperature of the liquid in a container is to be controlled. At some stage direct before process Heat is activated, the container is cooled down naturally. When hitting the critical temperature 50°C (as stated in the precondition in Figure 17), the process Heat is activated which heats the container with heating wires. When reaching temperature 100°C, process Heat should be left so that the correctly heated liquid can be processed further.

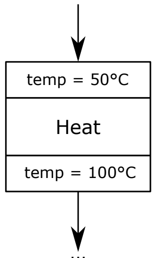

Figure 17: A temperature controlling process

We consider the following three questions which cannot be answered within the current SBD semantics:

Question 1: When does the process terminate? If only the information from postcondition is available, the process Heat is allowed to heat the liquid to any extremely high temperature (which shall not be allowed) and eventually cool it down to around 100°C. This ill-formed sequence can even be repeated multiple times, as we only need to guarantee that the temperature is 100°C when the process is terminated.

Question 2: Which variable(s) can(not) be manipulated? Naturally, a faithfully modelled plant shall represent the logic that the temperature of the liquid can increase only if it is heated. Thus, heating itself may be related to e.g. a boolean variable, by setting which to 1 the container is heated. If no information about which process can manipulate which variable (i.e. write values to a variable) is available, any process may freely heat the liquid.

Question 3: When should the control come into effect? Although the process Heat is intended to heat the liquid at 50°C, there is no information describing when the container will actually be heated. Thus, the temperature of the liquid can still decrease when the process Heat is in the process state busy, since starting the process Heat does not necessarily imply that the liquid is immediately heated.

Without being able to answer the above three questions, the translation result may conceivably allow spurious closed-loop behaviour and produce overly pessimistic verification result. To answer the questions, we propose several optional annotations for each process in the following respective subsections. Note that the extensions do not change the overall translation procedure as depicted in Figure 11. Instead, only the individual steps are modified.

2.3.1 Termination condition

Recall from (53) that a process $n \in \mathsf{Processes}$ (with an explicit done event $\mathsf{D}n$) cycles its process states by repeatedly executing $\mathsf{B}n \cdot \mathsf{D}n \cdot \mathsf{I}n$. As stated in (57), the busy event $\mathsf{B}n$ and the idle event $\mathsf{I}n$ are triggered by firing hyper-edges and thus are guarded by guard conditions. This inspires us to answer Question 1 by optionally assigning a *termination condition* to a *non-invoker process*. Similar to invoker processes, each non-invoker process n with specified termination condition is equipped with an explicit done event $\mathsf{D}n$, whose execution is guarded by the termination condition. The assignment of termination conditions is represented by the map

$$\mathsf{termcond} : \ \mathsf{Processes} \to \mathsf{Conditions} \,\dot{\cup}\, \{\emptyset\}. \tag{70}$$

where $\emptyset \notin \mathsf{Conditions}$ is dedicated to representing *unspecified* condition. Thus, we clearly have

$$\forall n \in \mathsf{Processes}.\ \mathsf{invoke}(n) \in \mathsf{SBDP} \ \ \Rightarrow \mathsf{termcond}(n) = \emptyset\,, \tag{71}$$

since the termination of a invoker process is implied by finishing the SBD it invokes. For any busy process $n \in \mathsf{Processes}$ with $\mathsf{termcond}(n) \neq \emptyset$ (i.e. the termination condition of n is *specified*), its process state must immediately be switched to done whenever $\mathsf{termcond}(n)$ evaluates true. This implies a refinement of the individual update of an SBD as stated in Definition 2.1.5, i.e. we shall append

$$\mathsf{ProcessState}_n(\iota + 1) = \mathsf{done} \tag{72}$$

for each $n \in \mathsf{Processes}$ with $\mathsf{termcond}(n) \in \mathsf{Conditions}$ if

(i) $\mathsf{ProcessState}_n(\iota) = \mathsf{busy}$;

(ii) $\mathsf{termcond}(n)$ evaluates true at time ι.

Remark 2.3.1. *Readers shall not confuse* unspecified *conditions with* trivial *conditions. A trivial condition is a condition that evaluates* true *at any time. However, as introduced in Definition 2.1.10, unspecified condition cannot be assigned to preconditions, postconditions, branch conditions and initial conditions.*

Recall from Section 2.1.4 that the termination of a process implies its postcondition. Thus, the termination condition of a process, if specified, must imply its postcondition. In this context, if a hyper-edge has only one source process, no branches involved and preconditions of all its target places evaluate true, the termination condition of the source process triggers the hyper-edge immediately, i.e. the tokens are instantaneously propagated before the value of any variable changes. Contrarily, if such a hyper-edge is delayed due to e.g. invalid precondition of some target processes, the process may remain in the process state done for a positive duration of physical time. In this time period, the postcondition may be invalidated due to e.g. the operation of other running processes. This circumstance will be addressed in detail in the following Section 2.3.2 where we discuss the write access to variables.

By reviewing the translation procedure introduced in Section 2.2, we implement the termination condition by modifying and extending the steps in Sections 2.2.2 and 2.2.3:

Section 2.2.2 For each process $n \in \mathsf{Processes}$ with $\mathsf{termcond}(n) \in \mathsf{Conditions} - \{\mathsf{true}\}$ (i.e. n has specified non-trivial termination condition), an explicit done event $\mathsf{D}n$ is equipped to n which can be seen as a generalised hyper-edge event. In this context, $\mathsf{D}n$ is enabled only if $\mathsf{termcond}(n)$ evaluates true. Recall that the condition automaton G_{COND} was originally composed by a set of automata associated with guard conditions of hyper-edges. This set is thus augmented by automata associated with termination conditions of all involved processes. Thus, a brief adaption is required for (AP1) in Section 2.2.2 as well so that for each process $n \in \mathsf{Processes}$ with $\mathsf{termcond}(n) \in \mathsf{Conditions} - \{\mathsf{true}\}$, we generate $(\mathsf{B}n \cdot \mathsf{D}n \cdot \mathsf{I}n)^*$ as well.

Section 2.2.3 Due to the introduction of termination condition, more hyper-edges can be determined as enabled from a per-state basis. Technically, all hyper-edge events $\sigma_h \in \Sigma_{\mathsf{HEs}}$ where

$$\forall n \in \mathsf{Sources}(h).\ n \in \mathsf{InitialNodes}\ \lor\ (n \in \mathsf{Processes} \land \mathsf{invoke}(n) \neq \emptyset) \\ \lor\ \mathsf{termcond}(n) \in \mathsf{Conditions} \qquad (73)$$

are considered as high priority events, i.e. we extend (69) by additionally allowing source places to have specified termination conditions. In addition, for each process $n \in \mathsf{Processes}$ so that $\mathsf{termcond}(n) \in \mathsf{Conditions} - \{\mathsf{true}\}$, its corresponding done event $\mathsf{D}n \in \Sigma_{\mathsf{D}}$ is with high priority as well.

Since more processes are equipped with explicit done events, it is worth mentioning that the construction of extended reachability automata as suggested in Remark 2.2.2 is influenced as well. As for the drill station example, we assume that processes with ID = 2 and 4 are specified with non-trivial termination conditions. The resulting extended reachability graph of the SBD T is depicted in Figure 18.

2.3.2 Writable and controlled variables

As pointed out by the example in Figure 17, some variables of an SBD may be associated with e.g. actuator manipulation. Note that generally, plant models do not restrict the write access to such variables since the plant model shall allow any kind of control instructions from the controller. Thus, to answer

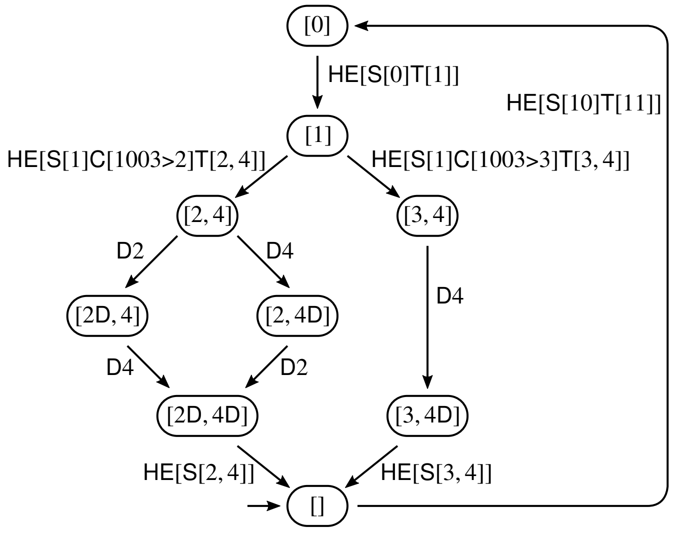

Figure 18: Extended reachability automaton for SBD T with specified termination conditions

Question 2, the write access to a set of *writable* variables needs to be restricted on a per-process and per-SBD basis.

Technically, *writable variables* are referred to as variables that can actively be manipulated by processes and define globally for an SBD project two disjoint sets

$$\mathsf{Variables}^{\mathsf{GL}} = \mathsf{WVariables}^{\mathsf{GL}} \mathbin{\dot{\cup}} \mathsf{UVariables}^{\mathsf{GL}} \tag{74}$$

where WVariables stands for *writable* variables and UVariables stands for *unwritable* variables, respectively. Typically, an unwritable variable describes the status of some sensor or some external control agent. On the other hand, a writable variable can conveniently be utilised to denote actuator status, internal operations or sometimes the consequence of some complicated control sequences.

For an SBD $S \in \mathsf{SBDP}$, owning some writable variable $v \in \mathsf{Variables}_{\mathsf{W},S}$ does not imply that there exists some process $n \in \mathsf{Processes}_S$ having write access to v, since v may be manipulated only in some other SBD $T \neq S$ while S only passively reads v to e.g. form guard conditions. This inspires us to define for each $S \in \mathsf{SBDs}$ and $n \in \mathsf{Processes}_S$ the set of *controlled variables*

$$\mathsf{CVariables}_S(n) \subseteq \{(v,l) \mid v \in \mathsf{WVariables}_S,\ l \in \mathsf{range}(v)\}, \tag{75}$$

where each $(v, l) \in \mathsf{CVariables}_S$ indicates that process n has the access to set the value of v to l. In addition, we write

$$\mathsf{CVariables}_S := \cup_{n \in \mathsf{Processes}_S} \mathsf{CVariables}_S(n) \tag{76}$$

to denote the set of all controlled variables of the SBD S. It is worth mentioning that $\mathsf{CVariables}_S(n) = \emptyset$ implies that the process n does not have write access to any writable variables. Moreover, for nested SBDs, we require that the controlled variables of a invoker process is identical to that of the SBD it invokes, i.e.

$$\forall S, T \in \mathsf{SBDP}, n \in \mathsf{Processes}_S. \, \mathsf{invoke}(n) = T \\ \rightarrow \mathsf{CVariables}_S(n) = \mathsf{CVariables}_T. \tag{77}$$

We are now in the position to represent controlled variables in automata. Naively, we could interpret each controlled variable (v, l) bijectively as one variable event $\sigma_{v,l} \in \Sigma_{\mathsf{VAR}}$. However, apart from to which value a variable is set, it is also important to determine which process has this write access when two parallel processes share some controlled variables. The reason for this assertion is that shared events are synchronously executed in synchronous composition. This indicates that for two parallel (non-invoker) processes $n, n' \in \mathsf{Processes}^{\mathsf{GL}}$ sharing the write access to some controlled variable (v, l), the write access of n to (v, l) may be disallowed by n' (e.g. since n' is currently not active). Thus, we shall, instead of (v, l), map (v, l, n) into an event $\sigma_{(v,l,n)}$ where $n \in \mathsf{Processes}$ and $(v, l) \in \mathsf{CVariables}(n)$. This motivates us to modify the variable event set as $\Sigma_{\mathsf{VAR}} := \cup_{v \in \mathsf{Variables}} \Sigma_v$, i.e. the alphabet Σ_v of an variable automaton G_v of an variable $v \in \mathsf{Variables}$ depends on its writability:

$$\Sigma_v := \begin{cases} \{\sigma_{v,l} \mid l \in \mathsf{range}(v)\} & \text{if } v \in \mathsf{UVariables}; \\ \{\sigma_{v,l,n} \mid l \in \mathsf{range}(v), \mathsf{invoke}(n) = \emptyset, \\ \qquad (v, l) \in \mathsf{CVariables}(n)\} & \text{if } v \in \mathsf{WVariables}. \end{cases} \tag{78}$$

Note that for an invoker process n', its write access to some controlled variable (v, l) is inherited from the SBD it invokes. Hence, no variable event $\sigma_{v,l,n'}$ should be introduced in this situation. Consider the variable automaton in Figure 13 again. Suppose the write access to $(\mathsf{light}, \mathsf{on})$ is shared by two non-invoker processes 1 and 2, while the write accesses to $(\mathsf{light}, \mathsf{off})$

and $(\mathsf{light}, \mathsf{blink})$ are exclusively owned by process 1. The corresponding variable automaton G_{light} should be modified as depicted in Figure 19, where we symbolically name the events with

$$\sigma_{v,l,n} \equiv \mathsf{VE}[v, l, n]. \tag{79}$$

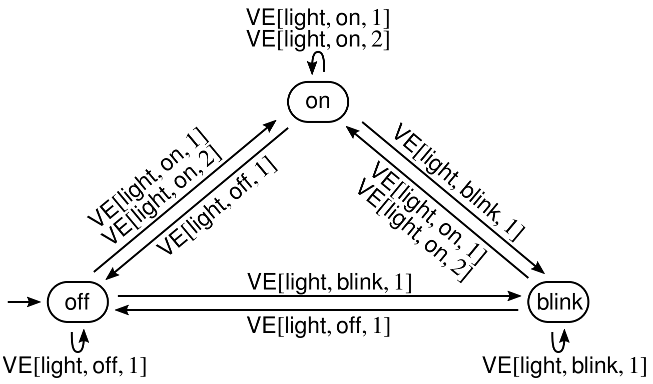

Figure 19: Variable automaton extended for shared write access

We now embed writable and controlled variables into our translation procedure. Technically, additional self-loops are introduced on the reachability automaton to activate/deactivate corresponding events. Note that variable manipulation shall be considered as a part of the process operation. Thus, the write access to controlled variables of a process should only be allowed if it is in the process state busy. This restriction should be specifically handled if a process has an explicit done state, i.e. the process is a invoker or is specified with a non-trivial termination condition. To this end, it is convenient to utilise the extended reachability automaton as suggested in Remark 2.2.2. In each state of the extended reachability automaton of $S \in \mathsf{SBDP}$, we determine on a per-state basis the set of processes $\mathsf{P} \subseteq \mathsf{Processes}_S$ that are currently busy (or possibly busy if it is a non-invoker process without specified termination condition). Afterwards, for each $n' \in \mathsf{P}$, we activate self-loops of events $\sigma_{v,l,n} \in \Sigma_{\mathsf{VAR}}$ where $n \in \mathsf{Processes}^{\mathsf{GL}}$ with $\mathsf{invoke}(n) = \emptyset$ if $(v, l) \in \mathsf{CVariables}(n')$ and either of the two following conditions hold:

(WV1) $n = n'$, or

(WV2) $n \neq n'$ and there exists some invocation sequence $S_0 S_1 \cdots S_k$ so that $n \in \mathsf{Processes}_{S_k}$, $n' \in \mathsf{Processes}_{S_0}$ and $\mathsf{invoke}(n') = S_1$.

In addition,

(WV3) self-loops of $\sigma_{v,l,n}$ should be applied to the empty configuration state in the extended reachability automaton of a *non-root* SBD $S \in \mathsf{SBDP}$.

Note that (WV3) is dedicated for such cases where there exists some process $n'' \in \mathsf{Processes}^{\mathsf{GL}} - \mathsf{invokedBy}(S)$ which accesses v when S is not activated.

Remark 2.3.2. *To satisfy* (77)*, it is possible that for some SBD* $S \in \mathsf{SBDP}$*, there exists* $(v,l) \in \mathsf{CVariables}_S$ *so that* v *does not contribute to any conditions guarding the behaviour of S. The corresponding variable event(s)* $\sigma_{v,l,n}$ *(where* $(v,l) \in \mathsf{CVariables}(n)$*) of such a controlled variable will then only appear as self-loops in the translation result, since they will not be considered when constructing condition automata.*

Remark 2.3.3. *For a non-root SBD* $T \in \mathsf{SBDP}$*, if* $\mathsf{CV} \subseteq \mathsf{CVariables}_T$ *is a set of controlled variables that can be accessed* only when T is active *(i.e. the corresponding variable events can be executed if and only if some processes in T are active), we clearly do not need to handle self-loops w.r.t. (WV2) and (WV3) in the extended reachability automaton of T.*

We again consider the drill station example. Globally, we envisage that the process GetObject performs some complicated control sequences, which eventually rotate the robot arm to the south position. Thus, we globally let $\mathsf{WVariables}^{\mathsf{GL}} = \{\mathsf{position}\}$. Typically, as we may only wish to move the robot arm in one direction, we let

$$\mathsf{CVariables}_T(1) = \{\mathsf{goS}\}; \tag{80}$$

$$\mathsf{CVariables}_T(2) = \mathsf{CVariables}_T(3) = \mathsf{CVariables}_T(4) = \emptyset, \tag{81}$$

following which

$$\mathsf{CVariables}_T = \mathsf{CVariables}_S(10) = \mathsf{CVariables}_S = \{\mathsf{goS}\} \tag{82}$$

can be figured out easily. For the extended reachability automaton of T as given in Figure 18, we append a self-loop of the event goS in the state $[1]$. Note that from Remark 2.3.3, self-looping goS in the state [] can be omitted for T. For S, we again follow Remark 2.3.3 in that we avoid self-looping goS in the state $[11]$ of the automaton depicted on the left side of Figure 15.

2.3.3 Immediate instructions

The access to writable variables generally constitutes control instructions that a process potentially executes. Yet, as pointed out in Question 3, it is sometimes undesired that these instructions are delayed arbitrarily when a process becomes busy. To address this problem, we recall that a process is started only if its precondition evaluates true. Thus, to answer Question 3, we optionally strengthen this semantic restriction so that some control instructions of a process are executed before its precondition is invalidated. Technically, for any process $n \in \mathsf{Processes}$, we optionally assign an integer value, which is referred to the *immediateness value*, to its controlled variable $(v, l) \in \mathsf{CVariables}(n)$ through the function

$$\mathsf{immediate}(v, l, n) \in \mathbb{N} \,\dot{\cup}\, \{\emptyset\}. \tag{83}$$

For $\mathsf{immediate}(v, l, n) \in \mathbb{N}$, the process n should execute $\sigma_{v,l,n}$ before $\mathsf{precond}(n)$ is invalidated. We also stipulate that if $\mathsf{immediate}(v, l, n) < \mathsf{immediate}(v', l', n)$, $\sigma_{v,l,n}$ must be executed *before* $\sigma_{v',l',n}$. If $\mathsf{immediate}(v, l, n) = \emptyset$, then no immediateness value is assigned to $(v, l) \in \mathsf{CVariables}(n)$. For convenience, we assume that for each process, the immediateness value of a controlled variable, if defined, is unique. This allows us to generate a unique event sequence $P_n \in \Sigma^*_{\mathsf{VAR}}$ which represents the instructions executed in a desired order for each process $n \in \mathsf{Processes}$. Thus, as soon as n is switched to the process state busy, we shall not allow invalidating $\mathsf{precond}(n)$ before the execution of P_n has been finished. This is realised by generating for each process n the regular expression

$$(\,\Sigma^*_{\mathsf{prio}} \cdot \mathsf{B}n \cdot P_n \cdot \Sigma^*_{\mathsf{prio}} \cdot \mathsf{I}n\,)^* \tag{84}$$

where $\Sigma_{\mathsf{prio}} \subseteq \Sigma_{\mathsf{VAR}}$ is defined by

$$\begin{aligned}\Sigma_{\mathsf{prio}} :=& \{\sigma_{v,l}, \sigma_{v,l,n'} \mid v \text{ is utilised in } \mathsf{precond}(n)\}\\ &\cup \{\sigma_{v,l,n} \mid (v, l) \in \mathsf{CVariables}(n)\}.\end{aligned} \tag{85}$$

Clearly, if Σ_{B} and Σ_{I} are to omit as suggested in Remark 2.2.2, $\mathsf{B}n$ and $\mathsf{I}n$ in (84) need to be substituted by $\Sigma^{\mathsf{TARGET}}_n$ and $\Sigma^{\mathsf{SOURCE}}_n$, respectively.

Remark 2.3.4. *Obviously, disallowing all variable events* $\{\sigma_{v,l}, \sigma_{v,l,n'} \mid v$ *is utilised in* $\mathsf{precond}(n)\}$ *is not necessary for guaranteeing that* $\mathsf{precond}(n)$ *is not invalidated. Alternatively, one could substitute this term in* (85) *by*

$$\{\sigma_{v,l} \mid \textit{executing } \sigma_{v,l} \textit{ invalidates } \mathsf{precond}(n)\}. \tag{86}$$

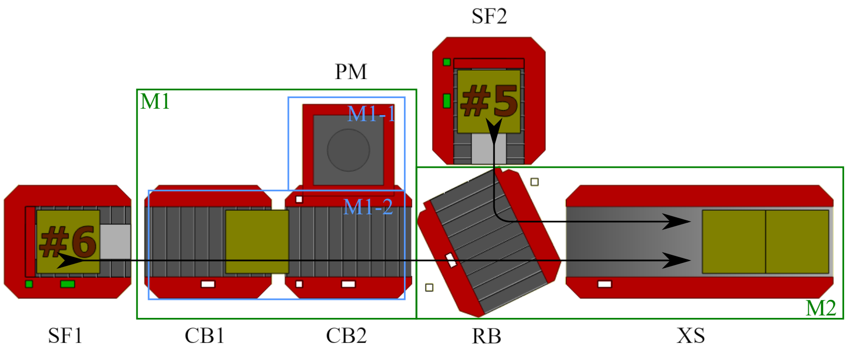

Figure 20: A production line example

However, the relative drawback of this alternative implementation is that if $\mathsf{precond}(n)$ *contains e.g. disjunction of equality propositions, then whether executing any* $\sigma_{v,l} \in \Sigma^{\mathsf{VAR}}$ *invalidates* $\mathsf{precond}(n)$ *depends on the current variable evaluation, which is rather cumbersome to figure out and renders the construction through* (84) *invalid.*

For the drill station example, consider the process with ID = 4 which is dedicated to activating the ventilator when the drill is operating. Note that without specifying immediate instructions, we may even tolerate such cases in which the ventilator activation is delayed until drilling has already finished. To disallow such cases to happen, we can refine the SBD by e.g. introducing new variables drill and ventilator with $\mathsf{range}(\mathsf{drill}) = \mathsf{range}(\mathsf{ventilator}) = \{\mathsf{on}, \mathsf{off}\}$ to indicate whether the drill or the ventilator is currently working, respectively. Afterwards, setting $\mathsf{drill} = \mathsf{off}$ as $\mathsf{precond}(4)$ and assigning $\mathsf{immediate}(4, \mathsf{ventilator}, \mathsf{on}) = 1$ (or any arbitrary natural number value) effectively restricts the SBD behaviour so that the ventilator must be turned on before the drill is turned on.

2.4 A practical example

So far, SBD semantics has been completely introduced. In the following, we design the control sequences of a production line through an SBD project with modular and hierarchical structure.

The production line is graphically depicted in Figure 20 which, as the plant, consists of two stack feeders (SFs), two conveyor belts (CBs), one processing machine (PM), one rotary table (RB) and one exit slide (XS). Besides, two operation buttons (OPs) are dedicated for user operations which are not represented in Figure 20. In addition, the intended usage of the plant is to transfer workpieces from SF1 and SF2 to XS. For the route from SF1 to XS, a workpiece is first transported via CB1 to CB2, where the workpiece must be processed by PM. Afterwards, the processed workpiece is sent to XS via RB. Note that RB can be oriented in either the west-east or the north-south orientation through rotation, and sending workpiece from CB2 to XS requires RB being in the west-east orientation. On the other hand, the route from SF2 to XS simply gathers a workpiece from SF2 to the west-east oriented RB and sends the workpiece to XS. Since multiple physical components are involved in the plant, it is desired to design the control programme within a modular and hierarchical structure. As illustrated in Figure 20, the plant is divided into two main modules M1 and M2, marked by green rectangles, where M1 further consists of two sub-modules $\mathrm{M1}-1$ and $\mathrm{M1}-2$, marked by blue rectangles. Note that both SFs are considered as being externally controlled and thus excluded from the SBD design.

We first list all involved variables in Table 2, including the plant components they belong to (comp.), possible values, variable descriptions and writability. Note that there are two "intern" variables P and M2_BUSY which belong to neither physical component. Instead, they are only internally utilised to organise the interaction between M1 and M2. Besides, we take the following conventions for brevity:

- Each unwritable variable corresponds to a sensor signal, indicating whether a specific location is occupied (by e.g. a workpiece). 0 means that the sensor is currently free, while 1 means being occupied.
- Each non-intern writable variable corresponds to an actuator signal. Value = 0 or 1 indicates that the corresponding actuator is idle or turned on, respectively. For those denoting a belt motor (*_BM as in Table 2), turning on the motor always drives the belt from west to east or from north to south.
- Each variable has a unique initial value, which is underlined.

By utilising the variables given in Table 2, five SBDs S_{PROC}, S_{TAKE}, S_{SEND}, S_1 and S_2 are constructed, which are depicted in Figures 21 and 22. For brevity, we take the following conventions for the graphical illustrations of SBDs in Figures 21 and 22.

- IDs of non-place nodes are hidden.
- All conditions are conjunction of equality propositions.

Table 2: Variables list of the production line example

comp.	variable	values	description	wrt.
CB1	CB1_BM	$\{\underline{0}, 1\}$	belt motor	yes
	CB1_WPS	$\{\underline{0}, 1\}$	workpiece sensor	no
CB2	CB2_BM	$\{\underline{0}, 1\}$	belt motor	yes
	CB2_WPS	$\{\underline{0}, 1\}$	workpiece sensor	no
PM	PM_PM	$\{-1, \underline{0}, 1\}$	positioning motor (1 = to south, 0 = stop, -1 = to north)	yes
	PM_PS+	$\{\underline{0}, 1\}$	south position sensor	no
	PM_PS-	$\{0 ,\underline{1}\}$	north position sensor	no
	PM_MOP	$\{\underline{0} ,1\}$	processing machine	yes
	PM_MRD	$\{0, \underline{1}\}$	ready to start processing machine	no
OP	OP1, OP2	$\{\underline{0}, 1\}$	operation button	no
RB	RB_BM	$\{\underline{0}, 1\}$	belt motor	yes
	RB_WPS	$\{\underline{0}, 1\}$	workpiece sensor	no
	RB_RM	$\{-1, \underline{0}, 1\}$	rotation motor (1 = clockwise, 0 = stop, -1 = counter-clockwise)	yes
	RB_SCW	$\{\underline{0}, 1\}$	orientation sensor, north-south position	no
	RB_SCCW	$\{0, \underline{1}\}$	orientation sensor, west-east position	no
XS	XS_WPS	$\{\underline{0}, 1\}$	workpiece sensor	no
intern	P	$\{\underline{0}, 1\}$	1 = PM has finished processing, 0 = otherwise	yes
	M2_BUSY	$\{\underline{0}, 1\}$	1 = M2 busy transporting workpiece, 0 = otherwise	yes

- For each SBD, its associated plant model and initial condition are directly given at the top of each SBD. Note that since S_1 and S_2 are root SBDs, their initial conditions are trivially true.
- A *non-invoker* process n has the controlled variable (v, l) if and only if $v := l$ appears in $\mathsf{postcond}(n)$. For denoting equality propositions, ":=" and "=" are semantically identical. Besides, all such controlled variables are immediate instructions where the execution order complies with the

top-to-bottom order in the corresponding figure. Controlled variables of an *invoker* process are never immediate instructions.

- For each *non-invoker* process n, we have $\mathsf{postcond}(n) = \mathsf{termcond}(n)$.

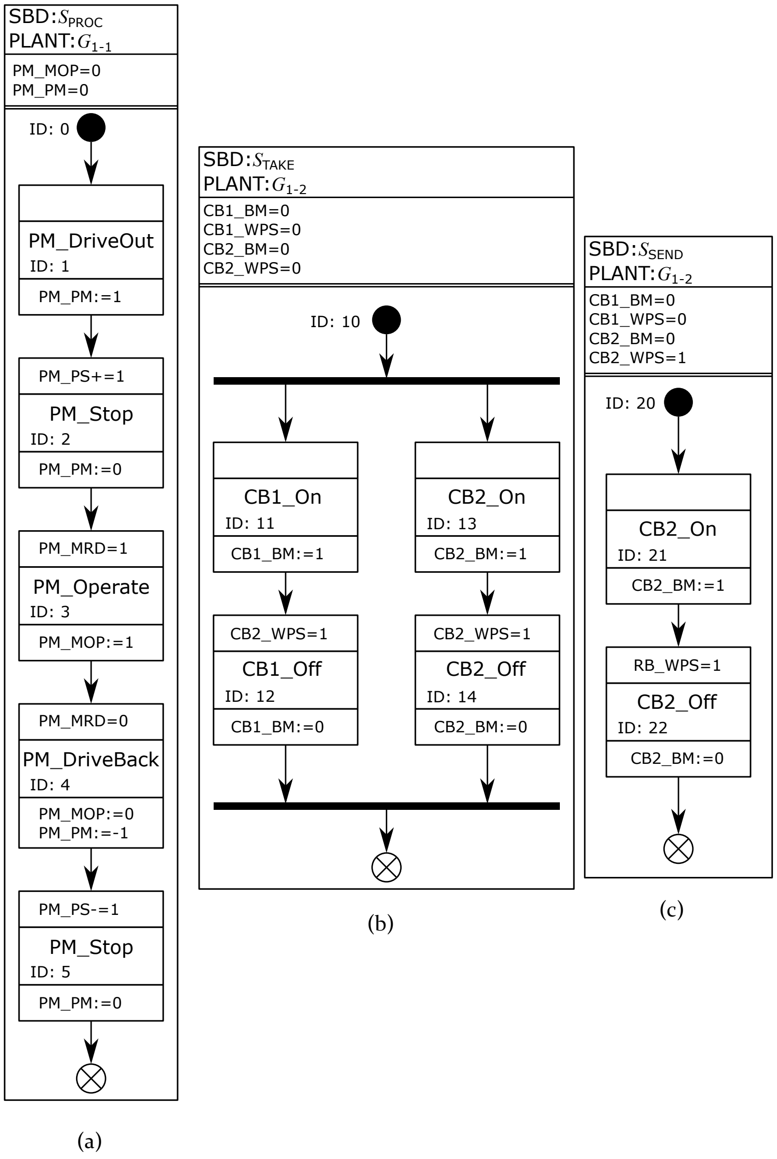

Figure 21: SBDs of modules $M1-1$ (a) and $M1-2$ (b,c)

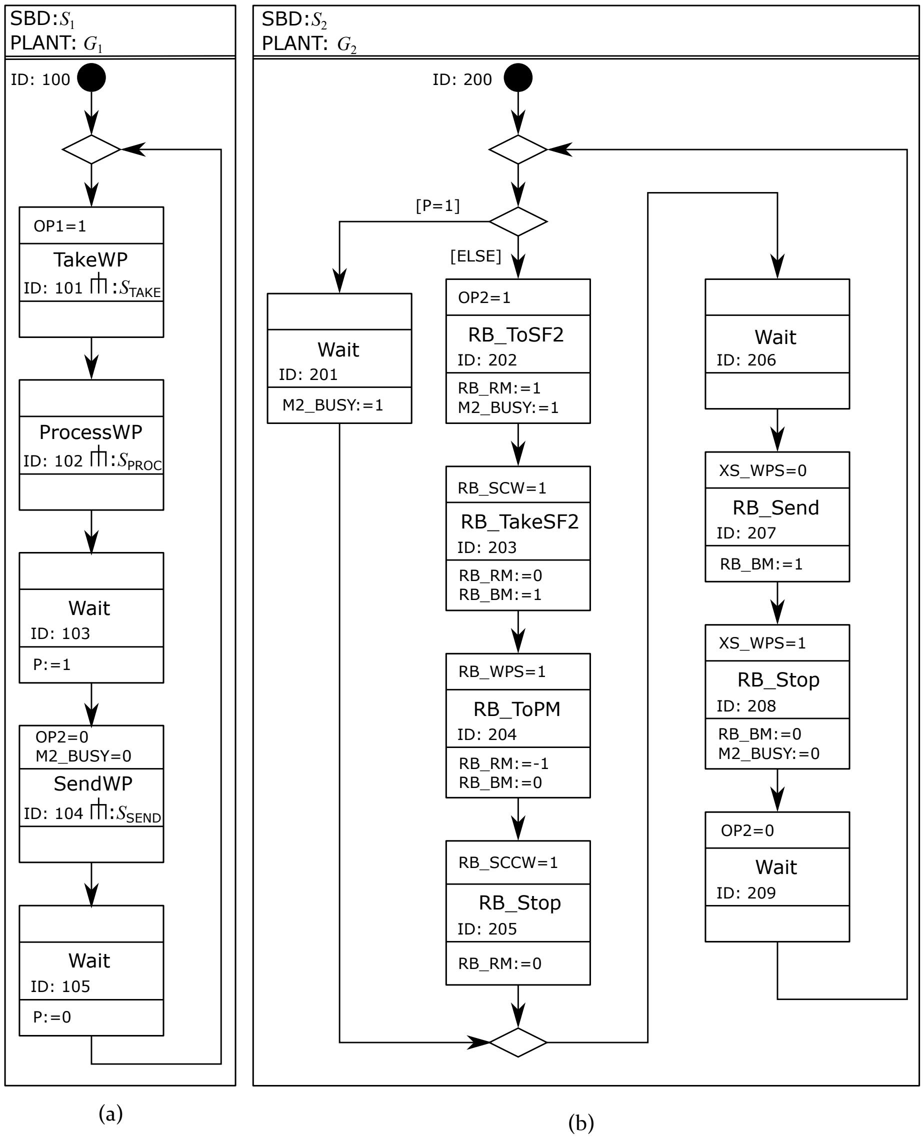

Figure 22: SBDs of modules M1 (a) and M2 (b)

The overall structure of the closed-loop behaviour is demonstrated in Figure 23. Each module constitutes a plant model which communicates with its associated SBD(s). Among all modules, the module M1 − 2 is associated with two SBDs S_{TAKE} and S_{SEND}, while each other model is associated with a single SBD. In the following, we explain the detailed functionality of each module. The corresponding plant models are given in Appendix A.

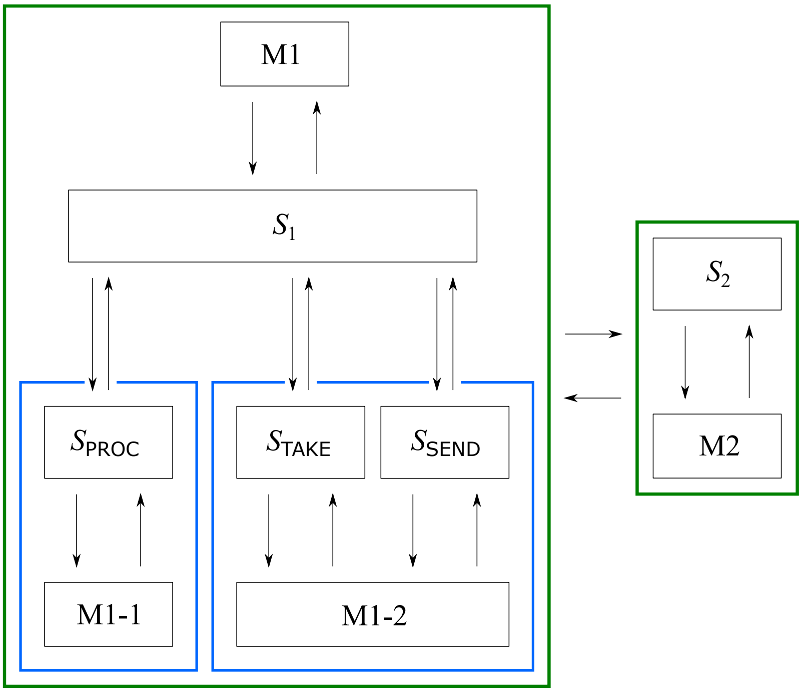

Figure 23: Structure of the modularised closed-loop behaviour

Module M1-1

Being associated with the SBD S_{PROC}, the module $\mathrm{M1}-1$ is responsible for workpiece processing. As shown in Figure 20, PM is initially located on the north side of CB2. When a workpiece is correctly positioned at CB2, PM drives out to the south position and positions the machine head above the workpiece. Afterwards, the workpiece is processed for a few seconds. Finally, PM drives back to the north position.

Module M1-2

Being associated with SBDs S_{TAKE} and S_{SEND}, the module $\mathrm{M1}-2$ handles the workpiece transport from SF1 to RB. Once S_{TAKE} is activated, both CB1 and CB2 are turned on until a workpiece arrives at the workpiece sensor of CB2. On the other hand, when S_{SEND} is activated, only CB2 will be turned on until RB receives the workpiece from CB2. Note that CB1_BM is also a local variable of S_{SEND}, but not controlled by any processes in S_{SEND}. Besides, activating S_{SEND} requires that a workpiece is actually available at CB2, which is indicated in its initial condition.

Module M1

Being associated with the SBD S_1, the module M1 organises the cooperation between $\mathrm{M1} - 1$ and $\mathrm{M1} - 2$. Generally, S_1 cyclically invokes SBDs S_{TAKE}, S_{PROC} and S_{SEND} when OP1 is pressed. In addition, after S_{PROC} has finished, the value of the internal variable P is set to 1 to indicate that a processed workpiece is ready to be sent to RB. If the module M2 is currently not busy with other workpiece transportation and OP2 is not pressed (indicating that no workpieces from SF2 is waiting for transport via RB), the processed workpiece will be transported through invoking S_{SEND}.

Module M2

Being associated with the SBD S_2, the module M2 is responsible for transporting workpieces from either CB2 or SF2 to XS via RB. To take a workpiece from CB2, M2 passively reads the value of P from M1 (note that P is not controlled in M2) and if $\mathsf{P} = 1$, RB stays in the west-east orientation and transports a (processed) workpiece from CB2 to XS. Otherwise, i.e. when $\mathsf{P} = 0$, RB turns clockwise to the north-south orientation whenever OP2 is pressed, indicating that SF2 is attempting to send a workpiece. Entering either route will directly set the internal variable $\mathsf{M2\_BUSY}$ to 1, which forbids M1 to invoke S_{SEND}. The value of $\mathsf{M2\_BUSY}$ is then set to 0 if XS has successfully received the workpiece. Note that since XS is designed to have maximal capacity, RB is allowed to send workpieces to XS only if $\mathsf{XS\_WPS} = 0$, i.e. the workpiece sensor at the entrance of XS is currently vacant.

Non-blockingness of SBDs

Following the translation procedure in Section 2.2, the global closed-loop behaviour of the production line example is described by five automata resulting from the five SBDs in Figures 21 and 22. In the current example, the two high-level SBDs S_1 and S_2 are cyclically structured with no terminal nodes utilised. In practice, it is important to ensure that both cyclic SBDs indeed repeat the cyclic execution indefinitely, i.e. for each SBD, there exists the possibility to proceed the execution at any state, i.e. to fire some subsequent hyper-edges. Such properties can be conveniently expressed by *non-blockingness*. A non-blocking system requires that in any reachable state, there exists the possibility to attain desired configurations in the future, which are often denoted by a set (or multiple sets) of *marking states* in the conventional automata and formal language theory (Cassandras and Lafortune, 2008).

Unfortunately, non-blockingness is very expensive to verify for modular systems (Cassandras and Lafortune, 2008; Malik, Streader et al., 2004), since conventionally, it again requires an explicit construction of the monolithic representation of the entire system, whose state space is generally in the exponential order w.r.t. the number of modules. To mitigate the high computational cost, one elegant approach is to utilise the so-called *compositional verification* (Flordal and Malik, 2009) which attempts to reduce the state space of each module before computing the overall composition. Recently, various contributions (Flordal and Malik, 2009; Pilbrow and Malik, 2015; Su et al., 2010; Ware and Malik, 2012) have utilised compositional verification for non-blockingness check and shown convincing results. However, as far as the author's knowledge, they all assume that the automata are synchronised through the ordinary synchronous composition (Cassandras and Lafortune, 2008; Milner, 1989). By referring to Section 2.2.4, this is unfortunately not the case for our SBD translation procedure since the monolithic closed-loop behaviour should be represented by the *shaped* synchronous composition of all automata. In this situation, it can be shown that most of the available results w.r.t. compositional non-blockingness verification need to be modified, which will be intensively discussed in the following chapter in detail.

Concluding remarks

To prepare for the formal verification of SBD projects, we have focused on translating SBDs into finite automata based on formalising SBD semantics in the current section. Basically, SBD semantics is represented by token propagation on an extended Petri-net in that processes in an SBD are referred to as places and hyper-edges are considered as (Petri-net) transitions. Since the state of a Petri-net is generally distributed over its current token configuration, it is natural to construct the reachability graph of an SBD so that the configuration can be determined on a per-state basis. In this context, the reachability graph is synonymous to an automaton whose transitions are labelled by hyper-edges. Furthermore, SBDs carry some features in addition to ordinary Petri-nets, i.e. guard conditions and process states, which restrict the free propagation of tokens. These were correspondingly represented by a set of constraint automata. Finally, by taking a plant model into consideration as well, the local closed-loop behaviour can be constructed by taking their synchronous composition. For a complicated project consisting of multiple SBDs, the translation procedure effectively generate for each SBD one automaton.

One specific feature of the translation result is that all events carry priority attributes. Particularly, if a hyper-edge is considered fireable, it must be fired immediately without waiting for other events, especially those generated by variable value changes. If the global behaviour is represented by a single automaton, since transitions with lower priority can be simply removed (which is also referred to as shaping) if some high-priority events are active. Afterwards, properties such as non-blockingness can be easily verified through e.g. enumeration-based reachability analysis. However, for complicated systems with multiple SBDs, SBD semantics stipulates that the priorities have global effects; namely, high-priority events in one module restrict the occurrence of low-priority events in other modules. In this context, challenges will arise when exploiting advanced verification techniques for modular systems, e.g. compositional verification. In the following chapter, we address the problem of compositional non-blockingness verification when events carry priority attributes and show verification results of the production line example.

3 Compositional verification with prioritised events

At the end of the last chapter, we briefly introduced the concept of non-blockingness. Generally, the non-blockingness of a single automaton can be simply verified by e.g. enumerating reachable states and check their backward reachability from desired configurations (a.k.a. co-reachability). However, when handling modular systems, we observe that non-blockingness generally cannot be reasoned in a modular fashion. In particular, the synchronisation of a family of non-blocking automata is not always non-blocking, which can be seen from e.g. the well-known *dining philosophers problem* (E. Dijkstra, 1971). Thus, the straightforward way to verify the non-blockingness of a modular system is again to construct its monolithic representation, which suffers from the notorious *state explosion problem*, i.e. the overall state count grows exponentially w.r.t. the count of modules. One well-established approach addressing this problem is the *compositional verification* (Flordal and Malik, 2009). Inspired by the *testing theory* (Brinksma et al., 1995; Natarajan and Cleaveland, 1995) originating from process algebra (Milner, 1989), compositional verification applies abstractions on each involved automaton in a modular system, which reduces the state count of each module by typically utilising their private events while preserving the property of interest, e.g. non-blockingness (Malik, Streader et al., 2004), from a global perspective. Afterwards, a strategically chosen set of automata are substituted by their composition. This substitution potentially renders more events private, which enables further applicability of abstractions. The abstraction-composition cycle is thus iteratively performed until only one automaton is left, whose non-blockingness coincides with that of the monolithic representation. Recently, compositional verification has been successfully applied in various contributions addressing the non-blockingness verification problem; see e.g. (Flordal and Malik, 2009; Malik, 2015; Pilbrow and Malik, 2015; Su et al., 2010; Ware and Malik, 2012). Several other properties, e.g. controllability (Flordal and Malik, 2009) and opacity (Mohajerani and Lafortune, 2020), can be addressed by compositional verification as well by converting them into non-blockingness verification. Besides, (Malik and Leduc, 2013; Ware and Malik, 2013) utilised compositional verification to check the generalised non-blockingness (Malik and Leduc, 2008), which is a weaker variant of the ordinary non-blockingness, and (Lennartson et al., 2020) showed the applicability of compositional verification to any temporal logical property within CTL*-X. On the other hand, it is worth mentioning that the idea of composition verification can be adapted in the Supervisory Control Theory as well, where the non-blockingness of the entire system is usually required; see e.g. (Malik and Teixeira, 2016; Mohajerani, Malik et al., 2014; Mohajerani, Malik et al., 2017).

In the current chapter, we investigate the possibility to extend available results w.r.t. compositional verification to the situation where events are all *prioritised*, i.e. each event has a *priority value*. In any state, events with lower priority are disabled if any event with higher priority is currently active, which is referred to as *preemption*. In particular, we stipulate that event priorities influence the global behaviour, i.e. high-priority events in one module also preempt low-priority events in other modules. This kind of system set-up was closely related to some variants of process algebra with prioritised events (Cleaveland et al., 2007; Lüttgen, 1998) and can be utilised to handle the translation result of an SBD project; see Section 2.2.4. In fact, not only SBDs, various other popular modelling languages, e.g. ADs (R. Eshuis, 2006) and Grafcet (Provost, J.-M. Roussel et al., 2011), exhibit similar behaviour as well in that upon qualifying some guard condition, the system shall proceed to subsequent tasks immediately. In addition, another use-case where event priority arises is when implementing modular automata synthesised by formal methods as control programmes. This typically includes the following two sub-cases: (i) at some state where multiple control instructions (or internal operations) are active, the choice of the action to execute is sometimes not fully random. Besides, (ii) when the execution of some instruction and the occurrence of some sensor event are both possible in some state, the executor typically takes the action immediately without “waiting” for the sensor event (Qamsane et al., 2016). The latter one can be seen as a kind of weak timed behaviour which is closely related to timed discrete event systems; see e.g. (Brandin and W. M. Wonham, 1994).

The content of this section is extended from (Tang and Moor, 2024) in that more technical details, such as algorithm complexity, is included. This section is organised as follows. Preliminaries and notation conventions are clarified in Section 3.1. Section 3.2 introduces the abstraction rules for compositional non-blockingness verification when taking prioritised events into consideration. The abstraction rules are applied to the complete compositional verification procedure introduced in Section 3.3, which is tested by several practical examples in the final section, including the SBD example constructed previously in Section 2.4.

3.1 Preliminaries

3.1.1 Prioritised events

Consider a universe of *symbols* $\mathfrak{E}$ also referred to as *events*, which are the basic elements to represent discrete-event dynamics. Besides, a *string* is a finite sequence of events. The *Kleene's closure* of a set of events $A \subseteq \mathfrak{E}$ is denoted A^* which is the set of all strings constructed by events in A, including the empty string $\epsilon \notin \mathfrak{E}$. Note that $\epsilon s = s = s\epsilon$ holds for any string s. In some contexts, the notation $(\cdot)^+$ is utilised as well to conveniently exclude the empty string from a Kleene's closure, i.e. $A^+ := A^* - \{\epsilon\}$. The *concatenation* of two strings s and t is denoted st. Besides, for two strings s and r, s is considered a *prefix* of r if there exists some string t so that $r = st$, denoted $s \leqslant r$. Besides, a *priority value* is assigned to each event. This is a means of representing execution semantics, e.g. when confronting a choice of executing either of two events with different priority,[1] the executor should always choose the one with higher priority. We also say that all events in the current framework are *prioritised*. In this regard, the *priority assignment function*

$$\mathsf{prio} : \mathfrak{E} \to \mathbb{N} \tag{87}$$

is formally utilised to denote the priority of each event. In particular, priorities are read as ordinal numbers, i.e. $1 \in \mathbb{N}$ is considered the *first* priority, $2 \in \mathbb{N}$ the *second* priority, etc. As a greater ordinal number denotes a lower priority, 1 is the unique highest priority. Thus, when writing e.g. $\mathsf{prio}(\sigma) < \mathsf{prio}(\rho)$, we intend to show that the priority of σ is *higher* than that of ρ. For convenience, the following notations are used for any event set $A \subseteq \mathfrak{E}$:

- events with priority higher (or not lower) than $n \in \mathbb{N}$ within A
  $A^{<n} := \{\, \alpha \in A \,|\, \mathsf{prio}(\alpha) < n \,\}$;
  $A^{\leq n} := \{\, \alpha \in A \,|\, \mathsf{prio}(\alpha) \leq n \,\}$;
- events with priority higher (or not lower) than $\mathsf{prio}(\alpha)$ for $\alpha \in \mathfrak{E}$ within A
  $A^{<\alpha} := A^{<\mathsf{prio}(\alpha)}$;
  $A^{\leq\alpha} := A^{\leq\mathsf{prio}(\alpha)}$;
- the lowest priority value within A
  $$\mathsf{lo}(A) := \begin{cases} \max\{\, \mathsf{prio}(\alpha) \,|\, \alpha \in A \,\} & \text{if } A \neq \emptyset; \\ 1 & \text{if } A = \emptyset. \end{cases}$$

[1] This does not necessarily indicate that the priority value of each event is unique.

In process algebra, representing internal behaviour which are irrelevant to the synchronisation with external systems is of particular interest, since it enables various system abstraction techniques. Internal behaviour is technically represented by *silent events* $\Upsilon \subset \mathfrak{E}$. On the other hand, events in $\mathfrak{E} - \Upsilon$ are considered *regular* and the terminology of *alphabet* is utilised to denote any *finite* regular event set $\Sigma \subset \mathfrak{E} - \Upsilon$. While regular events are shown explicitly to the external environment for synchronisation, silent events are anonymous for the external environment. Regarding the priority assignment, it suffices to let Υ be such that each priority value $n \in \mathbb{N}$ is bijectively mapped to one event in Υ in order to represent local behaviour with different priorities. This motivates us to symbolically represent each silent event $\tau \in \Upsilon$ where $\mathsf{prio}(\tau) = n$ with

$$\tau \equiv \tau_{(n)} \tag{88}$$

and we have

$$\Upsilon := \{\, \tau_{(n)} \,|\, n \in \mathbb{N} \,\}. \tag{89}$$

Most prominently, the current set-up of silent events guarantees that each regular event has a counterpart silent event with the same priority, which is one of the fundamental prerequisite for abstraction. Formally, a *hiding map* $\mathsf{hide} : (\mathfrak{E} - \Upsilon) \to \Upsilon$ is defined by

$$\mathsf{hide}(\sigma) = \tau_{(\mathsf{prio}(\sigma))} \tag{90}$$

for each $\sigma \in \mathfrak{E} - \Upsilon$. This set-up is also utilised in (Lüttgen, 1998) and constitutes an extension of the more common *single distinguished silent event* $\Upsilon = \{\tau\}$ in the ordinary context without prioritised events; see e.g. (Flordal and Malik, 2009; Milner, 1989). In this regard, we utilise *natural projection* $\mathsf{p} : \mathfrak{E}^* \to (\mathfrak{E} - \Upsilon)^*$ to remove all silent events from any string $s \in \mathfrak{E}^*$ (Cassandras and Lafortune, 2008). Formally, natural projection is iteratively defined by

$$\mathsf{p}(\epsilon) = \epsilon; \tag{91}$$

$$\mathsf{p}(s\alpha) = \begin{cases} \mathsf{p}(s) & \text{if } s \in \mathfrak{E}^*, \alpha \in \Upsilon; \\ \mathsf{p}(s)\alpha & \text{if } s \in \mathfrak{E}^*, \alpha \in \mathfrak{E} - \Upsilon. \end{cases} \tag{92}$$

3.1.2 Finite automata

Definition 3.1.1. *A* finite automaton *is a tuple $G = \langle Q, \Sigma, \to, Q^\circ, M \rangle$ where*

- *Q is the finite state set;*

- *Σ is the alphabet;*
- *$\rightarrow \subseteq Q \times (\Sigma \cup \Upsilon) \times Q$ is the transition relation;*
- *$Q^\circ \subseteq Q$ is the set of initial states;*
- *$M \subseteq 2^\Sigma$ is the marking set.*

The marking set in the automaton tuple is a generalisation of non-blockingness and is similar to the so-called *coloured marking* in the multitasking supervisory control theory (Hering de Queiroz et al., 2005); see Definition 3.1.2. Note that silent events are not legit to carry marking information. Besides, in the following, we utilise the notation $A_G := \Sigma \cup \Upsilon$ to denote the union of the alphabet of G with the silent event set. The subscript $(\cdot)_G$ of A_G is omitted if it is clear from the context. Finally, note that automata are not required to be deterministic; namely, it is possible that for $\alpha \in A$ and $x, y, y' \in Q$ where $y \neq y'$, both $(x, \alpha, y) \in \rightarrow$ and $(x, \alpha, y') \in \rightarrow$ hold. Generally, a control system does behave deterministically, while abstraction may introduce non-determinism.

We use the infix notation $x \xrightarrow{\alpha} y$ to denote $(x, \alpha, y) \in \rightarrow$ and the infix notation is iteratively extended to string-valued labels; namely, (i) let $x \xrightarrow{\epsilon} x$ for all $x \in Q$ and (ii) $x \xrightarrow{s\alpha} z$ for all $x,\ z \in Q$, $s \in A^*$ and $\alpha \in A$ if $x \xrightarrow{s} y$ and $y \xrightarrow{\alpha} z$ for some $y \in Q$. Moreover, we write $X \xrightarrow{s} Y$ for $X, Y \subseteq Q$ whenever there exist $x \in X$ and $y \in Y$ so that $x \xrightarrow{s} y$. The set-theoretic complement of the transition relation is denoted $\not\rightarrow$, i.e., $X \overset{s}{\not\rightarrow} Y$ is interpreted as such that for any $x \in X$ and $y \in Y$, $(x, s, y) \notin \rightarrow$ holds. $X \xrightarrow{s}$ and $G \xrightarrow{s}$ stand for $X \xrightarrow{s} Q$ and $Q^\circ \xrightarrow{s} Q$, respectively. For a state x in an automaton G, the set of *active events in x* is given by

$$G(x) := \{\, \alpha \in A \,|\, x \xrightarrow{\alpha} \}. \tag{93}$$

Finally, a *trace* is a sequence of alternating states and events, i.e. in the form of

$$x_0 \xrightarrow{\alpha_1} x_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} x_k. \tag{94}$$

We again introduce several convenient notations for brevity.

- active events in state x with priority higher (or not lower) than $n \in \mathbb{N}$
  $G^{<n}(x) := \{\alpha \in G(x) \,|\, \mathsf{prio}(\alpha) < n\}$;
  $G^{\leq n}(x) := \{\alpha \in G(x) \,|\, \mathsf{prio}(\alpha) \leq n\}$;
- silent active events in state x (with priority higher or not lower than $n \in \mathbb{N}$)
  $G_{\mathsf{slnt}}(x) := G(x) \cap \Upsilon$;

$G^{<n}_{\text{slnt}}(x) := G^{<n}(x) \cap \Upsilon;$
$G^{\leq n}_{\text{slnt}}(x) := G^{\leq n}(x) \cap \Upsilon;$

- regular active events in state x (with priority higher than $n \in \mathbb{N}$)
  $G_{\text{rglr}}(x) := G(x) - \Upsilon;$
  $G^{<n}_{\text{rglr}}(x) := G^{<n}(x) - \Upsilon;$
  $G^{\leq n}_{\text{rglr}}(x) := G^{\leq n}(x) - \Upsilon;$
- abstract transition relation $\Rightarrow \subseteq Q \times \Sigma^* \times Q$
  $x \overset{s}{\Rightarrow} y$ for $s \in \Sigma^*$ if and only if there exists $s' \in A^*$ so that $\mathsf{p}(s') = s$ and $x \xrightarrow{s'} y$;
- concatenation of different types of transitions
  $x \overset{s}{\rightarrow}\overset{s'}{\Longrightarrow} y$ if and only if there exists some state z so that $x \overset{s}{\rightarrow} z$ and $z \overset{s'}{\Longrightarrow} y$.

Regarding the liveness property of an automaton, its *non-blockingness* is of specific interest which states that desired system configurations are persistently reachable in the future in any reachable state (Cassandras and Lafortune, 2008). Particularly for SBD verification, it is desired that each SBD has the opportunity to proceed by e.g. firing some critical hyper-edges. This motivates us to classify desired configurations into different categories and we require that reaching each type of the desired configurations should be persistently possible. This idea is comparable with the *strong* non-blockingness utilised in the multitasking supervisory control theory (Hering de Queiroz et al., 2005).

Definition 3.1.2. *Given an automaton $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$, a state $x \in Q$ is* reachable *if there exists $s \in \Sigma^*$ so that $G \overset{s}{\Rightarrow} x$. A state $x \in Q$ is* co-reachable *if for all $\Omega \in M$, there exists $t \in \Sigma^*$ and $\omega \in \Omega$ so that $x \overset{t\omega}{\Longrightarrow}$. G is* non-blocking *if all its reachable states are co-reachable.*

Note that a regular event, say $\omega \in \Sigma$, can appear in multiple event sets in the marking set. If executing ω is possible in the future, all event sets in the marking set containing ω are qualified to achieve the non-blockingness. Consider the following example.

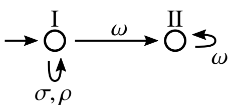

Figure 24: An example for non-blockingness

Example 3.1.1. *Consider the automaton $G = \langle Q, \Sigma, \to, Q^\circ, M\rangle$ given in Figure 24 with $\Sigma = \{\sigma, \rho, \omega\}$. In addition, two possibilities of defining the marking set M are discussed: if $M = \{\{\sigma, \omega\}, \{\rho, \omega\}\}$, then G is non-blocking provided ω appears in both event sets of the marking set, and ω is executable in the future for both states. On the other hand, if $M = \{\{\sigma, \omega\}, \{\rho\}\}$, G turns out to be blocking since ρ cannot be executed any more once state* II *is reached.*

We now define how prioritised events influence the execution semantics of automata. In any state with multiple active events, transitions labelled by events with lower priority should be disabled. In this regard, we say the lower-priority events are *preempted*. This is formally illustrated by *shaping* an automaton with the *shaping operator*.

Definition 3.1.3. *Given an automaton $G = \langle Q, \Sigma, \to, Q^\circ, M\rangle$, the* shaping operator $\mathcal{S}(\cdot)$ *is defined as such that $\mathcal{S}(G) := \langle Q, \Sigma, \to^{\mathcal{S}}, Q^\circ, M\rangle$ where*

$$x \overset{\alpha}{\to}^{\mathcal{S}} y \text{ if and only if } x \overset{\alpha}{\to} y \text{ and } G^{<\alpha}(x) = \emptyset. \tag{95}$$

In any state of an automaton after shaping, only the transitions labelled by events with the highest priority among all active events are preserved. Note that after shaping an automaton, some states may become unreachable and can be directly removed.

3.1.3 Synchronous composition and non-conflictingness

In practice, large-scale systems are commonly decomposed into modular pieces. All modules are cooperatively operated under certain synchronisation semantics. Suppose two automata with respective alphabets Σ_1 and Σ_2 are to synchronise. Their plain synchronised behaviour complies with their *synchronous composition* (Milner, 1989). In particular, events in $\Sigma_1 \cap \Sigma_2$ are *shared* which should be executed synchronously, while all other events in $\Sigma_1 \cup \Sigma_2 \cup \Upsilon$ are *private* which are asynchronously executed.

Definition 3.1.4. *Given two automata $G_1 = \langle Q_1, \Sigma_1, \to_1, Q_1^\circ, M_1\rangle$ and $G_2 = \langle Q_2, \Sigma_2, \to_2, Q_2^\circ, M_2\rangle$, their synchronous composition is defined by*

$$G_1 \parallel G_2 := \langle Q := Q_1 \times Q_2, \Sigma := \Sigma_1 \cup \Sigma_2, \to, Q^\circ := Q_1^\circ \times Q_2^\circ, M := M_1 \cup M_2\rangle, \tag{96}$$

where $\to \subseteq Q \times \Sigma \times Q$ is defined by

$$(x_1, x_2) \overset{\alpha}{\to} (x_1', x_2') \quad \text{if} \quad \alpha \in \Sigma_1 \cap \Sigma_2,\ x_1 \overset{\alpha}{\to}_1 x_1' \text{ and } x_2 \overset{\alpha}{\to}_2 x_2'; \tag{97}$$

$$(x_1, x_2) \xrightarrow{\alpha} (x_1', x_2) \quad \textit{if} \quad \alpha \in (\Sigma_1 - \Sigma_2) \cup \Upsilon \textit{ and } x_1 \xrightarrow{\alpha}_1 x_1'; \tag{98}$$
$$(x_1, x_2) \xrightarrow{\alpha} (x_1, x_2') \quad \textit{if} \quad \alpha \in (\Sigma_2 - \Sigma_1) \cup \Upsilon \textit{ and } x_2 \xrightarrow{\alpha}_2 x_2'. \tag{99}$$

A transition $(x_1, x_2) \xrightarrow{\alpha} (x_1', x_2')$ *is* driven by G_1 *if* $x_1 \xrightarrow{\alpha}_1 x_1'$ *in* G_1.

Clearly, since state names do not contribute to system behaviour, synchronous composition is considered commutative and distributive, i.e. $G_1 \parallel G_2$ = $G_2 \parallel G_1$ and $G_1 \parallel (G_2 \parallel G_3) = (G_1 \parallel G_2) \parallel G_3$. The synchronisation of a family of automata $(G_1)_{1 \leq i \leq k}$, i.e. $D = G_1 \parallel G_2 \parallel \cdots \parallel G_n$, is commonly referred to as a *modular system* while each G_i is referred to as a *module*. From the ordinary context where event prioritising is not considered, the non-blockingness of D is commonly referred to as the *non-conflictingness* of all modules. At this stage, it is worth mentioning that the non-blockingness of one module, or even each module, cannot imply non-conflictingness and vice versa. Thus, the conventional approach to checking non-conflictingness is to explicitly construct D, which is of exponential order w.r.t. the count of modules. This problem can be decently addressed by *compositional verification*. The core of compositional verification is to apply suitable abstraction on each module while the non-conflictingness is preserved. To this end, based on the testing theory framework (Brinksma et al., 1995; Natarajan and Cleaveland, 1995), the concept of *conflict equivalence* was introduced in (Malik, Streader et al., 2004) which sufficiently implies the preservation of non-conflictingness; namely, substituting any module with its conflict equivalent abstraction does not influence the non-conflictingness. After abstraction, the verification procedure alternates to the composition of a strategically chosen set of automata. This procedure is then iteratively performed until only one automaton is left, whose non-blockingness coincides with the non-conflictingness of the original modular system D.

In the scope of the current dissertation, it is stipulated that event prioritising influences the behaviour of the entire modular system, i.e. high-priority events in one module preempt low-priority events in other modules as well. In this context, the non-blockingness of the modular system *after* shaping, i.e. $\mathcal{S}(D)$, is of our interest and is exactly the property for which an efficient verification procedure is desired.

Definition 3.1.5. *A family* $(G_i)_{1 \leq i \leq k}$ *of automata is* non-conflicting w.r.t. prioritised events *if and only if* $\mathcal{S}(G_1 \parallel G_2 \parallel \cdots \parallel G_k)$ *is non-blocking.*

In the remainder of the current chapter, the terminology *non-conflicting* is concisely utilised to denote *non-conflicting w.r.t. prioritised events*. At this stage, following the idea of compositional verification, we consider again the entire modular system

$$\mathcal{S}(\underbrace{G_1}_{:=G} \parallel \underbrace{G_2 \parallel \cdots \parallel G_k}_{:=H}) \,. \tag{100}$$

Since synchronous composition is commutative and distributive, we choose $G_1 =: G$ as the automaton to abstract. Correspondingly, H in (100) is also referred to as the *synchronisation rest part*, or simply the *rest part*. Let G' be an abstraction of G, we obviously expect that $\mathcal{S}(G \parallel H)$ is non-blocking if and only if $\mathcal{S}(G' \parallel H)$ is non-blocking.

One fundamental abstraction is provided by *transition hiding*, which is technically referred to as replacing a regular transition label by its silent counterpart.

Definition 3.1.6. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be an automaton and let $t = (x, \sigma, y) \in \rightarrow$ be any transition in G.* Hiding t in G *results in an automaton $G/_t = \langle Q, \Sigma, \rightarrow_t, Q^\circ, M\rangle$ where*

$$\rightarrow_t := (\rightarrow - \{t\}) \cup \{(x, \mathsf{hide}(\sigma), y)\}. \tag{101}$$

When synchronising an automaton G with another automaton H, we say a transition t in G is *hidable w.r.t.* H if hiding t in G preserves the non-conflictingness.

Definition 3.1.7. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ and $H = \langle Q_H, \Sigma_H, \rightarrow_H, Q^\circ_H, M_H\rangle$ be two automata. A transition $t \in \rightarrow_G$ in G is* hidable w.r.t. *H if and only if*

$$G \text{ and } H \text{ are non-conflicting} \quad \Leftrightarrow \quad G/_t \text{ and } H \text{ are non-conflicting.} \tag{102}$$

At a first glance, any transition labelled by a regular private event seem to be hidable. However, special care should be taken to the marking set as it may also include some private events. Hiding all transitions labelled by private events carrying marking information is clearly not legit.

Proposition 3.1.8. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ and $H = \langle Q_H, \Sigma_H, \rightarrow_H, Q^\circ_H, M_H\rangle$ be two automata and let $t = (x, \sigma, y) \in \rightarrow_G$ be a transition in G where $x, y \in Q_G$ and $\sigma \in \Sigma_G - \Sigma_H$. If for all $\Omega_G \in M_G$, $\sigma \notin \Omega_G$, then t is hidable w.r.t. H.*

Proposition 3.1.8 conservatively suggests that transitions labelled by events with marking information should never be hidden. Nevertheless, some of such transitions are indeed hidable if their future behaviour is within some specific structure. We resume to this topic later in Proposition 3.3.1 after a deeper dive into synchronisation with prioritised events in the next section.

Note that hiding itself does not require an explicit representation or any specific structure from the rest part, which is a prominent feature we shall generally require for all abstraction rules. In particular, this concept can be explicitly guaranteed from the definition of *conflict equivalence w.r.t. prioritised events*. This is inspired by the conflict equivalence in the ordinary context (Malik, Streader et al., 2004) where automata are synchronised through the ordinary synchronous composition.

Definition 3.1.9. *Two automata G_1 and G_2 are* conflict equivalent w.r.t. prioritised events*, denoted $G_1 \simeq^{\mathcal{S}} G_2$, if for any automaton T, it holds that*

$$G_1 \text{ and } T \text{ are non-conflicting} \quad \Leftrightarrow \quad G_2 \text{ and } T \text{ are non-conflicting}.$$

In the remainder of this chapter, *conflict equivalence* concisely stands for *conflict equivalence w.r.t. prioritised events*. In particular, an abstraction of G, say G', is a *conflict-preserving abstraction of G* if $G' \simeq^{\mathcal{S}} G$. Note that conflict equivalence does not require any information about the rest part (even its alphabet), which implies that substituting G in (100) by an automaton G' with $G' \simeq^{\mathcal{S}} G$ indeed preserves the non-conflictingness, i.e.

$$\mathcal{S}(G \parallel H) \text{ is non-blocking} \quad \Leftrightarrow \quad \mathcal{S}(G' \parallel H) \text{ is non-blocking}.$$

Finally, it is worth mentioning that there is no unique minimal conflict-preserving abstraction of an arbitrarily given automaton. This can be seen from (Flordal and Malik, 2006) where an example in the ordinary context is given. This obviously applies to conflict equivalence (w.r.t. prioritised events) as well by simply assuming that all events have the same priority. Hence, developing conflict-preserving abstraction rules is valuable to address the compositional non-blockingness verification problem w.r.t. prioritised events.

3.2 Conflict-preserving abstraction rules

In this section, various conflict-preserving abstraction rules are developed for compositional verification. To this end, some specific definitions and observations w.r.t. prioritised events are first to clarify. We begin with the introduction of the Υ-*shaping* operator.

Definition 3.2.1. *Given an automaton $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$, the* Υ-shaping operator $\mathcal{S}_\Upsilon(\cdot)$ *is defined by* $\mathcal{S}_\Upsilon(G) := \langle Q, \Sigma, \rightarrow^{\mathcal{S}_\Upsilon}, Q^\circ\rangle$ *where*

$$x \xrightarrow{\alpha}{}^{\mathcal{S}_\Upsilon} y \text{ if and only if } x \xrightarrow{\alpha} y \text{ and } G^{<\alpha}_{\text{slnt}}(x) = \emptyset. \tag{103}$$

An automaton G is Υ-shaped *if and only if $G = \mathcal{S}_\Upsilon(G)$.*

Definition 3.2.1 introduces a "partial" shaping operator which only shapes transitions using silent events. Generally, the normal shaping operation does not commute with synchronous composition, i.e. we cannot shape an individual module locally before the overall synchronous composition is constructed since a shared high-priority event in one module may be deactivated by other modules. Nevertheless, we can always partially shape an automaton using Υ-shaping since silent events can never be disabled by synchronisation. Thus, for any $\tau \in \Upsilon$ and $\alpha \in A$ so that $x \xrightarrow{\tau}$ and $x \xrightarrow{\alpha}$ for some state x with $\mathsf{prio}(\tau) < \mathsf{prio}(\alpha)$, the latter transition will never be executed as long as shaping will eventually be performed, since each time when x is visited, either τ or some event with priority higher than τ must be active. This observation is illustrated by the following lemma.

Lemma 3.2.2. *For any two automata G_1 and G_2, it holds that*

$$\mathcal{S}(G_1 \parallel G_2) = \mathcal{S}(\mathcal{S}_\Upsilon(G_1) \parallel G_2). \tag{104}$$

Since synchronous composition is commutative and associative, it follows immediately that performing Υ-shaped on any module beforehand does not influence the monolithic representation of the entire system. This is a simple yet powerful conflict-preserving abstraction rule as well. In addition, Υ-shaping is indeed conflict-preserving from Lemma 3.2.2. In the remainder of this chapter, it is consistently assumed that the automaton to abstract is Υ-shaped, which simplifies many of the statements and definitions.

Remark 3.2.1. *Obviously, if the alphabet of the rest part is available, Υ-shaping can more aggressively be uniformly substituted by "private shaping", i.e. shaping using all private events including those regular events not appearing in the rest part. This substitution clearly yields a more remarkable state reduction. Note that, similar to hiding, "private shaping" is not conflict-preserving as well, but indeed preserves the non-conflictingness under a given rest part.*

We now shift our focus to silent loops where all transitions leaving the loop are labelled by regular events. Such silent loops are referred to as *live-locks* and have specific semantic meaning when considering prioritised events.

Definition 3.2.3. *Given a Υ-shaped automaton $G = \langle Q, \Sigma, \to, Q^\circ, M\rangle$, an n-*live-lock *in G is a set of states $X \subseteq Q$ where for all $x \in X$,*

(L1) $G_{\text{slnt}}(x) \neq \emptyset$;

(L2) for all $\tau \in G_{\text{slnt}}(x)$, $x \xrightarrow{\tau} x'$ implies $x' \in X$;

(L3) for all $x, y \in X$, there exists a trace $x \xrightarrow{\alpha_1} x_1 \xrightarrow{\alpha_2} x_2 \xrightarrow{\alpha_3} \cdots x_k \xrightarrow{\alpha_k} y$, where $x_i \in X$, $\alpha_i \in \Upsilon$ for all $i = 1, 2, \dots, k$,

and

(L4) $\mathsf{lo}(\cup_{x' \in X} G_{\text{slnt}}(x')) = n$.

We also concisely write α-live-lock to denote $\mathsf{prio}(\alpha)$-live-lock where $\alpha \in A$. Technically, a live-lock is a non-trivial (i.e. with at least one transition) silent Strongly Connected Component (SCC) (Aho et al., 1974) where neither state can leave this SCC through executing a silent transition. Due to prioritised events, a live-lock may indefinitely *trap* the behaviour of the rest part, i.e. when in some n-live-lock of an automaton, the rest part can never execute any event with priority lower than n. Most prominently, the trapping effect no longer exists when (L2) does not hold. Consider the following example.

Example 3.2.1. *Let G, G' and H be three automata as given in Figure 25. In particular, $\{\text{I}, \text{II}\}$ is a 2-live-lock in G. When G and H are synchronised, the only transition in H, which is labelled by $\tau_{(3)}$, can never be executed. On the other hand, $\{\text{I}', \text{II}'\}$ does* not *form any live-lock in G' due to the invalidation of (L2). By reaching III', the trapping effect is released which allows H to proceed.*

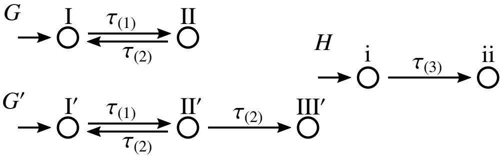

Figure 25: The trapping effect of a 2-live-lock

Note that for a Υ-shaped automaton, if both state sets X and Y are live-locks, then either $X = Y$ or $X \cap Y = \emptyset$. This implies that computing live-locks can be easily accomplished by seeking maximal silent SCCs, since one state can never be shared by two distinct live-locks.

With the notion of live-locks, we now discuss the construction of *quotient automata*, which is a well-known approach that reduces the state space of a given automaton by merging states according to proper partition of the state set. Given a set Q and an equivalence relation $\sim \subseteq Q \times Q$ on Q, we utilise $[x] := \{x' \in Q \mid (x, x') \in \sim\}$ to denote the *equivalence class* which includes the state $x \in Q$ w.r.t. $\sim$ and give the definition of quotient automaton as follows.

Definition 3.2.4. *Given a Υ-shaped automaton $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ and an equivalence relation $\sim \subseteq Q \times Q$, the* quotient automaton $G/\sim$ *of G w.r.t. $\sim$ is defined by $G/\sim\ := \langle Q/\sim, A, \underset{\sim}{\rightarrow}, \tilde{Q}^\circ, M\rangle$ where*

$$Q/\sim := \{\, [x] \mid x \in Q \,\}; \tag{105}$$

$$\tilde{Q}^\circ := \{\, [x^\circ] \mid x^\circ \in Q^\circ \,\}; \tag{106}$$

$$\begin{aligned} \underset{\sim}{\rightarrow} := \{\, [x] \overset{\alpha}{\rightarrow} [y] \mid x \overset{\alpha}{\rightarrow} y \,\} \\ - \{\, [x] \overset{\tau}{\rightarrow} [x] \mid \tau \in \Upsilon \textit{ and for any } X \subseteq [x], \\ X \textit{ is not a } \tau\textit{-live-lock in } G \,\}. \end{aligned} \tag{107}$$

Example 3.2.2. *Consider the automaton G given in Figure 26. The state set $\{\mathrm{I}, \mathrm{II}\}$ is a 2-live-lock and merging it results in a $\tau_{(2)}$-self-loop. On the other hand, neither* III *nor* IV *is in any live-lock. Merging them does not produce any silent self-loop according to* (107).

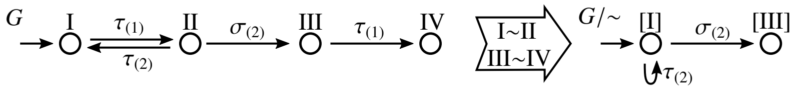

Figure 26: Quotient automaton

Comparing with the conventional quotient automaton construction (Flordal and Malik, 2009), (107) additionally requires that any silent self-loop in the quotient automaton which does not correspond to a live-lock in the original automaton should be removed. This construction attempts to preserve the trapping power after abstraction, which is crucial especially when an equivalence class includes acyclic silent event sequences. Obviously, (107) also remove some silent self-loops which were existent before abstraction, e.g. consider some automaton with only two states $x \neq y$ and two transitions $x \overset{\tau_1}{\longrightarrow} x$ and $x \overset{\tau_1}{\longrightarrow} y$. Constructing its quotient automaton w.r.t. the trivial partition removes the transition $[x] \overset{\tau_1}{\longrightarrow} [x]$. This constitutes a simple abstraction rule as well which will be discussed later in Lemma 3.2.10. We now show some useful properties of our quotient automaton construction.

Lemma 3.2.5. *Given a Υ-shaped automaton $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ and an equivalence relation $\sim \subseteq Q \times Q$, it holds that* *(i)* *For any transition* $[x] \overset{\alpha}{\rightarrow}_{\sim} [y]$ *in* $G/{\sim}$*, there exist* $x' \in [x]$ *and* $y' \in [y]$ *so that* $x' \overset{\alpha}{\rightarrow} y'$ *in* G*;*

(ii) *If* $G/{\sim}^{<n}_{\mathsf{slnt}}([x]) = \emptyset$ *in* $G/{\sim}$ *for some* $x \in Q$ *and* $n \in \mathbb{N}$*, then there exists* $x' \in [x]$ *so that* $G^{<n}_{\mathsf{slnt}}(x') = \emptyset$*.*

Proof. (i) Note the special case of $[x] \overset{\tau}{\rightarrow}_{\sim} [x]$ for some $\tau \in \Upsilon$. According to the definition of τ-live-lock, since $[x]$ includes all states of a τ-live-lock, there must exist $x', x'' \in [x]$ so that $x' \overset{\tau}{\rightarrow} x''$.

(ii) Since $G/{\sim}^{<n}_{\mathsf{slnt}}([x]) = \emptyset$, it follows that for any $x' \in [x]$ and $\tau \in \Upsilon^{<n}$, $x' \overset{\tau}{\rightarrow} \bar{x}$ implies $\bar{x} \in [x]$. Thus, if $G^{<n}_{\mathsf{slnt}}(x') \neq \emptyset$ holds for each state $x' \in [x]$, there must exist some m-live-lock $X \subseteq [x]$ so that $m < n$. This contradicts $G/{\sim}^{<n}_{\mathsf{slnt}}([x]) = \emptyset$ through the construction of quotient automaton. □

Following Definition 3.1.9, various statements and their proofs in the remainder of this chapter involve an automaton G to be abstracted and an arbitrary test T. In such cases, we take the following conventions for brevity:

- States in G are always indicated with a subscript $(\cdot)_G$, e.g. $x_G, x'_G, y_G, \dots$, while states in T are always indicated with a subscript $(\cdot)_T$.
- Subscripts $(\cdot)_G$ and $(\cdot)_T$ are omitted for transitions in G and T since they can be read from the states of the transition, e.g. $x_G \overset{\alpha}{\rightarrow} y_G$ must be a transition in G.
- Subscripts of transitions in $G \parallel T$, $\mathcal{S}(G \parallel T)$, $G/{\sim} \parallel T$ and $\mathcal{S}(G/{\sim} \parallel T)$ are omitted as well. A state in $G \parallel T$ or $\mathcal{S}(G \parallel T)$ must take the form $((\cdot)_G, (\cdot)_T)$, while a state in $G/{\sim} \parallel T$ or $\mathcal{S}(G/{\sim} \parallel T)$ must takes the form $([(\cdot)_G], (\cdot)_T)$.
- Since T is arbitrary and may carry private marking information, we aggressively assume that none of the transitions in T is silent (without losing generality, this assumption is sometimes dropped in examples since most examples utilise T to witness some undesired behaviour). In addition, the notation of $\Sigma_{T\setminus G} := \Sigma_T - \Sigma_G$ denotes the private event set of T where Σ_G and Σ_T are the alphabets of G and T, respectively. Notations

$$T_{\mathsf{prvt}}(x_T) := \{\, \tau \in \Sigma_{T\setminus G} \mid x_T \overset{\tau}{\rightarrow} \,\};$$
$$T^{<n}_{\mathsf{prvt}}(x_T) := \{\, \tau \in T_{\mathsf{prvt}}(x_T) \mid \mathsf{prio}(\tau) < n \,\}$$

are utilised to denote active private events (with priority higher than n) in state x_T, respectively. Note that, unlike what has been implicitly assumed so far, $\tau, \tau', \cdots$ will now range over $\Upsilon \cup \Sigma_{T\setminus G}$, but the natural projection will still only remove events in Υ. Furthermore, a trace is considered *asynchronous* if all event labels within this trace are from $\Upsilon \cup \Sigma_{T\setminus G}$.

3.2.1 Prioritised weak bisimulation

Based on the conventional process algebra CCS (Milner, 1989), a new process algebra CCS$^{\mathrm{ch}}$ which models concurrent systems with global event priority was introduced in (Lüttgen, 1998). In fact, the semantics of a shaped automaton in our framework is synonymous to the operational semantics of CCS$^{\mathrm{ch}}$. By extending the well-known *weak bisimulation* (a.k.a. *observational equivalence* in some contexts) from CCS, (Lüttgen, 1998) defined the *prioritised weak bisimulation (PWB)* as a CCS$^{\mathrm{ch}}$ reasoning framework. For brevity, the abbreviation *PW-bisimilar* is also utilised to refer to as *prioritised weak bisimilar*. Following the convention in (Lüttgen, 1998), several new types of transitions are defined.

Definition 3.2.6. *Given a Υ-shaped automaton $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$, define the following extended transition relations:*

(T1) $\xrightarrow[\Delta:n]{} \subseteq Q \times A \times Q$: $x \xrightarrow[\Delta:n]{\alpha} y$ *if* $x \xrightarrow{\alpha} y$ *and* $G^{<n}_{\mathrm{rglr}}(x) \subseteq \Delta$;

(T2) $\xRightarrow[\Delta:n]{} \subseteq Q \times \{\epsilon\} \times Q$: $x \xRightarrow[\Delta:n]{\epsilon} y$ *if* $x \xrightarrow[\Delta:n]{\tau_1}\xrightarrow[\Delta:n]{\tau_2} \cdots \xrightarrow[\Delta:n]{\tau_k} y$, $k \geq 0$ *and* $\tau_1 \cdots \tau_k \in (\Upsilon^{\leq n})^*$;

(T3) $\underset{n}{\Rightarrow} \subseteq Q \times \{\epsilon\} \times Q$: $x \overset{\epsilon}{\underset{n}{\Rightarrow}} y$ *if* $x \xrightarrow{\tau_1}\xrightarrow{\tau_2} \cdots \xrightarrow{\tau_k} y$, $k \geq 0$ *and* $\tau_1 \cdots \tau_k \in (\Upsilon^{\leq n})^*$.

In the following, notations $\xrightarrow[\Delta:\alpha]{}$, $\xRightarrow[\Delta:\alpha]{}$ and $\underset{\alpha}{\Rightarrow}$ for $\alpha \in A$ are utilised to refer to as $\xrightarrow[\Delta:\mathsf{prio}(\alpha)]{}$, $\xRightarrow[\Delta:\mathsf{prio}(\alpha)]{}$ and $\xRightarrow[\mathsf{prio}(\alpha)]{}$, respectively. Transition relations (T1) and (T2) are generally more difficult to be preempted – when being synchronised with another automaton, we wish that preemption caused by shared high-priority events shall not take place before the target state is reached. Thus, in (T1) and (T2), the set of active regular high-priority events is restricted in respective states. Also note that $x \xRightarrow[\Delta:n]{\epsilon} y$ implies $x \overset{\epsilon}{\underset{n}{\Rightarrow}} y$ for any $\Delta \subseteq \mathfrak{E}$. Furthermore, although (T1) generally can not be extended to string-valued labels, we still stipulate that $x \xrightarrow[\Delta:n]{\epsilon} x$, $x \xRightarrow[\Delta:n]{\epsilon} x$ and $x \overset{\epsilon}{\underset{n}{\Rightarrow}} x$ hold for any state

x, any event set Δ and any priority value n. It is worth mentioning that in these cases, there is in fact no restriction on the active event set in x. We are now in the position to define PWB over automata as follows.

Definition 3.2.7. *Let $G_1 = \langle Q_1, \Sigma, \rightarrow_1, Q_1^\circ, M\rangle$ and $G_2 = \langle Q_2, \Sigma, \rightarrow_2, Q_2^\circ, M\rangle$ be two Υ-shaped automata. A relation $\approxeq \subseteq Q_1 \times Q_2$ is a* PWB between G_1 and G_2 *if for any $x_1 \in Q_1$ and $x_2 \in Q_2$ so that $x_1 \approxeq x_2$, all the following statements hold:*

(P1) If $G^{<n}_{1,\mathrm{slnt}}(x_1) = \emptyset$ for some $n \in \mathbb{N}$, then there exists $y_2 \in Q_2$ so that $x_1 \approxeq y_2$, $G^{<n}_{2,\mathrm{slnt}}(y_2) = \emptyset$, $G^{<n}_{2,\mathrm{rglr}}(y_2) \subseteq \Delta$ and $x_2 \overset{\epsilon}{\underset{\Delta:n}{\Longrightarrow}}_2 y_2$ where $\Delta = G^{<n}_{1,\mathrm{rglr}}(x_1)$;

(P2) For any $\alpha \in A$ and $y_1 \in Q_1$ so that $x_1 \overset{\alpha}{\rightarrow}_1 y_1$, there exists $y_2 \in Q_2$ so that $y_1 \approxeq y_2$ and $x_2 \overset{\epsilon}{\underset{\Delta:\alpha}{\Longrightarrow}}_2 \overset{\mathsf{p}(\alpha)}{\underset{\Delta:\alpha}{\longrightarrow}}_2 \overset{\epsilon}{\underset{1}{\Rightarrow}}_2 y_2$ where $\Delta = G^{<\alpha}_{1,\mathrm{rglr}}(x_1)$;

(P3) If $G^{<n}_{2,\mathrm{slnt}}(x_2) = \emptyset$ for some $n \in \mathbb{N}$, then there exists $y_1 \in Q_1$ so that $x_2 \approxeq y_1$, $G^{<n}_{1,\mathrm{slnt}}(y_1) = \emptyset$, $G^{<n}_{1,\mathrm{rglr}}(y_1) \subseteq \Delta$ and $x_1 \overset{\epsilon}{\underset{\Delta:n}{\Longrightarrow}}_1 y_1$ where $\Delta = G^{<n}_{2,\mathrm{rglr}}(x_2)$;

(P4) For any $\alpha \in A$ and $y_2 \in Q_2$ so that $x_2 \overset{\alpha}{\rightarrow}_2 y_2$, there exists $y_1 \in Q_1$ so that $y_1 \approxeq y_2$ and $x_1 \overset{\epsilon}{\underset{\Delta:\alpha}{\Longrightarrow}}_1 \overset{\mathsf{p}(\alpha)}{\underset{\Delta:\alpha}{\longrightarrow}}_1 \overset{\epsilon}{\underset{1}{\Rightarrow}}_1 y_1$ where $\Delta = G^{<\alpha}_{2,\mathrm{rglr}}(x_2)$.

Two automata G_1 and G_2 are PW-bisimilar, denoted $G_1 \approxeq G_2$, if there exists a PWB between G_1 and G_2 so that for each $x_1^\circ \in Q_1^\circ$, there exists $x_2 \in Q_2$ so that $G_2 \overset{\epsilon}{\underset{1}{\Rightarrow}}_2 x_2$ and $x_1^\circ \approxeq x_2$ and vice versa.

It has been shown in (Lüttgen, 1998) that PWB is a congruence w.r.t. composition "|" and restriction "$/L$" in CCS$^{\mathrm{ch}}$. Thus, following the observation in (Malik, Streader et al., 2004), it is not surprising that two PW-bisimilar automata are conflict equivalent. In the following, we provide a brief proof to show that two PW-bisimilar automata are also conflict equivalent from an automata perspective.[2] In the following, the notation $\approxeq$ is concisely utilised to denote a PWB between two automata.

Proposition 3.2.8. *Let $G_1 = \langle Q_1, \Sigma_G, \rightarrow_1, Q_1^\circ, M_G\rangle$ and $G_2 = \langle Q_2, \Sigma_G, \rightarrow_2, Q_2^\circ, M_G\rangle$ be two Υ-shaped automaton so that $G_1 \approxeq G_2$. For any automaton*

[2] Generally, combining the CCS$^{\mathrm{ch}}$ composition combinator and restriction combinator results in a binary operation which is synonymous to shaping the synchronous composition of two automata in our framework. This was also mentioned in the original CCS (Milner, 1989), where the composition of automata was referred to as *conjunction*.

$T = \langle Q_T, \Sigma_T, \rightarrow_T, Q_T^\circ, M_T\rangle$, any transition $(x_1, x_T) \overset{\alpha}{\rightarrow}{}^{\mathcal{S}} (y_1, y_T)$ in $\mathcal{S}(G_1 \parallel T)$ and any $x_2 \in Q_2$ so that $x_1 \approxeq x_2$, there exists $y_2 \in Q_2$ so that $(x_2, x_T) \xRightarrow{\mathsf{p}(\alpha)}{}^{\mathcal{S}} (y_2, y_T)$ in $\mathcal{S}(G_2 \parallel T)$ and $y_1 \approxeq y_2$.

Proof. There are two cases:

(Case 1) Let $(x_1, x_T) \overset{\alpha}{\rightarrow}{}^{\mathcal{S}} (y_1, y_T)$ be driven by G_1 in $\mathcal{S}(G_1 \parallel T)$, i.e. $x_1 \overset{\alpha}{\rightarrow}_1 y_1$. By (P2), for all x_2 so that $x_1 \approxeq x_2$, we have $x_2 \xRightarrow[\Delta:\alpha]{\epsilon}_2 \bar{x}_2 \xrightarrow[\Delta:\alpha]{\mathsf{p}(\alpha)}_2 \bar{y}_2 \overset{\epsilon}{\underset{1}{\Rightarrow}}_2 y_2$ with $\Delta = G_{1,\text{rglr}}^{<\alpha}(x_1)$ which drives the transition

$$(x_2, x_T) \overset{\epsilon}{\Rightarrow} (\bar{x}_2, x_T) \xrightarrow{\mathsf{p}(\alpha)} (\bar{y}_2, y_T) \overset{\epsilon}{\Rightarrow} (y_2, y_T) \tag{108}$$

in $G_1 \parallel T$. We shall show that at least one trace in (108) will not be influenced by shaping. This holds trivially for the last fragment $(\bar{y}_2, y_T) \overset{\epsilon}{\Rightarrow} (y_2, y_T)$ by replacing it with $(\bar{y}_2, y_T) \overset{\epsilon}{\underset{1}{\Rightarrow}} (y_2, y_T)$. For the rest part, note that $(x_1, x_T) \overset{\alpha}{\rightarrow}{}^{\mathcal{S}} (y_1, y_T)$ in $\mathcal{S}(G_1 \parallel T)$ implies that $T(x_T) \cap (\Sigma_{T\setminus G}^{<\alpha} \cup G_{1,\text{rglr}}^{<\alpha}(x_1)) = \emptyset$. Thus from $x_2 \xRightarrow[\Delta:\alpha]{\epsilon}_2 \bar{x}_2 \xrightarrow[\Delta:\alpha]{\mathsf{p}(\alpha)} \bar{y}_2$, we must have $(x_2, x_T) \overset{\epsilon}{\Rightarrow}{}^{\mathcal{S}} (\bar{x}_2, x_T) \xrightarrow{\mathsf{p}(\alpha)}{}^{\mathcal{S}} (\bar{y}_2, y_T)$ in $\mathcal{S}(G_2 \parallel T)$.

(Case 2) Let $(x_1, x_T) \overset{\alpha}{\rightarrow}{}^{\mathcal{S}} (y_1, y_T)$ be not driven by G_1. This implies that $G_{1,\text{slnt}}^{<\alpha}(x_1) = \emptyset$. Let $\Delta = G_{1,\text{rglr}}^{<\alpha}(x_1)$ and from (P1), for all $x_2 \in Q_2$ so that $x_1 \approxeq x_2$, there exists $y_2 \in Q_2$ so that $G_{2,\text{slnt}}^{<\alpha}(y_2) = \emptyset$, $G_{2,\text{rglr}}^{<\alpha}(y_2) \subseteq \Delta$ and $x_2 \xRightarrow[\Delta:n]{\epsilon}_2 y_2$. Note that

$$(x_2, x_T) \overset{\epsilon}{\Rightarrow} (y_2, x_T) \overset{\alpha}{\rightarrow} (y_2, y_T) \tag{109}$$

in $G_2 \parallel T$. We clearly can guarantee that $(x_2, x_T) \overset{\epsilon}{\Rightarrow}{}^{\mathcal{S}} (y_2, x_T)$ in $\mathcal{S}(G_2 \parallel T)$ from the proof of Case 1. In addition, from $G_{2,\text{slnt}}^{<\alpha}(y_2) = \emptyset$ and $G_{2,\text{rglr}}^{<\alpha}(y_2) \subseteq \Delta$, we can also conclude that $(y_2, x_T) \overset{\alpha}{\rightarrow}{}^{\mathcal{S}} (y_2, y_T)$ in $\mathcal{S}(G_2 \parallel T)$. □

Theorem 3.2.9. *Let $G_1 = \langle Q_1, \Sigma_G, \rightarrow_1, Q_1^\circ, M_G\rangle$ and $G_2 = \langle Q_2, \Sigma_G, \rightarrow_2, Q_2^\circ, M_G\rangle$ be two Υ-shaped automata so that $G_1 \approxeq G_2$. It holds that $G_1 \simeq^{\mathcal{S}} G_2$.*

Proof. Let $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q_T^\circ, M_T\rangle$ be any automaton. Suppose $\mathcal{S}(G_1 \parallel T)$ is non-blocking, we shall prove that $\mathcal{S}(G_2 \parallel T)$ is non-blocking as well (The proof of the symmetric case is identical). Pick any $y_2 \in Q_2$ so that $(x_2^\circ, x_T^\circ) \overset{s}{\Rightarrow}^{\mathcal{S}} (y_2, y_T)$ in $\mathcal{S}(G_2 \parallel T)$ for some $s \in (\Sigma_G \cup \Sigma_T)^*$, $x_2^\circ \in Q_2^\circ$, $x_T^\circ \in Q_T^\circ$ and $y_T \in Q_T$. Since $G_1 \approxeq G_2$, there must exist some $x_1 \in Q_1$ so that $G_1 \underset{1}{\overset{\epsilon}{\Rightarrow}}_1 x_1$ and $x_1 \approxeq x_2^\circ$, directly implying $\mathcal{S}(G_1 \parallel T) \overset{\epsilon}{\Rightarrow}^{\mathcal{S}} (x_1, x_T^\circ)$. Furthermore, from Proposition 3.2.8, it follows from induction on concatenated transitions of any trace in $(x_2^\circ, x_T^\circ) \overset{s}{\Rightarrow}^{\mathcal{S}} (y_2, y_T)$ that there exists $y_1 \in Q_1$ so that $y_1 \approxeq y_2$ and $(x_1, x_T^\circ) \overset{s}{\Rightarrow}^{\mathcal{S}} (y_1, y_T)$ in $\mathcal{S}(G_1 \parallel T)$, i.e. $\mathcal{S}(G_1 \parallel T) \overset{s}{\Rightarrow}^{\mathcal{S}} (y_1, y_T)$. Moreover, since $\mathcal{S}(G_1 \parallel T)$ is non-blocking, for each $\Omega \in M_G \cup M_T$, there exists $\omega \in \Omega$ so that $(y_1, y_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$ in $\mathcal{S}(G_1 \parallel T)$ for some $t \in (\Sigma_G \cup \Sigma_T)^*$. Again from Proposition 3.2.8, we can conclude through induction that $(y_2, y_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$ in $\mathcal{S}(G_2 \parallel T)$, which closes the proof. □

In order to perform abstraction, given a Υ-shaped automaton, we are interested in how to construct a PW-bisimilar automaton. A first relative simple observation is that removing any silent self-loop which does *not* form a complete live-lock yields a PW-bisimilar automaton. Such silent self-loops are considered redundant and are also implicitly removed by the quotient automaton construction.

Lemma 3.2.10. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be such a Υ-shaped automaton that there exist $x, y \in Q$ and $\tau \in \Upsilon$ so that $x \neq y$, $x \xrightarrow{\tau} x$ and $x \xrightarrow{\tau} y$. Let $G' = \langle Q, \Sigma, \rightarrow - \{(x, \tau, x)\}, Q^\circ, M\rangle$. It holds that $G \approxeq G'$.*

Proof. The proof follows directly from (P1) and (P3). Note that G' is Υ-shaped as well. □

A more advanced way to construct a PW-bisimilar automaton is to construct quotient automaton w.r.t. a slightly modified version of PWB.

Definition 3.2.11. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be a Υ-shaped automaton. A* symmetric *relation $\approx \subseteq Q \times Q$ is a* PWB on *G if for any $x, x' \in Q$ so that $x \approx x'$, the following two statements hold:*

(P1') If $G_{\mathrm{slnt}}^{<n}(x) = \emptyset$ for some $n \in \mathbb{N}$, then there exists y' so that $x \approx y'$, $G_{\mathrm{slnt}}^{<n}(y') = \emptyset$, $G_{\mathrm{rglr}}^{<n}(y') \subseteq \Delta$ and $x' \underset{\Delta:n}{\overset{\epsilon}{\Longrightarrow}} y'$ where $\Delta = G_{\mathrm{rglr}}^{<n}(x)$;

(P2') For any $y \in Q$ and $\alpha \in A$ so that $x \xrightarrow{\alpha} y$, there exists $y' \in Q$ so that $y \approx y'$ and $x' \underset{\Delta:\alpha}{\overset{\epsilon}{\Longrightarrow}} \underset{\Delta:\alpha}{\overset{\mathsf{p}(\alpha)}{\longrightarrow}} \underset{1}{\overset{\epsilon}{\Rightarrow}} y'$ where $\Delta = G_{\mathrm{rglr}}^{<\alpha}(x)$.

Similar to (P3) and (P4), the symmetric part of Definition 3.2.11 can be supplemented directly and it is obvious that a PWB on an automaton G is an equivalence relation. At this stage, we intend to prove that a Υ-shaped automaton and its PWB-quotient automaton are indeed PW-similar. To this end, we first prove some useful properties, including showing that the Υ-shapedness is preserved in the PWB-quotient automaton.

Lemma 3.2.12. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be a Υ-shaped automaton with a PWB $\approx \subseteq Q \times Q$ on G.*

(i) For any $x \in Q$ and $n \in \mathbb{N}$, if $G^{<n}_{\mathrm{slnt}}(x) = \emptyset$, then $G/{\approx}^{<n}_{\mathrm{slnt}}([x]) = \emptyset$;

(ii) $G/{\approx}$ is Υ-shaped;

(iii) For any $x \in Q$ and $n \in \mathbb{N}$, if $G^{<n}_{\mathrm{slnt}}(x) = \emptyset$, then $G/{\approx}^{<n}_{\mathrm{rglr}}([x]) = G^{<n}_{\mathrm{rglr}}(x)$.

Proof. We prove all statements by contradiction:

(i) Suppose there exist $x \in Q$ and $n > 1$ so that $G^{<n}_{\mathrm{slnt}}(x) = \emptyset$ but there also exists $\tau \in G/{\approx}^{<n}_{\mathrm{slnt}}([x])$. There are two possibilities:

(Case 1) Suppose that there exists $y \in Q - [x]$ so that $[x] \xrightarrow{\tau}_{\approx} [y]$. In this case, from Lemma 3.2.5.(i), there must exist some $x' \in [x]$ and some $y' \in [y]$ so that $x' \xrightarrow{\tau} y'$. From (P2') and $x \approx x'$, there must exists some $y'' \in [y]$ so that $x \underset{\tau}{\overset{\epsilon}{\Rightarrow}} y''$, which contradicts $G^{<n}_{\mathrm{slnt}}(x) = \emptyset$ since $x \neq y''$ must hold but $\mathsf{prio}(\tau) < n$;

(Case 2) Since Case 1 does not hold, $[x] \xrightarrow{\tau}_{\approx} [x]$ must hold, implying that there is some τ-live-lock $X \subseteq [x]$ in G. From the definition of live-lock, for any $x'' \in X$, there does *not* exist any $y \in Q$ so that $x'' \overset{\epsilon}{\Rightarrow} y$ and $G^{<n}_{\mathrm{slnt}}(y) = \emptyset$ (recall that $\mathsf{prio}(\tau) < n$). This is not allowed by (P1') since $x'' \approx x$ shall hold.

(ii) Suppose $G/{\approx}$ is not Υ-shaped, i.e. there exist $x \in Q$ and $\alpha \in G/{\approx}([x])$ so that $G/{\approx}^{<\alpha}_{\mathrm{slnt}}([x]) \neq \emptyset$. This implies that there must exist $x' \in [x]$ so that $\alpha \in G(x')$ from Lemma 3.2.5.(i). Since G is Υ-shaped, $G^{<\alpha}_{\mathrm{slnt}}(x') = \emptyset$ must hold, which contradicts $G/{\approx}^{<\alpha}_{\mathrm{slnt}}([x]) \neq \emptyset$ from statement (i).

(iii) It suffices to prove the "$\subseteq$" part of the current statement as the "$\supseteq$" part holds trivially. Suppose there exists $\sigma \in G/{\approx}^{<n}_{\mathrm{rglr}}([x]) - G^{<n}_{\mathrm{rglr}}(x)$ for some $x \in Q$ and $n \in \mathbb{N}$. Then there must exist some $x' \in [x]$ so that $\sigma \in G_{\mathrm{rglr}}(x')$. Since $x \approx x'$, from (P2'), $x \underset{\sigma}{\overset{\epsilon}{\Rightarrow}} \underset{\sigma}{\overset{\sigma}{\rightarrow}} \underset{1}{\overset{\epsilon}{\Rightarrow}}$ must hold. This contradicts $G^{<n}_{\mathrm{slnt}}(x) = \emptyset$ since $\mathsf{prio}(\sigma) < n$ but $\sigma \notin G^{<n}_{\mathrm{rglr}}(x)$. □

Proposition 3.2.13. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be a Υ-shaped automaton with a PWB $\approx \subseteq Q \times Q$ on G. It holds that $G \approxeq (G/{\approx})$.*

Proof. We shall first attempt to prove that the relation $\mathsf{R} := \{(x, [x]) \mid x \in Q\}$ is a PWB between G and $G/{\approx}$. Note that the equivalence class $[\cdot]$ is defined by $\approx$. We show that R satisfies all (P1)–(P4) in Definition 3.2.7:

(P1) Pick any $x \in Q$, $n \in \mathbb{N}$ so that $G^{<n}_{\text{slnt}}(x) = \emptyset$. Trivially, we have $[x] \xrightarrow{\epsilon}_{\approx} [x]$ in $G/{\approx}$. From Lemma 3.2.12.(i), $G/{\approx}^{<n}_{\text{slnt}}([x]) = \emptyset$ holds. Furthermore, $G/{\approx}^{<n}_{\text{rglr}}([x]) \subseteq G^{<n}_{\text{rglr}}(x)$ holds as well from Lemma 3.2.12.(iii).

(P2) Pick any transition $x \xrightarrow{\alpha} y$ in G. If $\alpha \in \Upsilon$ and $x \not\approx y$, or $\alpha \in \Sigma$, it holds that $[x] \xrightarrow{\alpha}_{\approx} [y]$ in $G/{\approx}$. Otherwise, i.e. $\alpha \in \Upsilon$ and $x \approx y$, we have trivially $[x] \xrightarrow{\epsilon}_{\approx} [y]$. It remains to show for both cases that $G/{\approx}^{<\alpha}_{\text{rglr}}([x]) \subseteq G^{<\alpha}_{\text{rglr}}(x)$ hold. This is true from Lemma 3.2.12.(iii) since $x \xrightarrow{\alpha} y$ in G (which is Υ-shaped) implies $G^{<\alpha}_{\text{slnt}}(x) = \emptyset$.

(P3) Pick any $x \in Q$, $n \in \mathbb{N}$ so that $G/{\approx}^{<n}_{\text{slnt}}([x]) = \emptyset$. From Lemma 3.2.5.(ii), there exists $x' \in [x]$ so that $G^{<n}_{\text{slnt}}(x') = \emptyset$. Let $\Delta' = G^{<n}_{\text{rglr}}(x')$. From (P1'), since $x \approx x'$, there exists $y \in Q$ so that $x' \approx y$, $x \overset{\epsilon}{\underset{\Delta':n}{\Longrightarrow}} y$, $G^{<n}_{\text{slnt}}(y) = \emptyset$ and $G^{<n}_{\text{rglr}}(y) \subseteq \Delta'$. Note that $y \,\mathsf{R}\, [y]$ and $[y] = [x]$. Finally, let $\Delta = G/{\approx}^{<n}_{\text{rglr}}([x])$. Since $\Delta' \subseteq \Delta$, it follows directly that $x \overset{\epsilon}{\underset{\Delta:n}{\Longrightarrow}} y$ and $G^{<n}_{\text{rglr}}(y) \subseteq \Delta$.

(P4) Pick any transition $[x] \xrightarrow{\alpha}_{\approx} [y]$ in $G/{\approx}$. This implies that there exists $x' \in [x]$ and $y' \in [y]$ so that $x' \xrightarrow{\alpha} y'$ in G from Lemma 3.2.5.(i). Let $\Delta' = G^{<\alpha}_{\text{rglr}}(x')$. From (P2'), there must exist $y'' \in [y]$ so that $x \overset{\epsilon}{\underset{\Delta':\alpha}{\Longrightarrow}} \xrightarrow[\Delta':\alpha]{\mathsf{p}(\alpha)} \overset{\epsilon}{\underset{1}{\Rightarrow}} y''$. By further letting $\Delta = G/{\approx}^{<\alpha}_{\text{rglr}}([x])$, it can be directly concluded that $x \overset{\epsilon}{\underset{\Delta:\alpha}{\Longrightarrow}} \xrightarrow[\Delta:\alpha]{\mathsf{p}(\alpha)} \overset{\epsilon}{\underset{1}{\Rightarrow}} y''$ since $\Delta' \subseteq \Delta$.

The remaining step of the proof is to show that $G \approxeq (G/{\approx})$ by relating their initial states:

(1) Pick any $x^\circ \in Q^\circ$. $G/{\approx} \overset{\epsilon}{\underset{1}{\Rightarrow}} [x^\circ]$ directly holds since $[x^\circ] \in \tilde{Q}^\circ$.

(2) Pick any $[x] \in \tilde{Q}^\circ$. There must exist some $x^\circ \in [x] \cap Q^\circ$ and $G \overset{\epsilon}{\underset{1}{\Rightarrow}} x^\circ$ holds. □

It is worth mentioning that (Lüttgen, 1998) also introduced another identical equivalence, the *alternative prioritised weak bisimulation (APWB)*, to simplify the computation of PWB. The definition of APWB is generally based on expressing (P1') and (P2') using a single transition relation, which is given in the following.

Definition 3.2.14. *Given a Υ-shaped automaton $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$, define the extended transition relation $\Rrightarrow \subseteq Q \times A \times Q$ as such that $x \overset{\alpha}{\Rrightarrow} y$ if either of the following holds:*

(i) $\mathsf{prio}(\alpha) = 1$ *and* $x \underset{1}{\overset{\epsilon}{\Rightarrow}} \xrightarrow{\mathsf{p}(\alpha)} \underset{1}{\overset{\epsilon}{\Rightarrow}} y$*, or*

(ii) $\mathsf{prio}(\alpha) > 1$ *and there exists* $z \in Q$ *so that* $G^{<\alpha}_{\text{slnt}}(z) = \emptyset$ *and* $x \underset{n}{\overset{\epsilon}{\Rightarrow}} z \underset{\Delta:\alpha}{\overset{\epsilon}{\Longrightarrow}} \underset{\Delta:\alpha}{\xrightarrow{\mathsf{p}(\alpha)}} \underset{1}{\overset{\epsilon}{\Rightarrow}} y$ *where* $\Delta = G^{<\alpha}_{\text{rglr}}(z)$ *and* $n = \mathsf{prio}(\alpha) - 1$.

We shall note that, unlike $\Rightarrow$, a $\Rrightarrow$-transition can be labelled by a silent event. In addition, it is worth mentioning that (P1') has been implicitly encoded in the requirement (ii) of Definition 3.2.14, where "$G^{<\alpha}_{\text{slnt}}(z) = \emptyset$" is stipulated. This implies that $x \overset{\tau_{(n)}}{\Rrightarrow} x$ generally does *not* hold for all x and n. For example, we envisage that $\{x\}$ is a 2-live-lock in a Υ-shaped automaton G, i.e. there exists exactly one outgoing silent transition from x, which is $x \xrightarrow{\tau_{(2)}} x$. In this case, $x \overset{\tau_{(n)}}{\Rrightarrow} x$ holds only for $n = 1$ or $n = 2$. With this notion, APWB is defined as follows.

Definition 3.2.15. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$ be a Υ-shaped automaton. A symmetric relation $\approx^* \subseteq Q \times Q$ is an* APWB on G *if for any $x, x' \in Q$ so that $x \approx^* x'$, it holds that:*

(AP1) For any $y \in Q$ and $\alpha \in A$ so that $x \overset{\alpha}{\Rrightarrow} y$, there exists y' so that $x' \overset{\alpha}{\Rrightarrow} y'$ and $y \approx^* y'$.

Note that we have hidden the statement symmetric to (AP1) in Definition 3.2.15. It has been shown in (Lüttgen, 1998, Theorem 2.4.22) that PWB and APWB are indeed identical.

Proposition 3.2.16. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$ be a Υ-shaped automaton. A relation $\sim \subseteq Q \times Q$ on G is a PWB if and only if it is an APWB.*

Based on Proposition 3.2.16, the computation of PWB can be equivalently considered as the computation of APWB. As long as the transition relation

$\Rrightarrow$ is available, APWB can be computed through any ordinary bisimulation partition algorithms (Blom and Orzan, 2003; Fernandez, 1989; Paige and Tarjan, 1987).

Complexity of the partition of APWB If $\Rrightarrow$ is available, the complexity of computing APWB is $\mathcal{O}(\,|\Rrightarrow|\cdot\log|Q|\,)$ based on the algorithm introduced in (Fernandez, 1989). The relation $\Rrightarrow$ can be computed based on transition saturation (Milner, 1989) where we additionally need to compare the active regular event sets of the source state and target state of each transition before saturation. Note that $\Rrightarrow$ generally has infinite transitions since for any state x without outgoing silent transitions, $x \xRightarrow{\tau_{(n)}} x$ holds for all $n \in \mathbb{N}$. Nevertheless, for state partition, we only need to consider silent events $\Upsilon^{\leq N+1}$ where N is the lowest priority value appearing in this automaton. The reason is that, from Definition 3.2.14, $A \xRightarrow{\tau_{(N+1)}} B$ if and only if $A \xRightarrow{\tau_{(M)}} B$ where $M \geq N+1$. However, computing $\Rrightarrow$ is not a trivial task. In our current implementation, we first compute a transition relation

$$\Rrightarrow' := \{(x,n,y)\,|\,n = 1 \,\vee\, (n > 1, x \xRightarrow[n-1]{\epsilon} y \text{ and } G^{<n}_{\mathsf{slnt}}(y) = \emptyset)\} \tag{110}$$

which has the worst case complexity of $\mathcal{O}(\,|Q|^2\cdot N\,)$. This is the first half of the transition relation $\Rrightarrow$. Each transition in $\Rrightarrow'$ is then a seed for transition saturation, i.e. extended by $\xRightarrow[\Delta:\alpha]{\epsilon}\xrightarrow[\Delta:\alpha]{\mathsf{p}(\alpha)}\overset{\epsilon}{\underset{1}{\Rrightarrow}}$. This means that each transition in $\Rrightarrow'$ will be again maximally operated $|A|\cdot|Q|$ times where $|A| = |\Sigma| + N$. In each such operation, we need to compare the set of active high-priority regular events. This comparison has the complexity of $\mathcal{O}(\,|\Sigma|\,)$. Thus, our implementation of computing $\Rrightarrow$ is $\mathcal{O}(\,|Q|^3\cdot N\cdot|A|\cdot|\Sigma|\,)$. This complexity dominates the complexity of partitioning APWB, which is $\mathcal{O}(\,|\Rrightarrow|\cdot\log|Q|\,) = \mathcal{O}(\,|Q|^2\cdot|A|\cdot\log|Q|\,)$. $\square$

From Definition 3.2.11, we note that PWB is defined as such that if a regular event σ is to execute at some state, then an equivalent state must be able to execute σ either directly or after a delay of several silent steps with priority *not lower than* σ. The reason of this restriction can be seen from the following example. For brevity of examples in the remainder, we take the convention that, if not explicitly specified, the marking set of any automaton is $\{\{\omega\}\}$ with $\mathsf{prio}(\omega) = 1$.

Example 3.2.3. *Consider the automaton G given in Figure 27. It follows from (P1') directly that* I $\not\approx$ II. *If* I *and* II *are merged through some equivalence relation $\sim$ which generates $G/{\sim}$, a counterexample T can be constructed as* *given in Figure 27 to witness that $G \not\simeq^{\mathcal{S}} (G/{\sim})$, since $\mathcal{S}(G \parallel T)$ is blocking while $\mathcal{S}(G/{\sim} \parallel T)$ is not.*

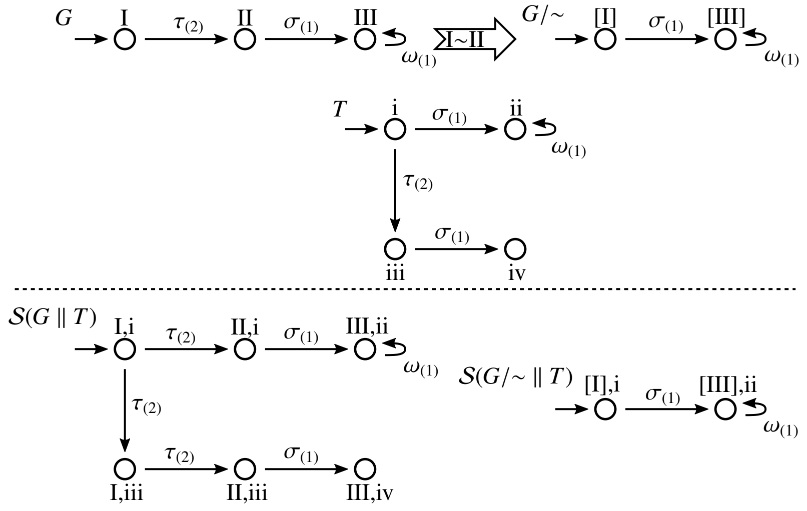

Figure 27: Silent step with priority lower than its delayed regular event may not be mergable

Consider the automaton G given in Figure 27 again. The failure of the abstraction is in fact caused by the *reachable* state (I, i) in $\mathcal{S}(G \parallel T)$, since $\tau_{(2)}$ in i will not be preempted by the shared event σ, whose priority is higher than $\tau_{(2)}$. However, this preemption indeed will happen in $([\mathrm{I}], \mathrm{i})$ in $\mathcal{S}(G/{\sim} \parallel T)$ due to the state merging. In this regard, our idea to ensure conflict equivalence is to add further restriction on the automaton so that such "bad" states will always be unreachable. As for G in Figure 27, consider adding a new state IV with a new transition $\mathrm{IV} \xrightarrow{\tau_{(3)}} \mathrm{I}$. Furthermore, let IV be the only new initial state. For such an automaton G' as given in Figure 28, merging I and II does yield a conflict-preserving abstraction. The intuition behind this modification is that, in order to visit II under synchronisation, IV must be visited at first. However, when $(\mathrm{IV}, x_T) \xrightarrow{\tau_{(3)}}{}^{\mathcal{S}} (\mathrm{I}, x_T)$ is executed for some x_T, the next step must be $(\mathrm{I}, x_T) \xrightarrow{\tau_{(2)}}{}^{\mathcal{S}} (\mathrm{II}, x_T)$ since I cannot execute any synchronised event and x_T cannot execute any private event with priority higher than 3 either. This observation motivates the definition of *redundant silent step* and it is shown in the following that merging a redundant silent step, which is referred to as the *redundant silent step rule*, is a conflict-preserving abstraction.

Definition 3.2.17. *Let $G = \langle Q, \Sigma, \rightarrow, Q^{\circ}, M \rangle$ be a Υ-shaped automaton. A transition $x \xrightarrow{\tau} x'$ with $x, x' \in Q$ and $\tau \in \Upsilon$ is a* redundant silent step *if this is*

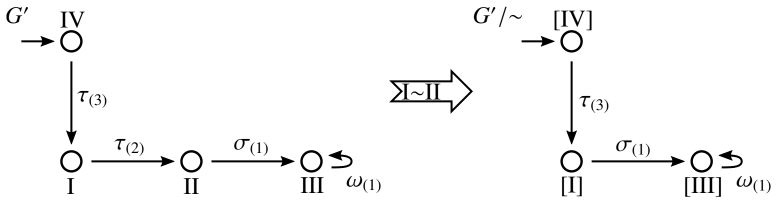

Figure 28: Redundant silent step rule

the only transition outgoing from x, $x \notin Q^\circ$ and $y \xrightarrow{\alpha} x$ for any $y \in Q$ implies $\alpha \in \Upsilon$ and $\mathsf{prio}(\alpha) > \mathsf{prio}(\tau)$. *An equivalence $\sim \subseteq Q \times Q$ on G is* induced by the transition $x \xrightarrow{\alpha} x'$ *if $x \sim x'$ and for all $y \in Q - \{x, x'\}$, $[y]$ is a singleton class.*

From Definition 3.2.17, we note that a silent self-loop can never be a redundant silent step. In addition, the definition of redundant silent step does not specifically handle the existence of live-locks. The reason is that the active event set of the target state of a redundant silent step can be completely preserved in the quotient automaton. This is stated by the following lemma.

Lemma 3.2.18. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be a Υ-shaped automaton and the equivalence $\sim \subseteq Q \times Q$ is induced by the redundant silent step $x \xrightarrow{\tau} x'$. It holds that $G(x') = G/{\sim}([x])$*

Proof. It suffices to consider the case that $[x] \xrightarrow{\tau'}_{\sim} [x]$ in $G/{\sim}$ for some $\tau' \in \Upsilon$. In this case, $[x]$ contains a τ'-live-lock from G which is formed either by $\{x, x'\}$ or solely by $\{x'\}$ (solely by $\{x\}$ is clearly impossible). The case of solely by $\{x'\}$ is rather trivial, while when $\{x, x'\}$ is a τ'-live-lock, we must have $x' \xrightarrow{\tau'} x$ since from the definition of redundant silent step, $\mathsf{prio}(\tau') > \mathsf{prio}(\tau)$ must hold. $\square$

Consider a redundant silent step $x_G \xrightarrow{\tau} x'_G$ in a Υ-shaped automaton G with some regular event σ so that $\mathsf{prio}(\sigma) < \mathsf{prio}(\tau)$, $\sigma \notin G(x_G)$ and $\sigma \in G(x'_G)$, we can assert that x_G and x'_G are never PW-bisimilar. Intuitively, this invalidates the property given in Proposition 3.2.8 if it is assumed that the resulting quotient automaton and the original one are "equivalent". More precisely, for some state x_T in a test automaton T, if $x_T \xrightarrow{\tau'}$ for some $\tau' \in \Sigma_{T\setminus G}$ where $\mathsf{prio}(\tau') \leq \mathsf{prio}(\tau)$, we must have $(x_G, x_T) \xrightarrow{\tau'}{}^{\mathcal{S}}$ in $\mathcal{S}(G \parallel T)$, while $([x_G], x_T) \xrightarrow{\tau'}{}^{\mathcal{S}}$ may *not* hold in $\mathcal{S}(G/{\sim} \parallel T)$ when $\sigma \in T(x_T)$ and $\mathsf{prio}(\sigma) < \mathsf{prio}(\tau')$. Interestingly, such (x_G, x_T) is never reachable in $\mathcal{S}(G \parallel T)$.

Proposition 3.2.19. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q_G^\circ, M_G\rangle$ be a Υ-shaped automaton and the equivalence $\sim \subseteq Q_G \times Q_G$ is induced by the redundant silent step $\bar{x}_G \xrightarrow{\tau} \bar{x}'_G$. Let $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q_T^\circ, M_T\rangle$ be any automaton. For all $\bar{x}_T \in Q_T$ so that $T^{\leq\tau}_{\mathsf{prvt}}(\bar{x}_T) \neq \emptyset$, $(\bar{x}_G, \bar{x}_T)$ is not reachable in $\mathcal{S}(G \parallel T)$.*

Proof. We prove by contradiction. Pick any $\bar{x}_T \in Q_T$ so that $T^{\leq\tau}_{\mathsf{prvt}}(\bar{x}_T) \neq \emptyset$. To reach $(\bar{x}_G, \bar{x}_T)$, one shall first reach some (y_G, y_T) where $y_G \in Q_G$, $y_T \in Q_T$ so that $y_G \xrightarrow{\tau'} \bar{x}_G$ with some $\tau' \in \Upsilon$. From Definition 3.2.17, it is clear that $\mathsf{prio}(\tau') \geq n$. This implies that $(y_G, \bar{x}_T) \overset{\tau'}{\not\rightarrow}^{\mathcal{S}} (\bar{x}_G, \bar{x}_T)$. With this observation, we continue the proof by attempting to construct a trace from (y_G, y_T) to $(\bar{x}_G, \bar{x}_T)$, which must fail. Consider the following cases:

(Case 1) $T^{\leq\tau}_{\mathsf{prvt}}(y_T) \neq \emptyset$. Let $y_T \xrightarrow{\tau''} \bar{y}_T$ for some $\bar{y}_T \in Q_T$ and $\tau'' \in T^{\leq\tau}_{\mathsf{prvt}}(y_T)$. Clearly, $\mathsf{prio}(\tau'') < \mathsf{prio}(\tau')$, and we concatenate $(y_G, y_T) \xrightarrow{\tau''}^{\mathcal{S}} (y_G, \bar{y}_T)$ (without losing generality, we can assume that $T^{<\tau''}_{\mathsf{prvt}}(y_T) = \emptyset$). If $T^{\leq\tau}_{\mathsf{prvt}}(\bar{y}_T) \neq \emptyset$ always holds for such concatenation, then the construction is trapped in Case 1 and $\bar{x}_G$ can never be visited. Otherwise, let $T^{\leq\tau}_{\mathsf{prvt}}(\bar{y}_T) = \emptyset$, which leads to Case 2.

(Case 2) $T^{\leq\tau}_{\mathsf{prvt}}(y_T) = \emptyset$. From (y_G, y_T), since only private events can be executed, consider the possibility of concatenating $(y_G, y_T) \xrightarrow{\tau'}^{\mathcal{S}} (\bar{x}_G, y_T)$ in $\mathcal{S}(G \parallel T)$, since executing a private transition in T indeed rolls the construction back to the beginning of either Case 1 or 2. However, if $(y_G, y_T) \xrightarrow{\tau'}^{\mathcal{S}} (\bar{x}_G, y_T)$, it implies that the next transition which can be concatenated must be $(\bar{x}_G, y_T) \xrightarrow{\tau}^{\mathcal{S}} (\bar{x}'_G, y_T)$ since $\mathsf{prio}(\tau) < \mathsf{prio}(\tau')$ and executing any shared event with priority higher than τ in $(\bar{x}_G, y_T)$ is not possible. Recall that $y_T \neq \bar{x}_T$ due to $T^{\leq\tau}_{\mathsf{prvt}}(\bar{x}_T) \neq \emptyset$, i.e. for any $z_T \in Q_T$ so that $(\bar{x}_G, z_T)$ is reachable in $\mathcal{S}(G \parallel T)$, $T^{\leq\tau}_{\mathsf{prvt}}(\bar{z}_T) = \emptyset$ must hold. This indeed closes the proof. □

When merging a redundant silent step, states characterised in Proposition 3.2.19 are exactly the "bad" states which potentially invalidate conflict equivalence. With this observation, the following proposition is derived which is similar to Proposition 3.2.8.

Proposition 3.2.20. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q_G^\circ, M_G\rangle$ be a Υ-shaped automaton and the equivalence $\sim \subseteq Q_G \times Q_G$ is induced by the redundant silent step $\bar{x}_G \xrightarrow{\tau} \bar{x}'_G$. Let $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q_T^\circ, M_T\rangle$ be any automaton.*

(i) For any transition $([x_G], x_T) \xrightarrow{\alpha}^{\mathcal{S}} ([y_G], y_T)$ *in* $\mathcal{S}(G/\!\sim \,\|\, T)$*, at least one of the following two statements is true for any* $x'_G \in [x_G]$*:*

a) There exists some $y'_G \in [y_G]$ *so that* $(x'_G, x_T) \overset{\mathsf{p}(\alpha)}{\Longrightarrow}^{\mathcal{S}} (y'_G, y_T)$ *in* $\mathcal{S}(G \,\|\, T)$*, or*

b) (x'_G, x_T) *is not reachable in* $\mathcal{S}(G \,\|\, T)$*.*

(ii) For any transition $(x_G, x_T) \xrightarrow{\alpha}^{\mathcal{S}} (y_G, y_T)$ *in* $\mathcal{S}(G \,\|\, T)$*, at least one of the following two statements is true:*

a) $([x_G], x_T) \xrightarrow{\mathsf{p}(\alpha)}^{\mathcal{S}} ([y_G], y_T)$ *in* $\mathcal{S}(G/\!\sim \,\|\, T)$*, or*

b) (x_G, x_T) *is not reachable in* $\mathcal{S}(G \,\|\, T)$*.*

Proof. (i) If $[x_G]$ is a singleton, then statement a) holds trivially. Thus, we let $[x_G] = [\bar{x}_G]$. In this case, note that if $([x_G], x_T) \xrightarrow{\alpha}^{\mathcal{S}} ([y_G], y_T)$ is not driven by G, then statement a) must be true as well since either $(\bar{x}_G, x_T) \xrightarrow{\alpha}^{\mathcal{S}} (\bar{x}_G, y_T)$ or $(\bar{x}_G, x_T) \xrightarrow{\tau}^{\mathcal{S}} (\bar{x}'_G, x_T) \xrightarrow{\alpha}^{\mathcal{S}} (\bar{x}'_G, y_T)$ holds in $\mathcal{S}(G \,\|\, T)$ from Lemma 3.2.18. Thus, let $([x_G], x_T) \xrightarrow{\alpha}^{\mathcal{S}} ([y_G], y_T)$ be driven by G. This implies $\alpha \in G(\bar{x}'_G)$ due to Lemma 3.2.18 and we pick $x'_G \in [x_G]$. There are two cases:

(Case 1) $x'_G = \bar{x}'_G$. We shall note that $G(\bar{x}'_G) = G/\!\sim([\bar{x}'_G])$ from Lemma 3.2.18. Thus, in this case, statement a) must hold.

(Case 2) $x'_G = \bar{x}_G$. We directly suppose that statement a) is not true, i.e. $(\bar{x}_G, x_T) \overset{\mathsf{p}(\alpha)}{\not\Longrightarrow}^{\mathcal{S}} (y'_G, y_T)$ in $\mathcal{S}(G \,\|\, T)$ for any $y'_G \in [y_G]$. This implies that $T^{<\tau}_{\text{prvt}}(x_T) \neq \emptyset$, since otherwise, we must be able to execute $(\bar{x}_G, x_T) \xrightarrow{\tau}^{\mathcal{S}} (\bar{x}'_G, x_T)$, which leads to Case 1. Note that $T^{<\tau}_{\text{prvt}}(x_T) \neq \emptyset$ implies $T^{\leq\tau}_{\text{prvt}}(x_T) \neq \emptyset$. Thus, in this case, statement b) must hold from Proposition 3.2.19.

(ii) Note that statement a) must hold if $[x_G]$ is a singleton. In addition, statement a) holds for $x_G = \bar{x}'_G$ as well from Lemma 3.2.18. Let $x_G = \bar{x}_G$. If $(x_G, x_T) \xrightarrow{\alpha}^{\mathcal{S}} (y_G, y_T)$ is driven by G, then $y_G = \bar{x}'_G$ and statement a) holds from a trivial transition $([x_G], x_T) \xrightarrow{\epsilon} ([y_G], x_T)$. Let $(x_G, x_T) \xrightarrow{\alpha}^{\mathcal{S}} (y_G, y_T)$ be not driven by G. In this case, statement b) must hold from Proposition 3.2.19 since $\mathsf{prio}(\alpha) \leq \mathsf{prio}(\tau)$, i.e. $\alpha \in T^{\leq\tau}_{\text{prvt}}(x_T)$. □

In Proposition 3.2.20, both statements (i).a) and (ii).a) are synonymous to Proposition 3.2.8. In fact, replacing the equivalence relation in Proposition 3.2.20 by PWB (on G) results in a true proposition as well where both a) statements are always true. We are now in the position to state the redundant silent step rule as follows.

Theorem 3.2.21 (redundant silent step rule). *Let $G = \langle Q, \Sigma, \rightarrow, Q^{\circ}, M \rangle$ be a Υ-shaped automaton and the equivalence $\sim \subseteq Q \times Q$ is induced by some redundant silent step. It holds that $G \simeq^{\mathcal{S}} (G/{\sim})$.*

Proof. The proof is indeed the same as the proof of Theorem 3.2.9 up to uniform substitution of the equivalence relation. Note that for all states reached by the induction, statements (i).a) and (ii).a) of Proposition 3.2.20 must hold. □

Complexity of the redundant silent step rule The redundant silent step rule can be applied by checking whether all incoming transitions of a state are silent (which is of order $|Q|$ in a Υ-shaped automaton since from one state, there is maximally one silent transition to a given target state) and redirecting these incoming transitions (which is again of order $|Q|$, since all transitions are silent). Since this procedure should be repeated for each state, the overall complexity is $\mathcal{O}(\ |Q|^3\ )$. □

3.2.2 Abstraction rules based on incoming equivalence

In the ordinary context without prioritised events, (Flordal and Malik, 2009) introduced several abstraction rules based on *incoming equivalence*. The current section attempts to adapt these rules for prioritised events, which is in general not as trivial as one might imagine.

The motivation of introducing incoming equivalence is to pre-partition states that can be reached in the same way; namely, when a state can be reached under synchronisation with some test, an incoming equivalent state must be reachable under the synchronisation with the same test as well. Incoming equivalence does not necessarily imply (ordinary) conflict equivalence, but serves as a filter to enable two conflict-preserving abstraction rules, i.e. the *active events rule* and the *silent continuation rule*. The key property of incoming equivalence in the ordinary context is, all states in the same class can be reached from the same state with a regular event, possibly with some silent events before or after the regular event. Since this property is rather cumbersome to achieve when considering prioritised events, a formal definition of this property is first given and named as *redirectability*.

Definition 3.2.22. *Let $G = \langle Q_G, \Sigma_G, \to_G, Q_G^\circ, M_G\rangle$ be a Υ-shaped automaton. An equivalence $\sim \subseteq Q_G \times Q_G$ is* redirectable *if and only if for any automaton $T = \langle Q_T, \Sigma_T, \to_T, Q_T^\circ, M_T\rangle$, $y_G \in Q_G$, $y_T \in Q_T$ and $s_T \in \Sigma^*_{T\setminus G}$, the following two statements hold:*

(R1) $(x_G, x_T) \xrightarrow{\sigma}^{\mathcal{S}} \xRightarrow{s_T}^{\mathcal{S}} (y_G, y_T)$ *in $\mathcal{S}(G \parallel T)$ for any $x_G \in Q_G$, $x_T \in Q_T$ and $\sigma \in \Sigma_G$ implies that for all $y'_G \in [y_G]$, $(x_G, x_T) \xRightarrow{\sigma s_T}^{\mathcal{S}} (y'_G, y_T)$ in $\mathcal{S}(G \parallel T)$;*

(R2) $\mathcal{S}(G \parallel T) \xRightarrow{s_T}^{\mathcal{S}} (y_G, y_T)$ *implies that for all $y'_G \sim y_G$, $\mathcal{S}(G \parallel T) \xRightarrow{s_T}^{\mathcal{S}} (y'_G, y_T)$.*

It is to observe from Definition 3.2.22 that, for a redirectable equivalence relation, the synchronised behaviour can choose any state in a class to proceed if at least one state in the class can be reached by a regular event followed by some private events (or the synchronised behaviour is currently in the initial state). From this observation, redirectability can commonly be utilised in such scenarios where a transition need to be redirected to a successor, in which desired future behaviour is guaranteed. This feature is especially useful when reasoning the original behaviour from the abstracted behaviour. In this regard, we review Lemma 3.2.5.(i), which is a general property for any arbitrary equivalence stating that a transition in the original behaviour can always be reconstructed from the abstracted behaviour. Note that the existence statement "*there* exists $y' \in [y]$..." in Lemma 3.2.5 does not allow concatenating multiple reconstructed transitions, i.e. we can *not* guarantee that e.g. $([x_G], x_T) \xrightarrow{\alpha}^{\mathcal{S}} ([y_G], y_T) \xrightarrow{\alpha'}^{\mathcal{S}} ([z_G], z_T)$ implies the existence of $x'_G \in [x_G]$, $y'_G \in [y_G]$ and $z'_G \in [z_G]$ so that $(x'_G, x_T) \xrightarrow{\alpha}^{\mathcal{S}} (y'_G, y_T) \xrightarrow{\alpha'}^{\mathcal{S}} (z'_G, z_T)$. Nevertheless, this problem can be solved by requiring redirectability on an equivalence if a trace begins with a regular event from G. This is stated by the following proposition which is inspired by (Flordal and Malik, 2009, Lemma 2).

Proposition 3.2.23. *Let $G = \langle Q_G, \Sigma_G, \to_G, Q_G^\circ, M_G\rangle$ be a Υ-shaped automaton with a redirectable equivalence $\sim \subseteq Q \times Q$ on G. For any automaton $T = \langle Q_T, \Sigma_T, \to_T, Q_T^\circ, M_T\rangle$, the following two statements hold:*

(i) For any trace

$$([x_{G0}], x_{T0}) \xrightarrow{\alpha_1}^{\mathcal{S}} ([x_{G1}], x_{T1}) \xrightarrow{\alpha_2}^{\mathcal{S}} \cdots \xrightarrow{\alpha_k}^{\mathcal{S}} ([x_{Gk}], x_{Tk}) \qquad (111)$$

in $\mathcal{S}(G/\!\sim\, \parallel T)$ *where* $k \geq 1$, $\alpha_1 \in \Sigma_G$ *and* $\alpha_i \in A_G \cup A_T$ *for all* $i \in \{2, \cdots, k\}$, *there exist* $x'_{G0} \in [x_{G0}]$ *and* $x'_{Gk} \in [x_{Gk}]$ *so that* $(x'_{G0}, x_{T0}) \xRightarrow{\mathsf{p}(\alpha_1 \cdots \alpha_k)}{}^{\mathcal{S}} (x'_{Gk}, x_{Tk})$ *in* $\mathcal{S}(G \parallel T)$*;*

(ii) If $\mathcal{S}(G/\!\sim\, \parallel T) \overset{s}{\Rightarrow}{}^{\mathcal{S}} ([x_G], x_T)$ *for some* $s \in (\Sigma_G \cup \Sigma_T)^*$*, then there exists* $x'_G \in [x_G]$ *so that* $\mathcal{S}(G \parallel T) \overset{s}{\Rightarrow}{}^{\mathcal{S}} (x'_G, x_T)$*.*

Proof. (i) We prove by induction:
(*Base case*) For $k = 1$, it holds immediately that there exists $x'_{G0} \in [x_{G0}]$ and $x'_{G1} \in [x_{G1}]$ so that $(x'_{G0}, x_{T0}) \xrightarrow{\alpha_1}{}^{\mathcal{S}} (x'_{G1}, x_{T1})$ in $\mathcal{S}(G \parallel T)$ from Lemma 3.2.5.(i) since $\alpha_1 \in \Sigma_G$.

(*Inductive step*) Suppose the proposition holds for some $k \geq 1$, i.e. for some trace

$$([x_{G0}], x_{T0}) \xrightarrow{\alpha_1}{}^{\mathcal{S}} ([x_{G1}], x_{T1}) \xrightarrow{\alpha_2}{}^{\mathcal{S}} \cdots \xrightarrow{\alpha_k}{}^{\mathcal{S}} ([x_{Gk}], x_{Tk}) \tag{112}$$

in $\mathcal{S}(G/\!\sim\, \parallel T)$ where $\alpha_1 \in \Sigma_G$ and $\alpha_i \in A_G \cup A_T$ for all $i \in \{2, \dots, k\}$, there exist $x'_{G0} \in [x_{G0}]$ and $x'_{Gk} \in [x_{Gk}]$ so that

$$(x'_{G0}, x_{T0}) \xRightarrow{\mathsf{p}(\alpha_1 \cdots \alpha_k)}{}^{\mathcal{S}} (x'_{Gk}, x_{Tk}) \tag{113}$$

in $\mathcal{S}(G \parallel T)$. From this hypothesis, we show that the proposition holds for $k+1$ as well. Consider any successive transition

$$([x_{Gk}], x_{Tk}) \xrightarrow{\alpha_{k+1}}{}^{\mathcal{S}} ([x_{Gk+1}], x_{Tk+1}) \tag{114}$$

of trace (112). This indeed implies the existence of $x''_{Gk} \in [x_{Gk}]$ and $x'_{Gk+1} \in [x_{Gk+1}]$ so that $(x''_{Gk}, x_{Tk}) \xrightarrow{\alpha_{k+1}}{}^{\mathcal{S}} (x'_{Gk+1}, x_{Tk+1})$ in $\mathcal{S}(G \parallel T)$ due to either Lemma 3.2.5.(i) (if (114) is driven by G) or Lemma 3.2.5.(ii) (if (114) is not driven by G). Now if $[x_{Gk}]$ is a singleton, the proof closes directly since $x'_{Gk} = x''_{Gk}$. Otherwise, from trace (112), we shall find the last regular transition driven by G, i.e. we consider the trace fragment

$$([x_{Gi-1}], x_{Ti-1}) \xrightarrow{\alpha_i}{}^{\mathcal{S}} ([x_{Gi}], x_{Ti}) \xrightarrow{\alpha_{i+1} \cdots \alpha_k}{}^{\mathcal{S}} ([x_{Gk}], x_{Tk}) \tag{115}$$

from (112) where $\alpha_i \in \Sigma_G$ and $\alpha_{i+1} \cdots \alpha_k \in (\Sigma_{T \setminus G} \cup \Upsilon)^*$. Let $s_T = \mathsf{p}(\alpha_{i+1} \cdots \alpha_k)$. From this and due to the inductive hypothesis, we can extract the fragment

$$(\bar{x}_G, \bar{x}_T) \xrightarrow{\alpha_i} \xRightarrow{s_T}{}^{\mathcal{S}} (x'_{Gk}, x_{Tk}) \tag{116}$$

from (113) for some $\bar{x}_G \in Q_G$ and $\bar{x}_T \in Q_T$. Since $\sim$ is redirectable, we have

$$(\bar{x}_G, \bar{x}_T) \xRightarrow{\alpha_i s_T}{}^{\mathcal{S}} (x''_{Gk}, x_{Tk}) \tag{117}$$

from (R1), which can be concatenated by $(x''_{Gk}, x_{Tk}) \xrightarrow{\alpha_{k+1}}{}^{\mathcal{S}} (x'_{Gk+1}, x_{Tk+1})$.

(ii) We separate the proof into two cases:

(Case 1) $s \in \Sigma^*_{T\setminus G}$. This case holds directly from (R2). Note that we have proven an even more general version of the current statement, i.e. the statement holds for all states in $[x_G]$ instead of the existence of some state in $[x_G]$, which will be utilised in the proof for the next case.

(Case 2) $s \notin \Sigma^*_{T\setminus G}$. Then let

$$\mathcal{S}(G/{\sim} \parallel T) \xRightarrow{s_T}{}^{\mathcal{S}} ([y_G], y_T) \xrightarrow{\sigma}{}^{\mathcal{S}} ([z_G], z_T) \xRightarrow{t}{}^{\mathcal{S}} ([x_G], x_T) \tag{118}$$

where $s_T \in \Sigma^*_{T\setminus G}$, $\sigma \in \Sigma_G$ and $t \in (\Sigma_G \cup \Sigma_T)^*$ so that $s_T\sigma t = s$. From Case 1, for all $y'_G \in [y_G]$, $\mathcal{S}(G \parallel T) \xRightarrow{s_T}{}^{\mathcal{S}} (y'_G, y_T)$. From statement (i), there exists $y''_G \in [y_G]$ and $x'_G \in [x_G]$ so that $(y''_G, y_T) \xRightarrow{\sigma t}{}^{\mathcal{S}} (x'_G, x_T)$, which closes the proof. □

In order to achieve redirectability, we are going to define incoming equivalence for prioritised events by adapting the ordinary version introduced in (Flordal and Malik, 2009, Definition 7). From the notion of PWB, intuitively, the transition relation $\xRightarrow[\Delta:\alpha]{\epsilon}\xrightarrow[\Delta:\alpha]{\mathsf{p}(\alpha)}\overset{\epsilon}{\underset{1}{\Rightarrow}}$ is tolerant against preemption and can possibly be utilised for the definition of incoming equivalence w.r.t. prioritised events. In particular, the execution of $\overset{\epsilon}{\underset{1}{\Rightarrow}}$ cannot be disturbed by any rest part due to preemption. In fact, this requirement can be relaxed when considering redirectability. Consider some new transition relations as follows.

Definition 3.2.24. *Given a Υ-shaped automaton $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$, define the following extended transition relations:*

(T4) $\underset{!}{\rightarrow} \subseteq Q \times \Upsilon \times Q$: $x \overset{\tau}{\underset{!}{\rightarrow}} y$ *if* $x \xrightarrow{\tau} y$ *and* $G^{<\tau}_{\text{rglr}}(x) = \emptyset$.

(T5) $\underset{n}{\hookrightarrow} \subseteq Q \times \{\epsilon\} \times Q$: $x \overset{\epsilon}{\underset{n}{\hookrightarrow}} y$ *if either of the following holds:*

(i) $n = 1$ *and* $x \overset{\epsilon}{\underset{1}{\Rightarrow}} y$, *or*

(ii) $n \geq 2$, $x \xrightarrow[!]{\tau_1}\xrightarrow[!]{\tau_2} \cdots \xrightarrow[!]{\tau_k} y$, $k \geq 1$ *and* $\mathsf{lo}(\{\tau_1 \cdots \tau_k\}) = n$.

Transition relations introduced in Definition 3.2.24 are generally more restrictive than those in Definition 3.2.6 in that preemption through regular events shall never take place on a $\hookrightarrow$-transition before the last state. Note that the new transition symbol "$\hookrightarrow$" is utilised intentionally to differ from $\rightarrow$ and $\Rightarrow$ since when $n \geq 2$, $x \underset{n}{\overset{\epsilon}{\hookrightarrow}} x$ generally does not hold for an arbitrary state x, because at least one $\tau_{(n)}$ transition must exist within $\underset{n}{\overset{\epsilon}{\hookrightarrow}}$. Based on Definition 3.2.24, the adapted definition of incoming equivalence is presented as follows.

Definition 3.2.25. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$ be a Υ-shaped automaton. An equivalence $\sim_{\mathit{inc}} \subseteq Q \times Q$ on G is an* incoming equivalence *if and only if for any $x, x' \in Q$ so that $x \sim_{\mathit{inc}} x'$, all the following statements hold:*

(I1) For any $\sigma \in \Sigma$, $n \in \mathbb{N}$ and $y \in Q$, $y \underset{\Delta:\sigma}{\overset{\epsilon}{\Longrightarrow}} \underset{\Delta:\sigma}{\overset{\sigma}{\longrightarrow}} \underset{n}{\overset{\epsilon}{\hookrightarrow}} x \Leftrightarrow y \underset{\Delta:\sigma}{\overset{\epsilon}{\Longrightarrow}} \underset{\Delta:\sigma}{\overset{\sigma}{\longrightarrow}} \underset{n}{\overset{\epsilon}{\hookrightarrow}} x'$ where $\Delta = G^{<\sigma}_{\mathrm{rglr}}(y)$;

(I2) For any $n \in \mathbb{N}$, $Q^\circ \underset{n}{\overset{\epsilon}{\hookrightarrow}} x \Leftrightarrow Q^\circ \underset{n}{\overset{\epsilon}{\hookrightarrow}} x'$;

(I3) If $x \neq x'$, then for any $y \in Q$ and $\tau \in \Upsilon$, $y \overset{\tau}{\rightarrow} \overset{\epsilon}{\Rightarrow} x$ or $y \overset{\tau}{\rightarrow} \overset{\epsilon}{\Rightarrow} x'$ implies $G^{<\tau}_{\mathrm{rglr}}(y) = \emptyset$.

Clearly, incoming equivalence distributes over arbitrary union. Hence, it is legit to utilise $\sim_{\mathrm{inc}}$ to denote the coarsest incoming equivalence of an automaton. In addition, any equivalence finer as an incoming equivalence is an incoming equivalence as well. Thus, the notation of $\sim \subseteq \sim_{\mathrm{inc}}$ is often utilised to indicate that $\sim$ is an incoming equivalence. Similar to the ordinary version in (Flordal and Malik, 2009), Definition 3.2.25 attempts to equalise states which can be reached in the same way, i.e. only the past of a state is considered and its future behaviour is totally ignored. However, such intuition is inadequate when prioritised events are taken into consideration, since redirectability requires that the same state y_T from some test T should be reached before and after abstraction. If no restrictions over the future behaviour of incoming equivalent states are given, redirectability can be easily invalidated if two equivalent states have different preemptive power. In addition, we notice that when abstracting an automaton through quotient automaton construction, it is almost always required that the quotient automaton of a Υ-shaped automaton shall be Υ-shaped as well, which can *not* be guaranteed solely by incoming equivalence. To this end, we first introduce our definitions of active-event equivalence and silent-continuation equivalence.

Definition 3.2.26. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$ be a Υ-shaped automaton. An equivalence $\sim_{\mathit{ae}} \subseteq Q \times Q$ on G is an* active-event equivalence *if for any $x, x' \in Q$ so that $x \sim_{\mathit{ae}} x'$ and $x \neq x'$, the following two statements hold:*

(AE1) $G_{\mathrm{slnt}}(x) = G_{\mathrm{slnt}}(x') = \emptyset$;

(AE2) $G_{\mathrm{rglr}}(x) = G_{\mathrm{rglr}}(x')$.

Definition 3.2.27. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be a Υ-shaped automaton. An equivalence $\sim_{sc} \subseteq Q \times Q$ on G is a* silent-continuation equivalence *if for any $x, x' \in Q$ so that $x \sim_{sc} x'$ and $x \neq x'$, all the following statements hold for some $\tau \in \Upsilon$:*

(SC1) $\tau \in G(x) \cap G(x')$;

(SC2) $G^{<\tau}_{\mathrm{rglr}}(x) = G^{<\tau}_{\mathrm{rglr}}(x') = \emptyset$;

(SC3) Neither x nor x' is in any live-lock.

Similar to $\sim_{\mathrm{inc}}$, we utilise $\sim_{\mathrm{ae}}$, $\sim_{\mathrm{sc}}$ to denote the coarsest active-event equivalence and silent-continuation equivalence and write $\sim \subseteq \sim_{\mathrm{ae}}$ or $\sim \subseteq \sim_{\mathrm{sc}}$ to denote that $\sim$ is an equivalence of the corresponding type, respectively. By combining $\sim_{\mathrm{inc}}$ with either $\sim_{\mathrm{ae}}$ or $\sim_{\mathrm{sc}}$, the redirectability can be achieved.

Proposition 3.2.28. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq Q \times Q$ on G be such that either $\sim \subseteq \sim_{inc} \cap \sim_{ae}$ or $\sim \subseteq \sim_{inc} \cap \sim_{sc}$. It holds that $\sim$ is redirectable.*

Before proceeding to prove Proposition 3.2.28, note that $\sim_{\mathrm{ae}}$ imposes a relatively strong restriction on equivalent states that silent events are never active on any state in a non-singleton class. Readers familiar with (Flordal and Malik, 2009) may be curious about the possibility of relaxing Definition 3.2.26 to equate states with regular active events delayed by $\underset{1}{\overset{\epsilon}{\Rightarrow}}$, i.e., by defining $\Delta_{\mathrm{ae}}(x) := \{\sigma \in \Sigma \mid x \underset{1}{\overset{\epsilon}{\Rightarrow}}\overset{\sigma}{\rightarrow}\}$, one may expect that $x \sim x'$ when $\Delta_{\mathrm{ae}}(x) = \Delta_{\mathrm{ae}}(x')$. However, combining such a “relaxed” active-event equivalence with incoming equivalence does *not* guarantee conflict equivalence. Consider the following example:

Example 3.2.4. *Consider automata G and T given in Figure 29. Note that G is Υ-shaped and* $\mathrm{I} \sim_{inc} \mathrm{III}$ *clearly holds since state* III *can be reached from the initial state through $\tau^*_{(1)}$. Furthermore, from $\Delta_{\mathrm{ae}}(x) = \Delta_{\mathrm{ae}}(x')$, we are able to equate* I *and* III *which results in $G/\sim$. In this case, although* $([\mathrm{II}], \mathrm{ii})$ *is reachable in $\mathcal{S}(G/\sim \parallel T)$,* $(\mathrm{II}, \mathrm{ii})$ *is not reachable in $\mathcal{S}(G \parallel T)$ since* $\mathrm{i} \xrightarrow{\tau_{(2)}} \mathrm{ii}$ *cannot happen before* $\mathrm{I} \xrightarrow{\tau_{(1)}} \mathrm{III}$ *and the transition* $\mathrm{I} \xrightarrow{\sigma} \mathrm{II}$ *is labelled by a shared event σ. One observes that in this example,* $\mathrm{I} \xrightarrow{\tau_{(1)}} \mathrm{III}$ *somewhat “disables”* $\mathrm{I} \xrightarrow{\sigma} \mathrm{II}$ *although both events are with the same priority. In this case, equating*

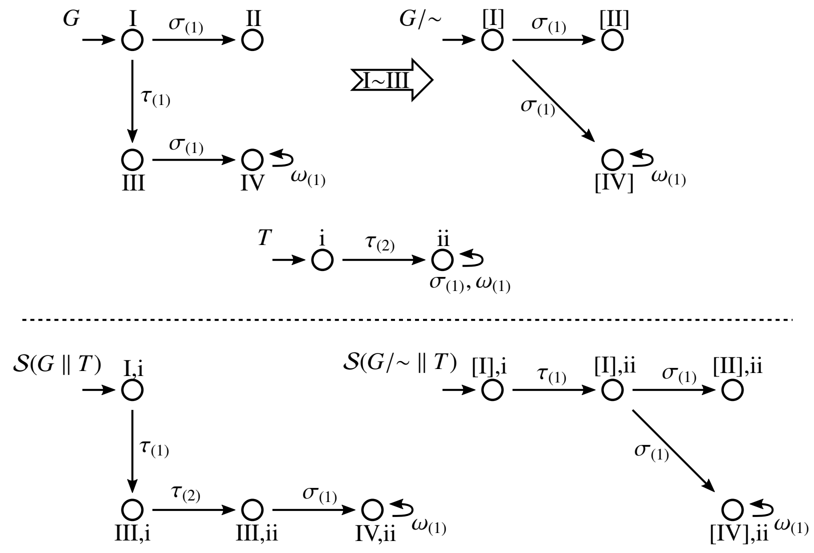

Figure 29: Counterexample of equating incoming equivalent states with the same set of delayed active events

I *and* III *is unacceptable, especially when both states have different future behaviour, e.g. one leads to a non-blocking future while another blocks. Finally, it is also worth noting that "preserving"* I $\xrightarrow{\tau_{(1)}}$ III *into a* $\tau_{(1)}$*-self-loop (which is against the quotient automaton construction) in* $G/\sim$ *does not solve the issue, since the trapping power is rendered inconsistent.*

As a counterexample, Example 3.2.4 infers that for two incoming equivalent states, additionally requiring them to have the same preemptive power is essential to achieve redirectability. Otherwise, private transitions in T may be inconsistently preempted. This can be guaranteed by $\sim_{\text{ae}}$ or $\sim_{\text{sc}}$, as being stated in the following lemma.

Lemma 3.2.29. *Let* $G = \langle Q_G, \Sigma_G, \to_G, Q_G^\circ, M_G\rangle$ *be a* Υ*-shaped automaton. Let* $\sim \subseteq Q \times Q$ *be an equivalence on G so that either* $\sim \subseteq \sim_{ae}$ *or* $\sim \subseteq \sim_{sc}$ *holds. For any automaton* $T = \langle Q_T, \Sigma_T, \to_T, Q_T^\circ, M_T\rangle$ *and any trace*

$$(x_G, x_{T0}) \xrightarrow{\tau_1}^{\mathcal{S}} (x_G, x_{T1}) \xrightarrow{\tau_2}^{\mathcal{S}} \cdots \xrightarrow{\tau_k}^{\mathcal{S}} (x_G, x_{Tk}) \tag{119}$$

in $\mathcal{S}(G \parallel T)$ *where* $k \geq 0$ *and* $\tau_i \in \Sigma_{T\setminus G}$ *for all* $i \in \{1, \dots, k\}$*, it holds that for any* $x'_G \in [x_G]$*, a trace*

$$(x'_G, x_{T0}) \xrightarrow{\tau_1}^{\mathcal{S}} (x'_G, x_{T1}) \xrightarrow{\tau_2}^{\mathcal{S}} \cdots \xrightarrow{\tau_k}^{\mathcal{S}} (x'_G, x_{Tk}) \tag{120}$$

exists in $\mathcal{S}(G \parallel T)$ as well.

Proof. The current statement is trivially true from (AE2), (SC1) and (SC2). $\square$

At the current stage, the fundamental components for achieving redirectability have indeed been collected. In fact, by strengthening Definition 3.2.25 as such that

- all $\hookrightarrow$-transitions are uniformly replaced by $\overset{\epsilon}{\underset{1}{\Rightarrow}}$ (strengthens (I1) and (I2)) and
- "implies $G^{<\tau}_{\text{rglr}}(y) = \emptyset$" in (I3) is uniformly replaced by "implies $\tau = \tau_{(1)}$" (strengthens (I3)),

redirectability would be easily achieved by the conjunction of a strengthened incoming equivalence with either $\sim_{\text{ae}}$ or $\sim_{\text{sc}}$. In particular, the strengthened version only allows $\tau_{(1)}$ to appear before reaching some state in a non-trivial equivalence class. Consider the following example.

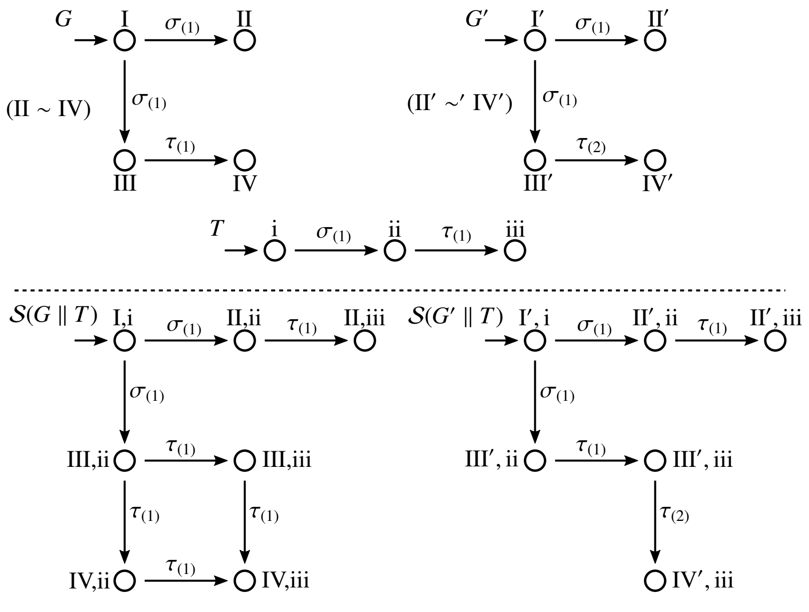

Figure 30: The conjunction of a strengthened incoming equivalence and an active-event equivalence is redirectable

Example 3.2.5. *Consider automata G and T given in Figure 30. Note that in G, states are partitioned by the equivalence $\sim$ so that $(\text{II}, \text{IV}) \in \sim \subseteq \sim_{inc} \cap \sim_{ae}$. In* *this case,* $\sim$ *is indeed redirectable, which can be "witnessed" by the automaton T. In particular, since state* $(\mathrm{II}, \mathrm{ii})$ *is reachable, the reachability of state* $(\mathrm{IV}, \mathrm{ii})$ *should be guaranteed as well to achieve redirectability since* $\mathrm{II} \sim \mathrm{IV}$*. This must hold since the only silent predecessor of* IV*, i.e.* III*, reaches* IV *via* $\tau_{(1)}$*. Thus, regardless the priority of successive transition in T, G can always execute all its* $\tau_{(1)}$*-transitions first, then T executes its private transitions. However, this is not the case if we replace the transition label of* $\mathrm{III} \xrightarrow{\tau_{(1)}} \mathrm{IV}$ *by e.g.* $\tau_{(2)}$*, which results in* G'*. The resulting equivalence relation* $\sim'$ *is no longer redirectable, since* $(\mathrm{IV}', \mathrm{ii})$ *is rendered unreachable.*

Despite the awareness that the strengthened incoming equivalence contributes to achieve redirectability, we are interested in a more relaxed definition, i.e. utilising the original Definition 3.2.25. By reviewing Example 3.2.5, the statement "*G can always execute all its* $\tau_{(1)}$*-transitions first, then T executes its private transitions*" can be relaxed by $\hookrightarrow$-transitions while still preserving redirectability. In the following, we consider the properties of $\hookrightarrow$-transitions by mainly focusing on traces under synchronisation with only private events. Such traces are referred to as *asynchronous traces*. Note that temporarily in Lemma 3.2.30 and Proposition 3.2.31, we do not require either automaton to be Υ-shaped since the discussed properties are stated for traces instead of for automata. This benefits some proofs in that two traces from their corresponding automata can be freely swapped.

Lemma 3.2.30. *Let* $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q_G^\circ, M_G\rangle$ *and* $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q_T^\circ, M_T\rangle$ *be two arbitrary automata and*

$$(x_G, x_{T0}) \xrightarrow{\tau_1}{}^{\mathcal{S}} (x_G, x_{T1}) \xrightarrow{\tau_2}{}^{\mathcal{S}} \cdots \xrightarrow{\tau_k}{}^{\mathcal{S}} (x_G, x_{Tk}) \xrightarrow{\tau_{k+1}}{}^{\mathcal{S}} (y_G, x_{Tk}) \qquad (121)$$

be an asynchronous trace in $\mathcal{S}(G \parallel T)$ *so that* $k \geq 0$ *and for all* $i \in \{1, \cdots, k\}$*,* $(x_G, x_{Ti-1}) \xrightarrow{\tau_j}{}^{\mathcal{S}} (x_G, x_{Tj})$ *is driven by T and* $(x_G, x_{Tk}) \xrightarrow{\tau_{k+1}}{}^{\mathcal{S}} (y_G, x_{Tk})$ *is driven by G. It holds that* $\mathsf{prio}(\tau_{k+1}) \geq \mathsf{lo}(\{\tau_1, \cdots, \tau_k\})$*.*

Proof. Note that for all $i \in \{1, \cdots, k\}$, $(x_G, x_{Ti}) \xrightarrow{\tau_{k+1}}$ in $G \parallel T$. Thus, the current statement must hold as the trace is in a shaped automaton $\mathcal{S}(G \parallel T)$. $\square$

The statement of Lemma 3.2.30 may seem verbose at first glance. Nevertheless, it induces an interesting property of asynchronous traces in shaped synchronous compositions: each time when the "transition-driving" automaton alternates, the priority of the silent event on the next transition cannot

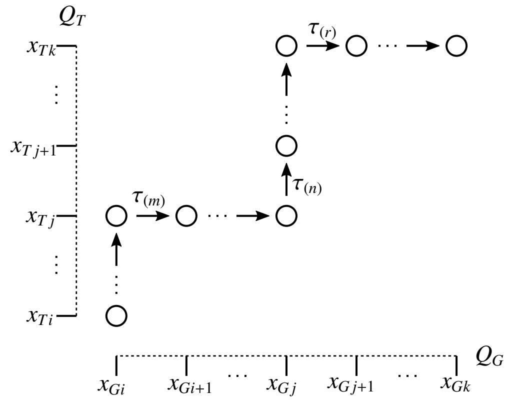

Figure 31: An asynchronous trace in shaped synchronous composition

elevate. Consider the sketch in Figure 31, where an asynchronous trace under shaped synchronous composition is given in grid. Points on the horizontal axis correspond to states in Q_G, while those on the vertical axis correspond to states in Q_T. Consider those states at which the driving automaton alternates, i.e. the "direction" of the trace changes. It is easy to conclude from Lemma 3.2.30 that $m \leq n \leq r$ must hold. More importantly, if the trace ends with a transition driven by G (this is indeed the case in Figure 31), it can be immediately concluded that the last "T-state" of the last state (x_{Tk} in Figure 31) cannot execute any private events whose priority is higher than any transition in the trace. At the same time, the lowest priority of all transitions driven by G cannot be higher than the lowest priority of any transition driven by T. These properties are formalised by the following proposition.

Proposition 3.2.31. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q_G^\circ, M_G\rangle$ and $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q_T^\circ, M_T\rangle$ be two arbitrary automata and*

$$(x_{G0}, x_{T0}) \xrightarrow{\tau_1}{}^{\mathcal{S}} (x_{G1}, x_{T1}) \xrightarrow{\tau_2}{}^{\mathcal{S}} \cdots \xrightarrow{\tau_k}{}^{\mathcal{S}} (x_{Gk}, x_{Tk}) \tag{122}$$

be an asynchronous trace in $\mathcal{S}(G \parallel T)$ where $k \geq 1$ and the last transition $(x_{Gk-1}, x_{Tk-1}) \xrightarrow{\tau_k}{}^{\mathcal{S}} (x_{Gk}, x_{Tk})$ is driven by G.

(i) Let $n = \mathsf{lo}(\{\tau_1, \cdots, \tau_k\})$. It holds that $T_{\text{prvt}}^{<n}(x_{Tk}) = \emptyset$;

(ii) If at least one transition in (122) is driven by T,then $n_G \geq n_T$ where

$$n_G = \mathsf{lo}(\{\tau_i \mid (x_{Gi-1}, x_{Ti-1}) \xrightarrow{\tau_i}{}^{\mathcal{S}} (x_{Gi}, x_{Ti}) \text{ is driven by } G\}); \tag{123}$$

$$n_T = \mathsf{lo}(\{\tau_i \mid (x_{Gi-1}, x_{Ti-1}) \xrightarrow{\tau_i}{}^{\mathcal{S}} (x_{Gi}, x_{Ti}) \text{ is driven by } T\}). \tag{124}$$

Proof. Note that both statements hold trivially if all transitions in (122) are driven by G. Thus, we assume that there exists at least one transition driven by T in (122).

(i) Let $\tau \in T_{\text{prvt}}(x_{Tk})$ and consider the trace fragment

$$(x_{Gi}, x_{Ti}) \xrightarrow{\tau_{i+1}}^{\mathcal{S}} \cdots \xrightarrow{\tau_j}^{\mathcal{S}} (x_{Gj}, x_{Tj}) \xrightarrow{\tau_{j+1}}^{\mathcal{S}} \cdots \xrightarrow{\tau_k}^{\mathcal{S}} (x_{Gk}, x_{Tk}) \tag{125}$$

where $0 \le i < j < k$ and all transitions before (x_{Gj}, x_{Tj}) are driven by T while all transitions after (x_{Gj}, x_{Tj}) are driven by G. It follows immediately that $\mathsf{prio}(\tau) \ge \mathsf{lo}\{\tau_{j+1}, \cdots, \tau_k\} \ge \mathsf{prio}(\tau_{j+1})$. Furthermore, from Lemma 3.2.30, we have $\mathsf{prio}(\tau_{j+1}) \ge \mathsf{lo}\{\tau_{i+1}, \cdots, \tau_j\} \ge \mathsf{prio}(\tau_{i+1})$. This is sufficient for an induction to reason the entire trace.

(ii) Consider the trace fragment $(x_{Gi}, x_{Ti}) \xrightarrow{\tau_{i+1}}^{\mathcal{S}} \cdots \xrightarrow{\tau_k}^{\mathcal{S}} (x_{Gk}, x_{Tk})$ where $0 < i < k$ and all transitions are driven by G but $(x_{Gi-1}, x_{Ti-1}) \xrightarrow{\tau_i}^{\mathcal{S}} (x_{Gi}, x_{Ti})$ is driven by T. The current statement is clearly true since $\mathsf{prio}(\tau_{i+1}) \ge n_T$ from statement (i) by swapping G and T, and $n_G \ge \mathsf{prio}(\tau_{i+1})$ must hold as well. □

Combining Proposition 3.2.31 and Lemma 3.2.29, we are now in the position to conclude the following property. In particular, the statement (ii) of the following proposition covers the $\overset{\epsilon}{\underset{1}{\Rightarrow}}$-transition in the strengthened incoming equivalence as a special case which was mentioned in Example 3.2.5. Note that we again require G to be Υ-shaped from now on.

Proposition 3.2.32. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q_G^\circ, M_G\rangle$ be a Υ-shaped automaton and*

$$(x_{G0}, x_{T0}) \xrightarrow{\tau_1}^{\mathcal{S}} (x_{G1}, x_{T1}) \xrightarrow{\tau_2}^{\mathcal{S}} \cdots \xrightarrow{\tau_k}^{\mathcal{S}} (x_{Gk}, x_{Tk}) \tag{126}$$

be an asynchronous trace in $\mathcal{S}(G \parallel T)$ where $k \ge 0$. Let $n = \mathsf{lo}(\{\tau_i \mid (x_{Gi-1}, x_{Ti-1}) \xrightarrow{\tau_i} (x_{Gi}, x_{Ti})$ is driven by $G\})$ and

$$x'_{G0} \xrightarrow{\tau'_1} x'_{G1} \xrightarrow{\tau'_2} \cdots \xrightarrow{\tau'_{k'}} x'_{Gk'} \tag{127}$$

with $k' \ge 0$ be a trace in G so that all events on this trace are silent, $\mathsf{lo}(\{\tau'_1, \cdots, \tau'_{k'}\}) = n$ and for all $i' \in \{1, \cdots, k'\}$, $G_{\text{rglr}}^{<\tau'_i}(x'_{Gi'-1}) = \emptyset$. The following two statements hold:

(i) *For* (126), *if* $k \geq 1$ *and the last transition* $(x_{Gk-1}, x_{Tk-1}) \xrightarrow{\tau_k}^{\mathcal{S}} (x_{Gk}, x_{Tk})$ *is driven by* G, *then* $(x'_{G0}, x_{T0}) \xRightarrow{\mathsf{p}(\tau_1 \cdots \tau_k)}^{\mathcal{S}} (x'_{Gk'}, x_{Tk})$ *in* $\mathcal{S}(G \parallel T)$ *where the last transition is driven by* G*;*

(ii) *Let* $\sim \subseteq Q_G \times Q_G$ *be an equivalence on* G *so that either* $\sim \subseteq \sim_{ae}$ *or* $\sim \subseteq \sim_{sc}$*. If* $x_{Gk} \sim x'_{Gk'}$*, then* $(x'_{G0}, x_{T0}) \xRightarrow{\mathsf{p}(\tau_1 \cdots \tau_k)}^{\mathcal{S}} (x'_{Gk'}, x_{Tk})$ *in* $\mathcal{S}(G \parallel T)$*.*

Proof. Note that the restriction $G^{<\tau'_i}_{\text{rglr}}(x'_{Gi'-1}) = \emptyset$ for $i' \in \{1, \cdots, k'\}$ excludes the possibility of preemption through regular events before reaching $x'_{Gk'}$. For convenience, let $n' = \mathsf{lo}(\{\tau'_1, \cdots, \tau'_{k'}\})$.

(i) It suffices to construct an asynchronous trace from (x'_{G0}, x_{T0}) to $(x'_{Gk'}, x_{Tk})$ which will not be influenced by shaping and the last transition is driven by G. Let $i' = j = 0$ and we start the construction from the first state $(x'_{Gi'}, x_{Tj}) = (x'_{G0}, x_{T0})$. Note that due to Case 2 of Step 2 in the following, it is not possible to reach $x'_{Gk'}$ before x_{Tk} is reached.

(Step 1) Consider two possible cases:

(Case 1) Only $j = k$ holds, i.e. x_{Tk} is reached. Consider the trace given in (126) and from Proposition 3.2.31.(i), it follows that $T^{<n}_{\text{prvt}}(x_{Tk}) = \emptyset$. Since $n = n'$ is required, we are able to directly complete the construction by concatenating the remaining transitions driven by G to reach $x'_{Gk'}$, i.e. we must have $(x'_{Gi'}, x_{Tk}) \overset{\epsilon}{\Rightarrow}^{\mathcal{S}} (x'_{Gk'}, x_{Tk})$ where all transitions are driven by G in $\mathcal{S}(G \parallel T)$, since priority of all remaining transitions driven by G cannot be lower than any $\tau \in T_{\text{prvt}}(x_{Tk})$ and preemption through shared events is impossible. This terminates the construction.

(Case 2) Neither $i' = k'$ nor $j = k$ holds. Proceed to Step 2.

(Step 2) Since preemption through shared prioritised events is not possible, we can proceed from $(x'_{Gi'}, x_{Tj})$ with either one transition driven by G or one driven by T, or both. Consider the two possible cases:

(Case 1) $\mathsf{prio}(\tau'_{Gi'+1}) \neq n'$. Then concatenate either $(x'_{Gi'}, x_{Tj}) \xrightarrow{\tau'_{i'+1}}^{\mathcal{S}} (x'_{Gi'+1}, x_{Tj})$ or $(x'_{Gi'}, x_{Tj}) \xrightarrow{\tau_{j+1}}^{\mathcal{S}} (x'_{Gi'}, x_{Tj+1})$ according to their priority and update either $i' := i' + 1$ or $j := j + 1$, respectively. Note that each time when the current case is met, we must have not reached $x'_{Gk'}$ yet since the transition with the lowest priority in (127) has not been reached yet. Go back to Step 1.

(Case 2) $\mathsf{prio}(\tau'_{Gi'+1}) = n'$. Since $n = n'$ was required, from Proposition 3.2.31.(ii), it follows that $\mathsf{prio}(\tau'_{Gi'+1}) = n \geq \mathsf{lo}(\{\tau_i \mid (x_{Gi-1}, x_{Ti-1}) \xrightarrow{\tau_i}^{\mathcal{S}} (x_{Gi}, x_{Ti})$ is driven by $T\})$. Thus, we are able to concatenate the remaining transitions driven by T to reach x_{Tk}, i.e. we have $(x'_{Gi'}, x_{Tj}) \overset{s_T}{\Longrightarrow}^{\mathcal{S}} (x'_{Gi'}, x_{Tk})$ where all transitions are driven by T in $\mathcal{S}(G \parallel T)$ and $s_T \in \Sigma^*_{T\setminus G}$ is the remaining private event sequence in T. Update $j := k$ and go to Step 1. We will be in Case 1 of Step 1.

(ii) The current statement holds trivially if all transitions in (126) are driven by G. In addition, the current statement holds directly if all transitions in (126) are driven by T from Lemma 3.2.29. Moreover, if the last transition in (126) is driven by G, the current statement holds directly as well from statement (i). The only remaining case is that (126) ends with such a trace fragment $(x_{Gi}, x_{Ti}) \xrightarrow{\tau_{i+1}}^{\mathcal{S}} \cdots \xrightarrow{\tau_k}^{\mathcal{S}} (x_{Gk}, x_{Tk})$ with $i \in \{1, \cdots, k-1\}$ where all transitions are driven by T (i.e. $x_{Gi} = x_{Gk}$) and $(x_{Gi-1}, x_{Ti-1}) \xrightarrow{\tau_i}^{\mathcal{S}} (x_{Gi}, x_{Ti})$ is driven by G. From statement (i), $(x'_{G0}, x_{T0}) \overset{\mathsf{p}(\tau_1\cdots\tau_i)}{\Longrightarrow}^{\mathcal{S}} (x'_{Gk'}, x_{Ti})$ in $\mathcal{S}(G \parallel T)$ holds. Furthermore, due to Lemma 3.2.29, we must be able to concatenate the remaining transitions driven by T to reach x_{Tk}, i.e. $(x'_{Gk'}, x_{Ti}) \overset{\mathsf{p}(\tau_{i+1}\cdots\tau_k)}{\Longrightarrow}^{\mathcal{S}} (x'_{Gk'}, x_{Tk})$. □

Proposition 3.2.32.(ii) shows us an important property between asynchronous traces when preemption through shared events is excluded: for two traces with the same lowest priority and both final states are equivalent w.r.t. either $\sim_{\mathrm{ae}}$ or $\sim_{\mathrm{sc}}$, they can be utilised to synchronise the same private-event trace. This matches the definition of $\hookrightarrow$-transition which is utilised in Definition 3.2.25. With all the preparation, we are now ready to prove that Proposition 3.2.28 is true.

Proof of Proposition 3.2.28. We prove (R1) as follows: let $(x_G, x_T) \xrightarrow{\sigma}^{\mathcal{S}} (\bar{x}_G, \bar{x}_T) \overset{s_T}{\Longrightarrow}^{\mathcal{S}} (y_G, y_T)$ in $\mathcal{S}(G \parallel T)$ for some $\bar{x}_G \in Q_G$ and $\bar{x}_T \in Q_T$. By (I3), we have $\bar{x}_G \underset{n}{\overset{\epsilon}{\hookrightarrow}} y_G$ in G with some $n \in \mathbb{N}$. From (I1), for each $y'_G \in [y_G]$, we must have some $\bar{x}'_G \in Q_G$ so that $x_G \underset{\Delta:\sigma}{\overset{\epsilon}{\Longrightarrow}}^{\mathcal{S}} \underset{\Delta:\sigma}{\overset{\sigma}{\longrightarrow}}^{\mathcal{S}} \bar{x}'_G \underset{n}{\overset{\epsilon}{\hookrightarrow}} y'_G$ where $\Delta = G^{<\sigma}_{\mathrm{rglr}}(x_G)$. Clearly, we directly have $(x_G, x_T) \overset{\sigma}{\Rightarrow}^{\mathcal{S}} (\bar{x}'_G, \bar{x}_T)$. In addition, $(\bar{x}'_G, \bar{x}_T) \overset{s_T}{\Longrightarrow}^{\mathcal{S}} (y'_G, y_T)$ can also be guaranteed from Proposition 3.2.32.(ii) or directly from Lemma 3.2.29. This indeed shows that (R1) of Definition 3.2.22 is fulfilled.

The proof for (R2) is similar by only considering $(\bar{x}_G, \bar{x}_T) \overset{s_T}{\Longrightarrow}^{\mathcal{S}} (y_G, y_T)$ and letting $(\bar{x}_G, \bar{x}_T)$ be any initial state in $\mathcal{S}(G \parallel T)$. □

With Proposition 3.2.28 being proved, the conjunction of $\sim_{\text{inc}}$ with either $\sim_{\text{ae}}$ or $\sim_{\text{sc}}$ guarantees that a trace in the original behaviour can be reconstructed from a trace after abstraction. To imply conflict-equivalence (which is an if-and-only-if statement), a similar property in the converse direction is to clarify as well.

Proposition 3.2.33. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q_G^\circ, M_G \rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq Q_G \times Q_G$ on G so that either $\sim \subseteq \sim_{ae}$ or $\sim \subseteq \sim_{sc}$ holds. For any automaton $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q_T^\circ, M_T \rangle$ and any transition $(x_G, x_T) \overset{\alpha}{\rightarrow}^{\mathcal{S}} (y_G, y_T)$ in $\mathcal{S}(G \parallel T)$, it holds that $([x_G], x_T) \xrightarrow{\mathsf{p}(\alpha)}^{\mathcal{S}} ([y_G], y_T)$ in $\mathcal{S}(G/\sim \parallel T)$.*

Proof. If $x_G \sim y_G$, $\alpha \in \Upsilon$ and $(x_G, x_T) \overset{\alpha}{\rightarrow}^{\mathcal{S}} (y_G, y_T)$ is driven by G, we will have a trivial transition $([x_G], x_T) \overset{\epsilon}{\rightarrow}^{\mathcal{S}} ([y_G], y_T) = ([x_G], x_T)$ in $\mathcal{S}(G/\sim \parallel T)$. Otherwise, $([x_G], x_T) \overset{\alpha}{\rightarrow} ([y_G], y_T)$ in $G/\sim \parallel T$. This transition will clearly not be shaped due to the definition of $\sim_{\text{ae}}$ and $\sim_{\text{sc}}$. □

We are now in the position to state two conflict-preserving abstraction rules, i.e. the active events rule and the silent continuation rule, in Theorems 3.2.35 and 3.2.36. For the active events rule, the following lemma is given to simplify the proof.

Lemma 3.2.34. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q_G^\circ, M_G \rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq \sim_{ae}$. For any automaton $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q_T^\circ, M_T \rangle$, if $([x_G], x_T) \xRightarrow{s_T\mathsf{p}(\alpha)}^{\mathcal{S}}$ in $\mathcal{S}(G/\sim \parallel T)$ for some $x_G \in Q_G$, $x_T \in Q_T$, $s_T \in \Sigma_{T\setminus G}^*$ and $\alpha \in A_G$, then for all $x'_G \in [x_G]$, $(x'_G, x_T) \xRightarrow{s_T\mathsf{p}(\alpha)}^{\mathcal{S}}$ in $\mathcal{S}(G \parallel T)$.*

Proof. Recall that for any non-singleton class $[x_G]$, $G_{\text{slnt}}(x_G) = \emptyset$ must hold. Consider two cases:

(Case 1) $\alpha \in \Upsilon$. If there is some trace in $([x_G], x_T) \overset{s_T}{\Longrightarrow}^{\mathcal{S}}$ where all transitions are not driven by G, the current statement is directly true due to Lemma 3.2.29. Otherwise, let

$$([x_G], x_T) \overset{t_T}{\Longrightarrow}^{\mathcal{S}} ([\bar{x}_G], y_T) \overset{\tau}{\rightarrow}^{\mathcal{S}} ([y_G], y_T) \overset{u_T}{\Longrightarrow}^{\mathcal{S}} \qquad (128)$$

in $\mathcal{S}(G/{\sim} \parallel T)$ for some $\tau \in \Upsilon$, $\bar{x}_G, y_G \in Q_G$, $y_T \in Q_T$, $t_T u_T = s_T$, $([\bar{x}_G], y_T) \xrightarrow{\tau}^{\mathcal{S}} ([y_G], y_T)$ is driven by G and all transitions in the fragment $([y_G], y_T) \overset{u_T}{\Longrightarrow}^{\mathcal{S}}$ are not driven by G. Note that all states on $[x_G] \overset{\epsilon}{\Rightarrow}_{\sim} [\bar{x}_G]$ in $G/{\sim}$ are singletons. Thus, there must exist $y'_G \in [y_G]$ so that $(x_G, x_T) \overset{t_T}{\Longrightarrow}^{\mathcal{S}} (\bar{x}_G, y_T) \xrightarrow{\tau}^{\mathcal{S}} (y'_G, y_T)$ in $\mathcal{S}(G \parallel T)$. In addition, $(y'_G, y_T) \overset{u_T}{\Longrightarrow}$ in $\mathcal{S}(G/{\sim} \parallel T)$ must hold due to Lemma 3.2.29.

(Case 2) $\alpha \in \Sigma_G$, i.e. $\mathsf{p}(\alpha) = \alpha$ and we have $([x_G], x_T) \overset{s_T}{\Longrightarrow}^{\mathcal{S}} \xrightarrow{\alpha}^{\mathcal{S}}$ in $\mathcal{S}(G/{\sim} \parallel T)$. Following Case 1, if there exists a trace on the fragment $([x_G], x_T) \overset{s_T}{\Longrightarrow}^{\mathcal{S}}$ where all transitions are not driven by G, then the current statement holds directly in that for all $x'_G \in [x_G]$, $\alpha \in G(x'_G)$ holds. Otherwise, consider concatenating an α transition at the end of (128), i.e.

$$([x_G], x_T) \overset{t_T}{\Longrightarrow}^{\mathcal{S}} ([\bar{x}_G], y_T) \xrightarrow{\tau}^{\mathcal{S}} ([y_G], y_T) \overset{u_T}{\Longrightarrow}^{\mathcal{S}} \xrightarrow{\alpha}^{\mathcal{S}} . \qquad (129)$$

Recall that all transitions on the fragment $([y_G], y_T) \overset{u_T}{\Longrightarrow}^{\mathcal{S}}$ are not driven by G, i.e. before executing the final $\xrightarrow{\alpha}^{\mathcal{S}}$-transition, $[y_G]$ will not execute any transition. Thus, from Lemma 3.2.29, $(x_G, x_T) \overset{t_T}{\Longrightarrow}^{\mathcal{S}} (\bar{x}_G, y_T) \xrightarrow{\tau}^{\mathcal{S}} (y'_G, y_T) \overset{u_T}{\Longrightarrow}^{\mathcal{S}} \xrightarrow{\alpha}^{\mathcal{S}}$ for some $y'_G \in [y_G]$ must hold. □

Theorem 3.2.35 (active events rule). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G \rangle$ be a Υ-shaped automaton with an equivalence ${\sim} \subseteq {\sim_{ae}} \cap {\sim_{inc}}$ on G. It holds $G \simeq_{\mathcal{S}} (G/{\sim})$.*

Proof. Let $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^\circ_T, M_T \rangle$ be any automaton:

($\Rightarrow$) Suppose $\mathcal{S}(G \parallel T)$ is non-blocking. Pick $x_G \in Q_G$, $x_T \in Q_T$ and $s \in (\Sigma_G \cup \Sigma_T)^*$ so that $\mathcal{S}(G/{\sim} \parallel T) \overset{s}{\Rightarrow}^{\mathcal{S}} ([x_G], x_T)$. By Proposition 3.2.23.(ii), there exists $x'_G \in [x_G]$ so that $\mathcal{S}(G \parallel T) \overset{s}{\Rightarrow}^{\mathcal{S}} (x'_G, x_T)$ and due to the non-blockingness of $\mathcal{S}(G \parallel T)$, for each $\Omega \in M_G \cup M_T$, there exists $\omega \in \Omega$ so that $(x'_G, x_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$ in $\mathcal{S}(G \parallel T)$ for some $t \in (\Sigma_G \cup \Sigma_T)^*$. By Proposition 3.2.33, it holds that $([x_G], x_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$.

($\Leftarrow$) Suppose $\mathcal{S}(G/{\sim} \parallel T)$ is non-blocking and pick $x_G \in Q_G$, $x_T \in Q_T$ and $s \in (\Sigma_G \cup \Sigma_T)^*$ so that $\mathcal{S}(G \parallel T) \overset{s}{\Rightarrow}^{\mathcal{S}} (x_G, x_T)$. From Proposition 3.2.33 and the non-blockingness of $\mathcal{S}(G/\!\sim \parallel T)$, for each $\Omega \in M_G \cup M_T$, there exists $\omega \in \Omega$ and $t \in (\Sigma_G \cup \Sigma_T)^*$ so that $\mathcal{S}(G/\!\sim \parallel T) \overset{s}{\Rightarrow}^{\mathcal{S}} ([x_G], x_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$. There are two cases:

(Case 1) $t \in \Sigma^*_{T\setminus G}$. This case holds directly from Lemma 3.2.34. Note that the sub-case of $\omega \in \Sigma_{T\setminus G}$ holds as well.

(Case 2) For any t, Case 1 does not hold. Then we must have $([x_G], x_T) \overset{s_T}{\Longrightarrow}\overset{\sigma}{\rightarrow}$ for some $\sigma \in \Sigma_G - \Omega$ and $s_T \in \Sigma^*_{T\setminus G}$. By applying Lemma 3.2.34, we have

$$(x_G, x_T) \overset{s_T}{\Longrightarrow}^{\mathcal{S}} (\bar{x}_G, \bar{x}_T) \overset{\sigma}{\rightarrow}^{\mathcal{S}} (y_G, y_T) \tag{130}$$

in $\mathcal{S}(G \parallel T)$ for some $\bar{x}_G, y_G \in Q_G$ and $\bar{x}_T, y_T \in Q_T$ so that $s_T\sigma \leqslant t$. From Proposition 3.2.33 and the non-blockingness of $\mathcal{S}(G/\!\sim \parallel T)$, $([y_G], y_T) \overset{t'\omega'}{\Longrightarrow}^{\mathcal{S}}$ must hold for some $t' \in (\Sigma_G \cup \Sigma_T)^*$ and $\omega' \in \Omega$. Consider the following two sub-cases (which are comparable with Case 1 and Case 2), i.e. either

(i) $t'\omega' \in \Sigma^+_{T\setminus G}$. From Lemma 3.2.34, we directly have $(y_G, y_T) \overset{t'\omega'}{\Longrightarrow}^{\mathcal{S}}$.

(ii) Case 2.(i) does not hold for any $t'\omega'$. By applying Proposition 3.2.33 and then Lemma 3.2.34 again, we have altogether

$$(x_G, x_T) \overset{s_T}{\Longrightarrow}^{\mathcal{S}} (\bar{x}_G, \bar{x}_T) \overset{\sigma}{\rightarrow}^{\mathcal{S}} \overset{t_T}{\Longrightarrow}^{\mathcal{S}} (\bar{y}_G, \bar{y}_T) \overset{\sigma'}{\longrightarrow}^{\mathcal{S}} \tag{131}$$

in $\mathcal{S}(G \parallel T)$ for some $\bar{y}_G \in Q_G$, $\bar{y}_T \in Q_T$, $t_T \in \Sigma^*_{T\setminus G}$ and $\sigma' \in \Sigma_G$. From Proposition 3.2.33, Proposition 3.2.23.(i) and the non-blockingness of $\mathcal{S}(G/\!\sim \parallel T)$, there exists $\bar{y}'_G \in [\bar{y}_G]$, $\omega'' \in \Omega$ and $u \in (\Sigma_G \cup \Sigma_T)^*$ so that $(\bar{y}'_G, \bar{y}_T) \overset{u\omega''}{\Longrightarrow}^{\mathcal{S}}$ and $\sigma' \leqslant u\omega''$. Note that $(\bar{x}_G, \bar{x}_T) \overset{\sigma}{\rightarrow}^{\mathcal{S}} \overset{t_T}{\Longrightarrow}^{\mathcal{S}} (\bar{y}_G, \bar{y}_T)$. From Proposition 3.2.28, $\sim$ is redirectable and we thus have $(\bar{x}_G, \bar{x}_T) \overset{\sigma t_T}{\Longrightarrow}^{\mathcal{S}} (\bar{y}'_G, \bar{y}_T) \overset{u\omega''}{\Longrightarrow}^{\mathcal{S}}$. □

Example 3.2.6. *Consider the automaton G given in Figure 32.* I $\sim_{inc}$ III *must hold since they both are initial states and can be reached from* IV *via ρ. Besides, since they cannot execute silent events and they have the same set of active regular events,* I $\sim_{ae}$ III *holds. Thus,* I *and* III *can be merged through the active events rule which results in the conflict equivalent $G/\!\sim$.*

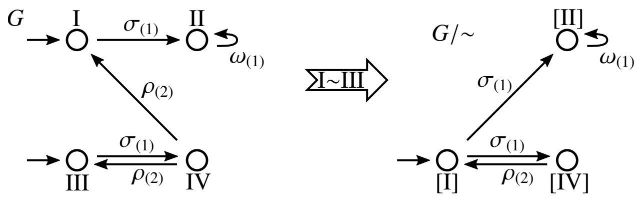

Figure 32: Active events rule

Theorem 3.2.36 (silent continuation rule). *Let $G = \langle Q_G, \Sigma_G, \to_G, Q^\circ_G, M_G\rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq \sim_{inc} \cap \sim_{sc}$. It holds $G \simeq_{\mathcal{S}} (G/{\sim})$.*

Proof. Let $T = \langle Q_T, \Sigma_T, \to_T, Q^\circ_T, M_T\rangle$ be any automaton:

($\Rightarrow$) Same as the proof of Theorem 3.2.35

($\Leftarrow$) Suppose $\mathcal{S}(G/{\sim} \parallel T)$ is non-blocking. Pick $x_G \in Q_G$ and $x_T \in Q_T$ so that $\mathcal{S}(G \parallel T) \overset{s}{\Rightarrow}^{\mathcal{S}} (x_G, x_T)$ for some $s \in (\Sigma_G \cup \Sigma_T)^*$. From Proposition 3.2.33 and the non-blockingness of $\mathcal{S}(G/{\sim} \parallel T)$, for all $\Omega \in M_G \cup M_T$, there exists $t \in (\Sigma_G \cup \Sigma_T)^*$ and $\omega \in \Omega$ so that $\mathcal{S}(G/{\sim} \parallel T) \overset{s}{\Rightarrow}^{\mathcal{S}} ([x_G], x_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$. Consider three cases:

(Case 1) $[x_G]$ is a singleton and there exists some trace in $([x_G], x_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$ which begins with $([x_G], x_T) \overset{\sigma}{\to}^{\mathcal{S}}$ for some $\sigma \in \Sigma_G$. From Propositions 3.2.28, $\sim$ is redirectable. Thus, this case is directly true from 3.2.23.(i).

(Case 2) $[x_G]$ is not a singleton. Since x_G is not in any live-lock but there exists $\tau \in G_{\text{slnt}}(x_G)$, there must exist some $y_G \in Q_G$ so that $x_G \overset{\epsilon}{\Rightarrow} y_G$ and $G_{\text{slnt}}(y_G) = \emptyset$ in G. There are two further possibilities:

(i) There exists some $s_T \in \Sigma^*_{T\setminus G}$, $y_T \in Q_T$ and $\sigma \in \Sigma_G$ so that $(x_G, x_T) \overset{s_T}{\Longrightarrow}^{\mathcal{S}} (y_G, y_T) \overset{\sigma}{\to}^{\mathcal{S}}$ in $\mathcal{S}(G \parallel T)$. Note that $([y_G], y_T)$ must be co-reachable since from Proposition 3.2.33, $([y_G], y_T)$ is reachable in $\mathcal{S}(G/{\sim} \parallel T)$ which is non-blocking. In addition, since $G_{\text{slnt}}(y_G) = \emptyset$, $[y_G]$ must be a singleton. Thus we have reached a Case 1 situation.

(ii) If Case 2.(i) does not hold, then there exist $z_G \in Q_G - \{y_G\}$, $z_T \in Q_T$ and $t_T \in \Sigma^*_{T\setminus G}$ so that $(x_G, x_T) \overset{t_T}{\Longrightarrow}^{\mathcal{S}} (z_G, z_T)$ and $z_G \overset{\tau'}{\longrightarrow}\overset{\epsilon}{\Rightarrow} y_G$ for some $\tau' \in \Upsilon$. In addition, the execution of $z_G \overset{\tau'}{\longrightarrow}^{\mathcal{S}}$ in (z_G, z_T) is disallowed. This could be caused by

a) $(z_G, z_T) \xrightarrow{\sigma}^{\mathcal{S}}$ in $\mathcal{S}(G \parallel T)$ for some $\sigma \in \Sigma_G$ so that $\mathsf{prio}(\sigma) < \mathsf{prio}(\tau')$. This again implies that $[z_G]$ is a singleton state from (SC1) and (SC2), i.e. a Case 1 situation is reached; or

b) z_T is in some n-live-lock[3] in T with $n < \mathsf{prio}(\tau')$. Note that $([z_G], z_T)$ must be co-reachable since from Proposition 3.2.33, $([z_G], z_T)$ is reachable in $\mathcal{S}(G/{\sim} \parallel T)$ which is non-blocking. In this situation, $[z_G]$ cannot execute any transition driven by G in $\mathcal{S}(G/{\sim} \parallel T)$ as well (this is clear if $[z_G]$ is a singleton; otherwise $[z_G]$ is not a singleton, then from (SC2), all its active events are not executable due to the n-live-lock in T, which includes z_T). This implies $M_G = \emptyset$. In addition, $([z_G], z_T)$ is co-reachable in $\mathcal{S}(G/{\sim} \parallel T)$ implies that (z_G, z_T) is co-reachable in $\mathcal{S}(G \parallel T)$.

Note that we do not need to take special care to the situation where the execution of $z_G \xrightarrow{\tau'}^{\mathcal{S}}$ in (z_G, z_T) is preempted by a private active event in z_T whose priority is higher than τ'. This situation must lead to either (i), (ii).a) or (ii).b) in the current case.

(Case 3) $[x_G]$ is a singleton and all traces in $([x_G], x_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$ begin with an event $\alpha \notin \Sigma_G$. If there exists some trace in $([x_G], x_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$ where each state consists of a singleton state from $Q_G/{\sim}$, the current statement is trivially true. Otherwise, let

$$\begin{aligned}([x_G],x_T) = ([x_{G0}], x_{T0}) &\xrightarrow{\alpha_1}^{\mathcal{S}} ([x_{G1}], x_{T1}) \xrightarrow{\alpha_2}^{\mathcal{S}} \cdots \\ \cdots \xrightarrow{\alpha_k}^{\mathcal{S}} ([x_{Gk}], x_{Tk}) &\xrightarrow{\alpha_{k+1}}^{\mathcal{S}} ([x_{Gk+1}], x_{Tk+1}) \xrightarrow{\alpha_{k+2}}^{\mathcal{S}} \cdots \end{aligned} \tag{132}$$

be a trace in $([x_G], x_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$ where $k \geq 0$, $[x_{Gk+1}]$ is not a singleton and all $[x_{Gi}]$ with $i \in \{0, \dots, k\}$ are singletons. Clearly, $([x_{Gk}], x_{Tk}) \xrightarrow{\alpha_{k+1}}^{\mathcal{S}} ([x_{Gk+1}], x_{Tk+1})$ is driven by $G/{\sim}$ since $[x_{Gk}]$ is a singleton while $[x_{Gk+1}]$ is not. Clearly, there exists $x'_{Gk+1} \in [x_{Gk+1}]$ so that $(x_{Gk}, x_{Tk}) \xrightarrow{\alpha_{k+1}}^{\mathcal{S}} (x'_{Gk+1}, x_{Tk+1})$ in $\mathcal{S}(G \parallel T)$. This indicates that Case 3 always reaches a Case 2 situation if at least one non-singleton state is visited in $([x_G], x_T) \overset{t\omega}{\Longrightarrow}^{\mathcal{S}}$. □

Example 3.2.7. *Consider the automaton G given in Figure 33. Clearly,* $\mathrm{II} \sim_{inc} \mathrm{III}$ *holds. In addition,* $\tau_{(2)} \in G_{\mathrm{slnt}}(\mathrm{II}) \cap G_{\mathrm{slnt}}(\mathrm{III})$ *while* $G^{<2}_{\mathrm{rglr}}(\mathrm{II}) = G^{<2}_{\mathrm{rglr}}(\mathrm{III}) = \emptyset$.

[3] Here, we slightly abuse the definition of live-lock in T in that we uniformly substitute all G_{slnt} with T_{prvt} in Definition 3.2.3.

This implies that II $\sim_{sc}$ III *and merging* II *and* III *yields a conflict-preserving abstraction.*

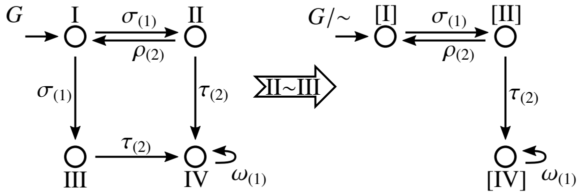

Figure 33: Silent continuation rule

Remark 3.2.2. *Recall that* $\underset{n}{\overset{\epsilon}{\hookrightarrow}}$ *requires for* $n \geq 2$ *at least one silent transition with priority* n. *This requirement can be implicitly fulfilled by adding redundant silent self-loops, which is a PW-bisimilar operation from Lemma 3.2.10. Consider the automaton G given in Figure 34. At first glance,* II $\not\sim_{\text{inc}}$ III *since* I $\overset{\sigma}{\to}\underset{2}{\overset{\epsilon}{\hookrightarrow}}$ II *holds but* I $\overset{\sigma}{\to}\underset{2}{\overset{\epsilon}{\hookrightarrow}}$ III *does* not *hold. Nevertheless, the latter can be rendered valid through appending a* $\tau_{(2)}$*-self-loop in* III, *which is a redundant silent self-loop. This operation enables merging* II *and* III *through silent continuation rule. Thus, when computing the set of* $\hookrightarrow$*-transitions, we always have* $x \underset{n}{\overset{\epsilon}{\hookrightarrow}} x$ *if* $x \xrightarrow{\tau_{(n)}}$ *and no regular event with priority higher than* n *is active in* x.

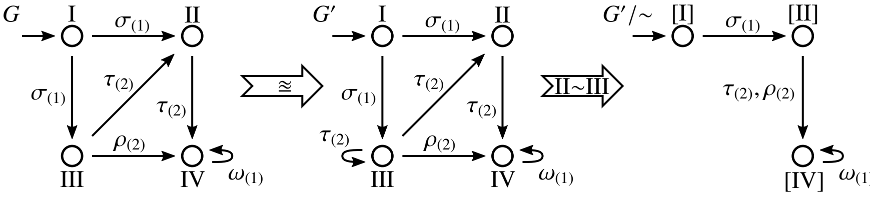

Figure 34: Combining silent continuation rule with redundant silent self-loops

Remark 3.2.3 (A discussion on relaxing $\hookrightarrow$). *At the end of this subsection, we provide a short discussion on the possibility of relaxing the definition of* $\hookrightarrow$*, from which Definition 3.2.25 can potentially be relaxed while the redirectability is still preserved. In particular,* $\hookrightarrow$*-transitions exclude the possibility of preemption through regular events, which is required in Proposition 3.2.32 where we attempted to equate traces that can be synchronised with a same trace from a test. Nevertheless, an obvious situation which is not covered by Proposition 3.2.32 achieves the same goal. We consider the automaton fragment G given in Figure 35, where* II *and* III *are not incoming equivalent from*

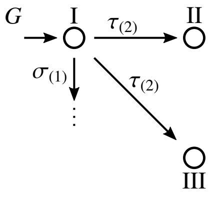

Figure 35: A case of redirectable equivalence which is not strictly incoming equivalent

(I3) since $G^{<2}_{\mathrm{rglr}}(\mathrm{I}) \neq \emptyset$*. Nevertheless, an equivalence only equating* II *and* III *is obviously redirectable (if there are no other incoming transitions in* II *and* III*), since traces* $\mathrm{I} \xrightarrow{\tau_{(2)}} \mathrm{II}$ *and* $\mathrm{I} \xrightarrow{\tau_{(2)}} \mathrm{III}$ *are "the same", not only because their lengths and the silent events on both transitions are the same, but also because the set of active regular events with priority higher than the silent event to execute in each state is the same. In other words, traces* $\mathrm{I} \xrightarrow{\tau_{(2)}} \mathrm{II}$ *and* $\mathrm{I} \xrightarrow{\tau_{(2)}} \mathrm{III}$ *qualify the property stated in Proposition 3.2.32 as well. This observation indeed extends active events rule and silent continuation rule, where the latter one will be exploited in the* only silent out going rule *below; see Definition 3.2.38.*

At the current stage, one may be interested in finding a general relaxation of Definition 3.2.25 which considers preemption through regular events while still achieves redirectability. We believe, however, that such a relaxation is with relatively few practical value. Consider the automaton fragment G given in Figure 36. Consider the trace $\mathrm{I} \xrightarrow{\tau_{(2)}} \mathrm{II} \xrightarrow{\tau_{(3)}} \mathrm{III}$ *in G where* $G^{<2}_{\mathrm{rglr}}(\mathrm{I}) = \{\sigma\}$ *and* $G^{<3}_{\mathrm{rglr}}(\mathrm{II}) = \{\rho\}$ *hold. In this regard, one asks whether an equivalence equating* III *and* VI *is redirectable. In particular, the trace* $\mathrm{IV} \xrightarrow{\tau_{(3)}} \mathrm{V} \xrightarrow{\tau_{(2)}} \mathrm{VI}$ *results from swapping states* I *and* II *in* $\mathrm{I} \xrightarrow{\tau_{(2)}} \mathrm{II} \xrightarrow{\tau_{(3)}} \mathrm{III}$*, including their set of active regular events. Unfortunately, even such a conservative approach cannot achieve redirectability, which can be witnessed by the trace in T as shown in Figure 36. In*

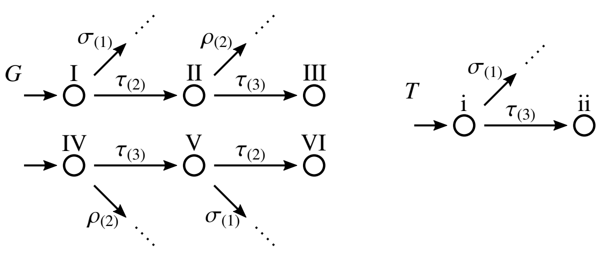

Figure 36: Invalidating redirectability through preemption

particular, when considering the trace $\mathrm{IV} \xrightarrow{\tau_{(3)}} \mathrm{V} \xrightarrow{\tau_{(2)}} \mathrm{VI}$*, preemption through* σ *can be avoided by first executing* $\mathrm{i} \xrightarrow{\tau_{(3)}} \mathrm{ii}$ *in* T*, while this preemption is inevitable in state* (I, i)*. We observe from this example that when preemption through regular events is accounted, the order of the silent transitions is relevant, which in addition requires recording all active regular events in each state on the trace. Recall that given a target state, one only needs to record one incoming* $\hookrightarrow$*-transition through its source state and the lowest priority value among all silent transitions on it. However, allowing preemption through regular events requires recording each silent transition with preemption possibility through regular events explicitly. This drastically enlarges the set of incoming transitions required to compute incoming equivalence. This is from a practical perspective an obvious drawback and thus abandoned in the scope of the current dissertation.*

Complexity of the partition of incoming equivalence From the observation in (Flordal and Malik, 2009), the complexity of computing an incoming equivalence is the same as the complexity of computing the entire incoming transition set, which in our case is the transition relation $\overset{\epsilon}{\underset{\Delta:\sigma}{\Longrightarrow}}\overset{\sigma}{\underset{\Delta:\sigma}{\longrightarrow}}\overset{\epsilon}{\underset{n}{\hookrightarrow}}$. Thus, the overall complexity of computing an incoming equivalence is $\mathcal{O}(\,|Q|^2 \cdot |\Sigma|^2 \cdot N\,)$, where $|Q|^2 \cdot |\Sigma| \cdot N$ is the maximal size of the transition relation $\overset{\epsilon}{\underset{\Delta:\sigma}{\Longrightarrow}}\overset{\sigma}{\underset{\Delta:\sigma}{\longrightarrow}}\overset{\epsilon}{\underset{n}{\hookrightarrow}}$ and is multiplied by $|\Sigma|$ for the active regular event set comparison. $\square$

Complexity of the partition of active-event equivalence The worst case of computing an active-event equivalence is $\mathcal{O}(\,|Q| \cdot |\Sigma|\,)$, i.e. we shall record the set of active regular events for each state. $\square$

Complexity of the partition of silent-continuation equivalence As required in (SC3), we first compute all silent SCCs of the automaton which has the complexity of $\mathcal{O}(\,|\rightarrow|\,) = \mathcal{O}(\,|Q|^2 \cdot |\Sigma|\,)$ based on Tarjan's algorithm (Tarjan, 1972a). This dominates the complexity of comparing whether two states both have outgoing silent transitions, which has the complexity of $\mathcal{O}(\,|Q|\,)$. Thus, the overall complexity of computing an silent-continuation equivalence is $\mathcal{O}(\,|Q|^2 \cdot |\Sigma|\,)$. Note that $\rightarrow$ has at most $|Q|^2 \cdot (|\Sigma| + 1)$ transitions (instead of $|Q| \cdot |A|$) in a Υ-shaped-automaton. $\square$

3.2.3 Further abstraction rules

In this subsection, we extend and modify further abstraction rules introduced in (Flordal and Malik, 2009). First, two abstraction rules resulting from

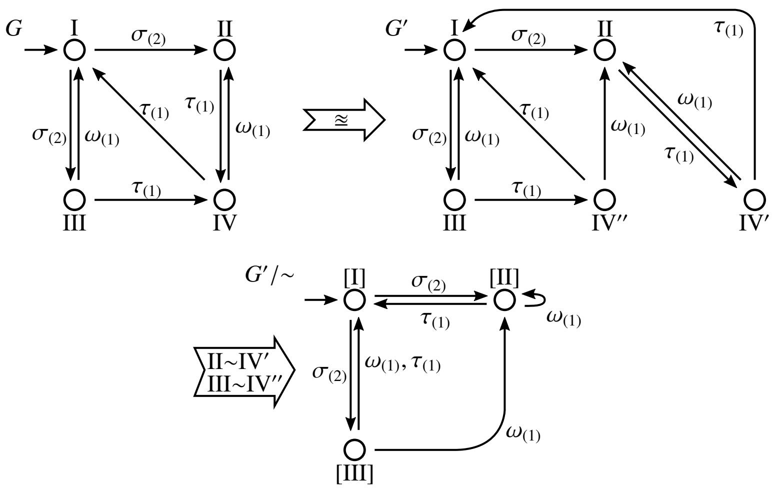

Figure 37: Only silent incoming rule

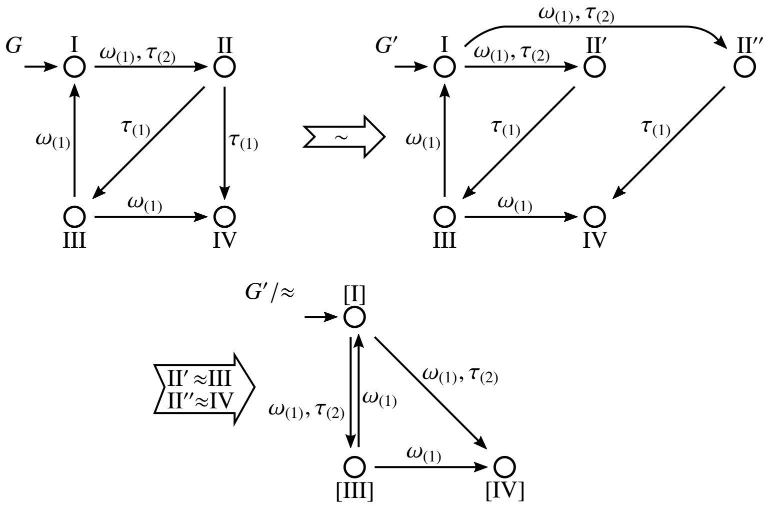

Figure 38: Only silent outgoing rule

combining PWB and the silent continuation rule are to address, i.e. the *only silent incoming rule* and the *only silent outgoing rule*. The idea of modifying the former rule can be illustrated by the following example.

Example 3.2.8. *Consider the automaton G given in Figure 37, from which we construct G' by splitting the state* IV *into two states* IV′ *and* IV″. *It holds* *that $G \approxeq G'$. Afterwards,* II *and* IV$'$ *as well as* III *and* IV$''$ *qualify the silent continuation rule. Merging both classes results in $G'/{\sim}$.*

The observation in the above example inspires the following theorem. The proof is synonymous to that of (Flordal and Malik, 2009, Proposition 2).

Theorem 3.2.37 (only silent incoming rule). *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$ be a Υ-shaped automaton and let $\bar{x} \in Q$ be such that $\bar{x}$ is not in any live-lock, $\tau_{(1)} \in G(\bar{x})$ and $y \xrightarrow{\alpha} \bar{x}$ implies $\alpha = \tau_{(1)}$. For the automaton $G' = \langle Q, \Sigma, \rightarrow', Q^\circ, M \rangle$ with*

$$\rightarrow' = \{(x, \alpha, y) \mid x \xrightarrow{\alpha} y \text{ and } y \neq \bar{x}\} \cup \{(x, \alpha, y) \mid x \xrightarrow{\tau} \bar{x} \xrightarrow{\alpha} y\}, \qquad (133)$$

it holds that $G \simeq^{\mathcal{S}} G'$.

Note that the silent event utilised to enable the only silent incoming rule must be $\tau_{(1)}$. As for G' in Figure 37, this ensures that II $\sim_{\text{inc}}$ IV$'$ (and III $\sim_{\text{inc}}$ IV$''$). Replacing e.g. all transition labels $\tau_{(1)}$ with $\tau_{(2)}$ in G' results in a situation where II $\overset{\epsilon}{\underset{1}{\hookrightarrow}}$ II but II $\overset{\epsilon}{\underset{1}{\not\hookrightarrow}}$ IV$'$, which invalidates the incoming equivalence. In addition, it is worth mentioning that in Theorem 3.2.37, it suffices to check that $\bar{x}$ is not in any live-lock, since this implies that none of its predecessors is in any live-lock.

Complexity of the only silent incoming rule The only silent incoming rule can be implemented as such that for each state, we first check whether all its incoming transitions are labelled by $\tau_{(1)}$ and redirect all its outgoing transitions to its predecessor states. The operations for a single state has thus the complexity of $\mathcal{O}(\,|Q|^2 \cdot |A|\,)$, which implies that the overall complexity of the current implementation is $\mathcal{O}(\,|Q|^3 \cdot |A|\,)$. □

We now consider the only silent outgoing rule, which first conversely applies the silent continuation rule, then utilises PWB. Consider the following example.

Example 3.2.9. *Consider the automaton G given in Figure 37. By conversely applying the silent continuation rule, state* II *is split into two states* II$'$ *and* II$''$ *in G' so that* II$'$ *and* II$''$ *qualify the silent continuation rule. Note that* II$'$ *and* II$''$ *are not strictly incoming equivalent from Definition 3.2.25. Nevertheless, silent continuation rule can still be applied through the observation in Remark 3.2.3 and Figure 35. Afterwards, a PWB $\approx$ on G' can be found where* II$'$ $\approx$ III *and* II$''$ $\approx$ IV*. By constructing the quotient automaton of G' w.r.t. $\approx$, $G'/{\approx}$ is obtained.*

The observation in the above example inspires the following theorem. Its proof is synonymous to that of (Flordal and Malik, 2009, Proposition 3).

Theorem 3.2.38 (only silent outgoing rule). *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$ be a Υ-shaped automaton and let $\bar{x} \in Q$ be such that $\bar{x}$ is not in any live-lock and $G(\bar{x}) = \{\tau_{(1)}\}$. Let $\bar{Q} := \{y \in Q \mid \bar{x} \xrightarrow{\tau_{(1)}} y\}$ and $G' = \langle Q - \{\bar{x}\}, \Sigma, \rightarrow', Q^{\circ\prime}, M \rangle$ with*

$$Q^{\circ\prime} = \begin{cases} Q^\circ & \text{if } \bar{x} \notin Q^\circ \\ (Q^\circ - \{\bar{x}\}) \cup \bar{Q} & \text{if } \bar{x} \in Q^\circ \end{cases}; \tag{134}$$

$$\rightarrow' = \{(x, \alpha, y) \mid x \xrightarrow{\alpha} y \text{ and } \bar{x} \notin \{x, y\}\} \cup \{(x, \alpha, y) \mid x \xrightarrow{\alpha} \bar{x} \text{ and } y \in \bar{Q}\}. \tag{135}$$

It holds that $G \simeq^{\mathcal{S}} G'$.

Note that in Theorem 3.2.38, again, all outgoing transitions of a state for applying the only silent outgoing rule must be $\tau_{(1)}$. This is caused by the $\overset{\epsilon}{\underset{1}{\Rightarrow}}$-fragment in transitions defining PWB; see Definition 3.2.7.

Complexity of the only silent outgoing rule Similar to the only silent incoming rule, the only silent outgoing rule can be implemented as such that for each state, we check whether all its outgoing transitions are labelled by $\tau_{(1)}$ and redirect all its incoming transitions to its successor states. The overall complexity of the current implementation is thus $\mathcal{O}(\,|Q|^3 \cdot |A|\,)$ as well. □

Another powerful conflict-preserving abstraction rule is the *certain conflicts rule*. Basically, if only the non-blockingness is to check, the exact structure of the blocking part of the automaton is not of our interest and thus can be merged into a single blocking state. In addition, upon reaching some co-reachable states, blockage under synchronisation is inevitable. Outgoing transitions from such states are thus (partially) removed. Consider the following two examples.

Example 3.2.10. *Consider the automaton G given in Figure 39, where* III *is a blocking state while all other states are non-blocking. For any automaton T so that $\mathcal{S}(G \parallel T)$ can reach* II *in G, ρ in* II *must be executable in order to let $\mathcal{S}(G \parallel T)$ be non-blocking. However, in this case, $\tau_{(2)}$ must be executable in the same state as well, which leads to the blocking state* III*. Thus, reaching* II *under shaped synchronous composition certainly leads to a blocking situation. We thus remove all outgoing transitions from* II *which renders* II *blocking and* *subsequently renders* I *blocking as well. By merging all blocking states, G' is constructed.*

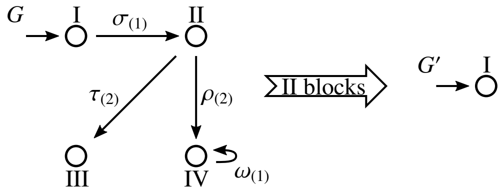

Figure 39: Blocking silent transition

Example 3.2.11. *Consider the automaton G given in Figure 40, where* III *is blocking state while all other states are non-blocking. For any automaton T so that $\mathcal{S}(G \parallel T)$ can reach* II *in G, ρ in* II *must be executable in order to let $\mathcal{S}(G \parallel T)$ be non-blocking by reaching* IV*. However, in this case, the blocking* III *can be reached by ρ as well. Thus, the transition* II $\overset{\sigma}{\rightarrow}$ IV *does not contribute to the non-conflictingness and could be removed.*

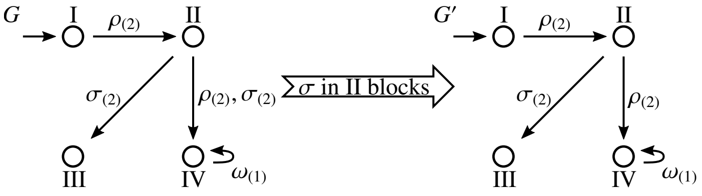

Figure 40: Blocking non-deterministic regular transitions

The above two examples are inspired by the *limited certain conflicts rule* introduced in (Malik and Ware, 2020), which motivates the following statement. Note that merging blocking states is omitted for brevity.

Theorem 3.2.39 (certain conflicts rule). *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$ be a Υ-shaped automaton. Let $Q_\mathrm{c} \subseteq Q$ be the set of co-reachable states in G and $Q_\mathrm{uc} := Q - Q_\mathrm{c}$ the set of non-co-reachable states in G. Define two transition sets as*

$$\rightarrow_1 := \{x \overset{\alpha}{\rightarrow} y \,|\, x \in Q_\mathrm{c}, \alpha \in A, y \in Q \text{ and } \exists y' \in Q_\mathrm{uc}, \tau \in \Upsilon .\, G^{<\tau}_\mathrm{rglr}(x) = \emptyset \wedge x \overset{\tau}{\rightarrow} y' \}; \tag{136}$$

$$\rightarrow_2 := \{x \overset{\sigma}{\rightarrow} y \,|\, x \in Q_\mathrm{c}, \sigma \in \Sigma, y \in Q_\mathrm{c}, G^{<\sigma}_\mathrm{rglr}(x) = \emptyset \text{ and } \exists y' \in Q_\mathrm{uc} .\, x \overset{\sigma}{\rightarrow} y' \} \tag{137}$$

and let $G' = \langle Q, \Sigma, \rightarrow - (\rightarrow_1 \cup \rightarrow_2), Q^\circ, M \rangle$. It holds that $G \simeq^{\mathcal{S}} G'$.

It is worth mentioning that the certain conflicts rule as suggested in Theorem 3.2.39 abstracts an automaton through transition removal, which may render co-reachable states non-co-reachable. Thus, the certain conflicts rule can be iteratively applied until reaching the fix-point, as the transition removal operation in Theorem 3.2.39 is obviously monotonic.

Complexity of the certain conflicts rule The transition removal is based on finding blocking states in an automaton. This can be achieved by a backward depth-first search on states marked by each marking set, which has the complexity of $\mathcal{O}(\,|\rightarrow| \cdot |M|\,) = \mathcal{O}(\,|Q|^2 \cdot |\Sigma| \cdot |M|\,)$. In addition, the transition removal can be iteratively performed. In the worst case, each iteration renders one co-reachable state un-co-reachable, which leads to maximally $|Q|$ iterations. Thus, the overall complexity of the certain conflicts rule is $\mathcal{O}(\,|Q|^3 \cdot |\Sigma| \cdot |M|\,)$. □

3.3 Compositional verification

With the abstraction rules developed in the previous section, we are now in the position to perform compositional non-blockingness verification w.r.t. prioritised events. Recall that given a family of automata $(G_i)_{1\leq i\leq k}$, the global behaviour amounts to $G := \mathcal{S}(G_1 \parallel G_2 \parallel \cdots \parallel G_k)$, where each G_i is to abstract through the developed abstraction rules. Afterwards, by iteratively choosing modules to compose and perform abstraction on the composed automaton, only one automaton lefts, whose non-blockingness coincides with the non-conflictingness of the input family of automata. Following (Pilbrow and Malik, 2015, Algorithm 1), this procedure is illustrated by pseudo codes in Algorithm 2 where the main function IsNonConflicting performs the compositional verification procedure and invokes the function ConflictPreservingAbstraction to apply individual abstraction rules. In the following, we clarify Algorithm 2 in detail.

The main function IsNonConflicting takes a family of automata $\mathfrak{G} = \{G_1, \ldots, G_k\}$ whose non-conflictingness is to check. For at least two automata in $\mathfrak{G}$, each $G \in \mathfrak{G}$ is abstracted through conflict-preserving abstractions, which is addressed by the for-loop in Line 3. To introduce silent events, recall from Definition 3.1.6 that transition hiding is to perform. In addition, the automaton to abstract should be in Υ-shaped form. To achieve these two prerequisites, the set of private events Π (which includes silent events) is figured out. From Remark 3.2.1, we can shape w.r.t. private events, which not only implies Υ-shapedness but also renders more states unreachable. This shaping operation is performed by the $\mathcal{S}_\Pi(\cdot)$-operator in Line 5, whose definition can

Algorithm 2 Compositional non-blockingness verification

```
 1: function IsNonConflicting($\mathfrak{G}$)
 2:     if $|\mathfrak{G}| > 1$ then
 3:         for all $G \in \mathfrak{G}$ do
 4:             $\Pi \leftarrow \{\alpha \in \mathfrak{E} \mid \alpha$ is private in $G$ w.r.t. $\mathfrak{G}\}$
 5:             $G \leftarrow \mathcal{S}_\Pi(G)$                    ▷ Shape w.r.t. to private events
 6:             $G \leftarrow$ Hide$(G, \Pi)$
 7:             $G \leftarrow$ ConfilctPreservingAbstraction$(G)$
 8:         end for
 9:         while $|\mathfrak{G}| > 1$ do
10:             pick $G_i, G_j \in \mathfrak{G}$ and let $H = G_i \parallel G_j$   ▷ Strategically choose
    modules to compose
11:             $\Pi \leftarrow \{\alpha \in \mathfrak{E} \mid \alpha$ is private in $H$ w.r.t. $\mathfrak{G} - \{G_i, G_j\}\}$
12:             $H \leftarrow \mathcal{S}_\Pi(H)$
13:             $H \leftarrow$ Hide$(H, \Pi)$
14:             $H \leftarrow$ ConfilctPreservingAbstraction$(H)$
15:             $\mathfrak{G} \leftarrow (\mathfrak{G} - \{G_i, G_j\}) \cup \{H\}$
16:         end while
17:     end if
18:     let $G$ be the only automaton left in $\mathfrak{G}$
19:     return IsNonBlocking$(\mathcal{S}(G))$
20: end function

21: function ConfilctPreservingAbstraction($G$)
22:     $G \leftarrow$CertainConflictsRule$(G)$
23:     $G \leftarrow$RedundantSilentStep$(G)$
24:     $G \leftarrow$OnlySilentRules$(G)$       ▷ only silent incoming and outgoing
    rules
25:     $G \leftarrow$PrioritisedWeakBisimulation$(G)$
26:     $G \leftarrow$IncomingEquivalenceRules$(G)$ ▷ active event rule and silent
    continuation rule
27:     return $G$
28: end function
```

be synonymously obtained from Definition 3.2.1 through uniform substitutions. Afterwards, transition hiding is performed through the function Hide. Note that since $\mathcal{S}_\Pi$ has been performed, hiding any private regular transition into a silent transition preserves Υ-shapedness.

At this stage, we shall take a deeper look into the transition hiding operation. Generally, function HIDE shall iterate over all transitions and check whether it is hidable; see Definition 3.1.7. To this end, Proposition 3.1.8 suggested that transitions labelled by private regular events not carrying marking information can be hidden. This conservative statement can be relaxed by analysing the following example.

Example 3.3.1. *Consider the automaton G given in Figure 41 and suppose that the event ω, which carries marking information, is private w.r.t. some given rest part H in a modular system. In this circumstance, hiding $\mathrm{I} \xrightarrow{\omega} \mathrm{II}$ in G preserves the non-conflictingness with H. If $\mathcal{S}(G \parallel H)$ is non-blocking, then upon reaching* I *in G, executing ω in* I *must be possible under synchronisation. If this is possible, then since ω is private in G, subsequently executing ω in* II *must be possible as well. Thus, transition $\mathrm{II} \xrightarrow{\omega} \mathrm{III}$ is sufficient for reasoning non-conflictingness from any state being in* I*, indicating that $\mathrm{I} \xrightarrow{\omega} \mathrm{II}$ can be hidden. Indeed,* I *and* II *in G' can be merged through e.g. PWB, which was not possible in G.*

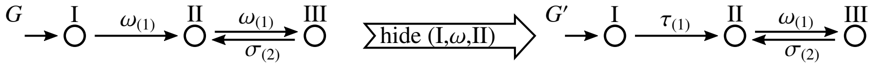

Figure 41: Hiding private transition with marking information

From the above example, it can be concluded that for a private marking transition, it can be hidden if it is ensured that in the future, another private marking transition (within the same marking set) can be reached. This requirement can be fulfilled by the $\overset{\epsilon}{\underset{\Delta:n}{\Longrightarrow}}$-transition, which motivates the following proposition.

Proposition 3.3.1. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q_G^\circ, M_G \rangle$ be a Υ-shaped automaton and $H = \langle Q_H, \Sigma_H, \rightarrow_H, Q_H^\circ, M_H \rangle$ be an automaton. Let $t = (\bar{x}_G, \sigma, \bar{y}_G) \in \rightarrow_G$ with $\sigma \in \Sigma_G - \Sigma_H$ be such that for all $\Omega_G \in M_G$ so that $\sigma \in \Omega_G$, there exists $\sigma' \in \Omega_G - \Sigma_H$ so that the following two statements hold:*

(i) $\mathsf{prio}(\sigma') \leq \mathsf{prio}(\sigma)$;

(ii) $\bar{y}_G \overset{\epsilon}{\underset{\Delta:\sigma}{\Longrightarrow}} \bar{z}_G \xrightarrow[\Delta:\sigma']{\sigma'}$ *for some $\bar{z}_G \in Q_G - \{\bar{x}_G\}$ and $\Delta = \Sigma(\bar{x}_G)$.*

It holds that t is hidable w.r.t. H.

Proof. Since the $(\Leftarrow)$ case is trivial, we only prove the $(\Rightarrow)$ case, i.e. we assume that $\mathcal{S}(G \parallel H)$ is non-blocking and attempt to prove that $\mathcal{S}(G/_t \parallel H)$ is non-blocking. Note that $\mathcal{S}(G \parallel H)$ and $\mathcal{S}(G/_t \parallel H)$ have the same set of reachable states. Let $x_G \in Q_G$ and $x_H \in Q_H$ be such that (x_G, x_H) is reachable in $\mathcal{S}(G/_t \parallel H)$. Obviously, (x_G, x_H) is reachable in $\mathcal{S}(G \parallel H)$ as well. For all $\Omega_G \in M_G$ so that $\sigma \in \Omega_G$ and

$$(x_G, x_H) \overset{s}{\Rightarrow}^{\mathcal{S}} (\bar{x}_G, \bar{x}_H) \overset{\sigma}{\rightarrow}^{\mathcal{S}} (\bar{y}_G, \bar{x}_H) \tag{138}$$

in $\mathcal{S}(G \parallel H)$ for some $\bar{x}_H \in Q_H$ and $s \in (\Sigma_G \cup \Sigma_H)^*$, we must have

$$(\bar{x}_G, \bar{x}_H) \overset{\tau}{\rightarrow}^{\mathcal{S}} (\bar{y}_G, \bar{x}_H) \overset{\epsilon}{\Rightarrow}^{\mathcal{S}} (\bar{z}_G, \bar{x}_H) \overset{\sigma'}{\longrightarrow}^{\mathcal{S}} \tag{139}$$

in $\mathcal{S}(G/_t \parallel H)$ where $\tau = \mathsf{hide}(\sigma)$. Note that $(\bar{x}_G, \bar{x}_H)$ can be reached from (x_G, x_H) in $\mathcal{S}(G/_t \parallel H)$ as well. □

Complexity of searching hidable transitions For each transition, all its multi-step silent successor states are to determine (maximally $|Q|$) where a comparison of the active regular event set is necessary (with complexity $\mathcal{O}(\,|\Sigma|\,)$). For each furthest silent successor, one check for each marking set (maximally $|M|$) whether some private active regular event is in this marking set (with the complexity $\mathcal{O}(\,|\Sigma|\,)$). The overall complexity of transition hiding is thus $\mathcal{O}(\,|Q| \cdot |{\rightarrow}| \cdot |\Sigma|^2 \cdot |M|\,) = \mathcal{O}(\,|Q|^3 \cdot |\Sigma|^3 \cdot |M|\,)$. □

Note that Proposition 3.3.1 covers Proposition 3.1.8, i.e. it includes the trivial case where the transition to hide is irrelevant to marking. In addition, in (ii) of Proposition 3.3.1, $\bar{z}_G \neq \bar{x}_G$ ensures that $\bar{z}_G \overset{\sigma'}{\longrightarrow}$ and $\bar{x}_G \overset{\sigma}{\rightarrow} \bar{y}_G$ must be two distinct transitions. With the help of Proposition 3.3.1, we are now able to achieve more abstraction possibilities due to the potentially enlarged set of silent transitions.

We now resume the clarification of Algorithm 2. After hiding in Line 6, G can be abstracted by applying abstraction rules developed in Section 3.2. This invokes the function ConflictPreservingAbstraction in Line 21 which performs individual abstraction rules in a strategical order. After all automata have been abstracted, the while-loop in Line 9 is entered. The loop start with the composition of a strategically picked pair of modules, which may drastically influence the verification performance. In the context with prioritised events, it is heuristically preferred that choosing the modules shall render as many regular events private as possible. The reason for applying this strategy is such that this benefits the $\mathcal{S}_\Pi$-shaping operation, which itself is very efficient to perform (with the complexity $\mathcal{O}(\,|Q| \cdot |A|\,)$). After composing the chosen modules G_i and G_j into H, private events in H are figured out and the abstraction procedure is applied to H again. Overall, the while-loop in Line 9 reduces the size of $\mathfrak{G}$ by one in each iteration. Finally, only one automaton is left in $\mathfrak{G}$, say G. The non-conflictingness of the input $\mathfrak{G}$ coincides with the non-blockingness of $\mathcal{S}(G)$, which is returned as the result of the entire algorithm.

3.4 Case studies

In this section, two typical use-cases are considered where the behaviour of a modular discrete event system is restricted by prioritised events. The first case addresses the SBD verification problem, where the model established in Section 2.4 is utilised. The second case handles the problem of mimicking executor semantics, where an executor always discards low-priority events when other high-priority events are active. For relevant performance evaluations where the computation duration is mentioned, the verification algorithm is implemented and tested on an office computer with an Intel Core i7-10510U 2.30 GHz CPU and 16 GB RAM within the C++ framework of the libFAUDES library (Moor, Schmidt et al., 2008).

3.4.1 Synchronised SBDs

We recall the example modelled in Section 2.4 where five (partially) nested SBDs describe the control sequences of a modular system. Following the translation procedure in Section 2.2, the global closed-loop behaviour is represented by five automata, whose non-conflictingness is to verify. The five automata translated from the five SBDs S_{PROC}, S_{TAKE}, S_{SEND}, S_1 and S_2 are named F_{PROC}, F_{TAKE}, F_{SEND}, F_1 and F_2, respectively. By recalling Sections 2.2.3 and 2.3.1, it is clear that for the current example, all hyper-edge events and done events are with the highest priority 1 while all other events (i.e. variable events) are with priority 2; see Table 3.

Table 3: Priority assignment of the SBD example

events	priority
$\Sigma^{\mathsf{GL}}_{\mathsf{HEs}}$	1
$\Sigma^{\mathsf{GL}}_{\mathsf{D}}$	1
$\Sigma^{\mathsf{GL}}_{\mathsf{VAR}}$	2

To verify the non-conflictingness, the marking set of each automaton is first to determine. From a practical perspective, it is to expect that each SBD always has the opportunity to proceed, i.e. firing hyper-edges is always possible in the future. This motivates the following marking set assignment of F_{PROC}, F_{TAKE}, F_{SEND}, F_1 and F_2, respectively:

$$M_{\mathsf{PROC}} = M_{\mathsf{TAKE}} = M_{\mathsf{SEND}} = \emptyset; \tag{140}$$

$$M_1 = \{\{\ \mathsf{HE[S[101]T[102]}\,\}\}; \tag{141}$$

$$M_2 = \{\{\mathsf{HE[S[201]T[206]}\,\}, \{\mathsf{HE[S[205]T[206]}\,\}\}. \tag{142}$$

Note that it is safe to aggressively let $M_{\mathsf{PROC}} = M_{\mathsf{TAKE}} = M_{\mathsf{SEND}} = \emptyset$. If their invokers can be reached indefinitely in the root SBD S_1, then each non-root SBD can be indefinitely started as well. On the other hand, the termination of all non-root SBDs is necessary for S_1 to proceed. Besides, for S_1, it suffices to choose any hyper-edge in the sole loop for an event set in the marking set M_1, e.g. HE[S[101]T[102]. However, this is not the case for SBD S_2, since it contains two branching loops. To guarantee that both loops can be entered in the future, it is necessary to assign two event sets in the marking set M_2, where each event set corresponds to one branch. In addition, recall from Line 10 of Algorithm 2 that a wise choice of composing modules may drastically influence the duration of verification. In particular, we shall attempt to render as much regular events private as possible. To this end, we arrange all input automata in a given sequential order and implement Line 10 of Algorithm 2 as such that always the first two automata of the sequence are composed. In addition, Line 15 always pushes the newly composed abstracted automaton to the first position of the sequence. In this regard, we arrange the five automata in the order of

$$F_1, F_{\mathsf{PROC}}, F_{\mathsf{TAKE}}, F_{\mathsf{SEND}}, F_2. \tag{143}$$

This order attempts to handle M1 and its submodules first, which will efficiently render events in M1 private. Under the current set-up, the verification terminates in 1.03s, which shows that the global closed-loop behaviour is in fact blocking.

To analyse how blocking states are reached, it is sometimes desired to construct a trace as a counterexample to show how blocking states can be reached. Note that the final automaton resulting from the compositional verification is normally much smaller than the explicit monolithic representation. Thus, computing counterexample based on the final automaton only results in a too abstract trace, which is difficult for the user to diagnose how blocking states are reached, since such an abstract counterexample cannot be executed in the monolithic representation. To this end, we experimentally implement the State Merging Expansion (SME) algorithm introduced in (Malik and Ware, 2020) to compute a trace that reaches some blocking state in the monolithic

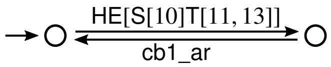

Figure 42: Hypothesis which resolves the blockage

representation.[4] From the resulting counterexample, it can be diagnosed that the blockage can be caused by lacking preconditions for CB1 to take a workpiece. From the plant model in Appendix A, especially G_{1-2_1} in Figure 59, CB1 can receive a workpiece as long as no workpiece is currently available at the workpiece sensor of CB1 and the motor of CB1 is off. This indicates the possibility that, when SBD S_{TAKE} is active with the token configuration $[11, 13]$, more than one workpiece can pass through CB1. This contradicts the design purpose, since it is intended that each time when S_{TAKE} is invoked, only one workpiece is transported from SF1 to CB2. To solve the issue, consider the hypothesis illustrated in Figure 42, which sets firing $\mathsf{HE}[\mathsf{S}[10]\mathsf{T}[11, 13]]$ as a precondition for CB1 to receive a workpiece. This requirement can be realised by externally restricting the behaviour of SF1. By considering the automaton in Figure 42 as an additional plant component when translating S_{TAKE}, the global behaviour turns out to be non-conflicting.

Table 4: Verification duration of the SBD example

orig. order, full abst.	orig. order, shape only	bad order, full abst.
1.03s	10.40s	20.75s

To evaluate the performance of compositional verification, we consider two suboptimal configurations for the compositional verification of the current example. Note that the troubleshooting automaton in Figure 42 is not taken into consideration. The resulting verification durations are recorded in Table 4.

[4] If only state-merging abstraction rules (i.e. all abstraction rules introduced in Section 3.2.1, 3.2.2 and 3.2.3 except the certain conflicts rule) are considered, we could utilise SME to expand a counterexample: suppose a $\mathcal{S}_{\Pi}$-shaped automaton G is abstracted into G', and the rest part of the synchronisation is H. If a counterexample in $\mathcal{S}(G' \parallel H)$ is given, SME expands the counterexample so that it can be executed in $\mathcal{S}(G \parallel H)$. Technically, by performing A*-search (Hart et al., 1968), SME searches successors within each equivalence class, which were merged into a single state in the abstract counterexample. Meanwhile, reaching the silent successor should also be allowed by the rest part H. In this regard, SME effectively unfolds each equivalence class so that abstracted information (such as merged non-deterministic choices and merged silent sequences) are reconstructed.

- Skip the function ConflictPreservingAbstraction in Algorithm 2, i.e. we only utilise $\mathcal{S}_\Pi$-shaping as a naive abstraction rule. In this situation, the monolithic global behaviour is indeed explicitly constructed and the verification procedure takes 10.40s to terminate.
- All abstraction steps are executed, but the input order of automata is rearranged by
$$F_1, F_2, F_{\mathsf{PROC}}, F_{\mathsf{TAKE}}, F_{\mathsf{SEND}}. \tag{144}$$
Conceivably, this sequence is rather inefficient in that the local behaviour in M1, i.e. taking and sending workpieces from M1 which are specified in F_{TAKE} and F_{SEND}, comes at the end of the sequence. In this case, the verification procedure takes 20.75s to terminate.

3.4.2 Priority in control hardware

In this subsection, we investigate the scenarios when finite automata are implemented as control programmes in hardware; see e.g. (Fabian and Hellgren, 1998; Moor, 2022). In particular, we focus on the choice of simultaneously activated events, i.e. at some state in an automaton, multiple outgoing transitions labelled by different events can be executed. We envisage the following cases:

- Consider implementing a finite automaton as the controller of a conveyor belt (see Figure 20). In particular, the user may wish to stop the conveyor belt whenever a workpiece arrives. Figure 43 shows a fragment of the controlled behaviour, where events ar, lv and off denote the arrival / departure of the workpiece at / from the workpiece sensor and turning off the conveyor belt motor, respectively. Note that ar and lv correspond to the behaviour of the workpiece sensor, while off is associated with some control instruction. In fact, similar concept of “writable variables” was proposed for SBDs in Section 2.3.2 as well. At the current stage, one may ask why event lv is active at state II while event off is active. This

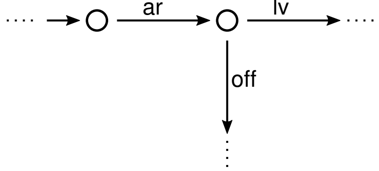

Figure 43: A fragment of the controlled behaviour of a conveyor belt

can be caused by the synthesis rule of the controller, e.g. in Supervisory Control Theory, sensor events are generally not allowed to be disabled. In particular, if other automata are to synchronously operate the conveyor belt, off may be disabled by other modules in the state II, which possibly need to be verified. To this end, it is natural to propose that control instructions shall have higher priority over sensor events (Qamsane et al., 2016). This can indeed be considered as an assumption over the timed behaviour of the automata (Brandin and W. M. Wonham, 1994), i.e. control instructions are always taken sufficiently rapidly without "waiting" for subsequent sensor events.

- In addition to the above situation, one may also expect to assign different priorities to different instructions. This enables more flexibility in expressing the control specification as well.

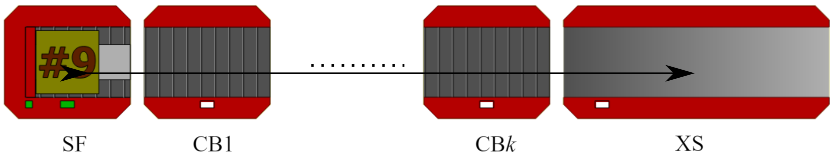

Figure 44: Concatenated conveyor belts

Similar to Figure 20, we consider another practical use-case as depicted in Figure 44 where workpieces are transported from a stack feeder (SF) on the left to an exit slide (XS) on the right via k concatenated conveyor belts (CBi). Each component is equipped with a sensor to indicate the availability of a workpiece, while each CB is driven by a motor. The plant behaviour of each conveyor belt is described by G_i in Figure 45, where C_i additionally describes

Table 5: Events in the conveyor belts example

event	description	priority
on_i	turn on the motor of CBi	2
off_i	turn off the motor of CBi	2
ar_i	workpiece arrival at the sensor of CBi ($i = 0$ for SF, $i = k + 1$ for XS)	3
lv_i	workpiece departure from the sensor of CBi ($i = 0$ for SF, $i = k + 1$ for XS)	3
sd_i	send workpiece from CBi to CB$i + 1$ ($i = 0$ for SF)	1

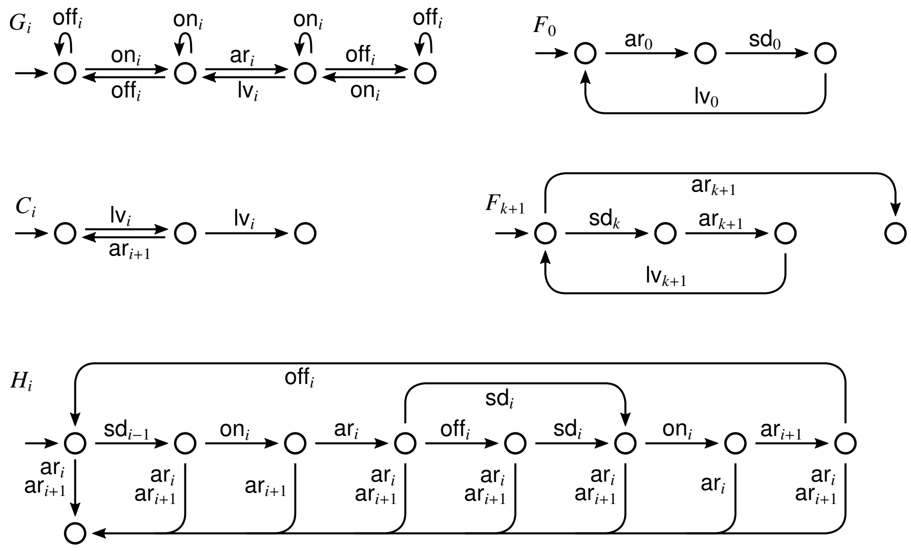

Figure 45: Automata of the conveyor belts example

the coupling between CBi and CB$i+1$. To control each conveyor belt CBi in a modular fashion, a modular controller H_i for each CBi is given in Figure 45 as well. In particular, the event sd$_{i-1}$ constitutes an internal synchronisation instruction, by executing which a workpiece is sent from CBi to CB$i+1$. Each modular controller H_i cyclically

(i) takes a workpiece from CB$i-1$ (or SF as CB0) by turning on the motor;

(ii) when a workpiece arrives, either proceeds sending the workpiece to CB$i+1$ (or XS as CB$k+1$) by directly executing sd$_i$, or stops CBi until sd$_i$ becomes executable, and

(iii) stops CBi after CBi has received the workpiece.[5]

It is worth mentioning that the event sd$_i$ appears both in H_i and H_{i+1}. Thus, sd$_i$ in H_i may be deactivated by H_{i+1}, i.e. CB$i+1$ is not ready to receive a workpiece. It is of course safe to always first execute off$_i$, then wait until sd$_i$ becomes executable. However, this implementation causes a "stutter" if CB$i+i$ is indeed directly ready for receiving, i.e. the motor of CBi is turned off and then immediately turned on. To solve this issue, the user can specify that the priority of sd$_i$ is higher than that of off$_i$, i.e. if sd$_i$ is executable, the possibility of executing off$_i$ should be discarded. Finally, we note that each

[5] A branching similar to (ii) can be realised here as well, which is omitted for simplicity.

controller H_i rejects all unexpected occurrence of sensor events, which lead to a dedicated blocking state.

With all the events appearing in the current example listed in Table 5, we are in the position to consider their priority assignment. As discussed above, we assume that the controller always reacts immediately upon the occurrence of sensor events. This motivates us to set sensor events with the lowest priority. Furthermore, we assume that the controller always prefer internal instructions (sd_i) to actuator manipulations (on_i, off_i), indicating that sd_i are assigned with the highest priority. In addition, each conveyor belt CBi, $1 \leq i \leq k$ forms a local closed-loop

$$F_i := G_i \parallel H_i. \tag{145}$$

Besides, SF and XS in Figure 44 are considered being controlled externally which result in individual local closed loops F_0 and F_{k+1} in Figure 45, respectively. In this regard, the overall closed-loop behaviour with k conveyor belts complies with

$$F := \mathcal{S}\left(\big\|_{0\leq i\leq k} (\underbrace{F_i \parallel C_i}_{E_i}) \parallel \underbrace{F_{k+1}}_{E_{k+1}} \right), \tag{146}$$

whose non-blockingness is to verify.

Again, to prepare for the verification, we shall first determine the marking set of each input automaton. From (146), it is considered that $k+2$ automata $E_0, E_1, \dots, E_{k+1}$ are available as input for the compositional verification. These are assigned with marking sets

$$M_i = \{\{\mathsf{ar}_i\}\} \tag{147}$$

for all $i \in \{0, 1, \dots, k+1\}$. In addition, the input order of the automata for the verification is

$$E_0, E_1, \dots, E_{k+1}. \tag{148}$$

This order iteratively packs the left side of the plant into a single module and localises all plant events in the left most module. The elapsed time for verification as well as the final state count are listed in Table 6, where the first column shows the count of conveyor belts and the second column shows the state count of the monolithic behaviour. Column "mono time" lists the elapsed time for compositional verification in second if only $\mathcal{S}_\Pi$-shaping is applied as the single abstraction rule, while the entire function ConflictPreservingAbstraction in Algorithm 2 is skipped. In this case, the complete monolithic behaviour is indeed constructed at the end. It can be observed that both the second and the third column grow exponentially

Table 6: State count and elapsed time

k	mono. state cnt.	mono. time	abst. state cnt.	abst. time
5	3.4×10^3	0.28s	35	0.06s
6	9.9×10^3	0.81s	40	0.09s
7	2.8×10^4	2.79s	45	0.12s
8	7.7×10^4	8.60s	50	0.17s
9	2.1×10^5	26.23s	55	0.22s

w.r.t. to the count of conveyor belts. Contrarily, the last two columns show the information when the entire Algorithm 2 is applied, i.e. we do not skip the function ConflictPreservingAbstraction. The fourth column shows the state count in the final automaton while the last column shows the elapsed time for the entire verification procedure. We observe that, when all available abstraction rules are applied, the state count of the final automata (as well as the elapsed time) grows linearly, which is caused by the fact that the global behaviour only depends on the availability of a workpiece at each sensor. In addition, a significant reduction of computational cost can be observed as well by comparing the third and the fifth column. Finally, it is worth mentioning that assigning the marking sets as

$$M_0 = \{\{\mathsf{ar}_0\}\}; \tag{149}$$

$$M_1 = M_2 = \cdots = M_{k+1} = \emptyset \tag{150}$$

is indeed reasonable for the current example, since if SF never receives a workpiece any more, all subsequent CBs will eventually block due to the lack of workpiece supply; on the other hand, if any CB jams, all preceding CBs will eventually jam as well, which prevents SF to take a new workpiece. In this situation, if all available abstraction rules are applied and the input order suggested in (148) is adopted, the final state count of the abstraction is constantly 10 for any $k \in \mathbb{N}$. The reason is that the global behaviour now only depends on whether CB1 currently owns a workpiece (w.r.t. to the marking requirement) and the subsystem consisting of all components from CB2 to XS behaves equivalently to a single XS. In other words, due to relaxed marking requirements, more transitions are hidable in this situation.

Concluding remarks

In this section, the compositional non-blockingness verification problem w.r.t. prioritised events has been addressed. To verify the non-conflictingness of a modular system, compositional verification iteratively alternates between performing conflict-preserving abstractions and composing strategically chosen modules until there is only one module left, whose non-blockingness is essentially the non-conflictingness of the original modular system. Although this framework has been intensively studied in recent years, the available results do not consider prioritised events, which is not only essential for SBD models, but also useful in other general application scenarios. In this context, we have extended and modified existing abstraction rules in the current chapter to adapt the semantic restriction imposed by prioritised events. Afterwards, the new abstraction rules have applied to various practical examples, where the entire state space as well as the time need for verification have been successfully reduced.

4 Sequential function chart

In industrial manufacturing, a great number of logic control programmes are implemented in *programmable logical controllers (PLCs)*, which is a type of computer specialised in reliable operation in industrial environments. In particular, the IEC $61131 - 3$ standard defines five programming languages for control programmes in PLCs, among which the *Sequential Function Chart (SFC)* is specifically of our interest due to its similarity to SBD. Basically, both SFC and SBD are structured based on Petri-nets. In addition, various concepts in SBDs, e.g. hyper-edges, conditions, writable variables, have corresponding similar counterparts in SFCs as well. These observations naturally raise the question of whether the compositional verification procedure developed so far in Chapter 3 can be applied to verify modular SFC programmes as well.

Similar to Chapter 2, we first consider the problem of formalizing SFC semantics. This has been addressed in various articles (Bauer, Engell et al., 2004; Bauer, Huuck et al., 2004; Blech and Ould Biha, 2011; Stursberg et al., 2005) since the original IEC $61131 - 3$ standard does not sufficiently formalise SFC semantics. Generally, as a programming language designed for a specific platform, the formal semantics of SFCs strongly rely on the operation rules of PLCs. By cyclically reading input from the plant, manipulating actions, firing enabled SFC transitions and bringing values to outputs, SFC dynamics is *cycle-triggered* (Stursberg et al., 2005) and runs over the physical time axis. However, the compositional verification method introduced in Chapter 3 is based on finite automata which suits event-based models. To minimise this gap, we attempt to interpret the cycle-triggered SFC semantics over the dense logic time axis, which is the first major challenge to handle in the current chapter. Technically, this is achieved by taking several reasonable assumptions, from which explicit enumeration of PLC cycles can be avoided while critical logical structures of SFCs are still preserved.

With SFC semantics over the dense time axis, a modular SFC programme can be translated into a collection of synchronised finite automata. However, the compositional verification approach introduced in Chapter 3 turns out to be not directly applicable due to issues with the priority assignment. Recall that in each PLC cycle, a PLC always first manipulates all executable actions and then fire all enabled transitions afterwards. By following the idea in Section 2.2, we consider each action execution and SFC transition as an event. Besides, action execution events are with higher priorities than transition events. However, this approach is problematic if multiple SFC transitions, say

Trans1 and Trans2, are to fire in the same PLC cycle. Recall from Section 2.2 that firing multiple enabled transitions is modelled as alternative interleaving sequences in finite automata. In this regard, if e.g. Trans1 is fired first, some subsequent action execution events (which are in the subsequent PLC cycle) may preempt Trans2 (which is still to fire in the current PLC cycle). To prevent any event from happening before all enabled SFC transitions are fired, we introduce the *unification operator* $\mathcal{U}(\cdot)$ on an automaton which unifies alternative interleaving sequences into a single simultaneous transition. In this context, the global monolithic behaviour is not solely restricted by $\mathcal{S}(\cdot)$, but by $\mathcal{S}(\mathcal{U}(\cdot))$. Nevertheless, the introduction of the unification operator also indicates that the compositional verification procedure introduced in Chapter 3 needs careful revision. Fortunately, all results in Chapter 3 are either directly applicable or only need subtle changes.

The current chapter is organised as follows: in Section 4.1, we clarify SFC semantics over the dense time axis, based on which the closed-loop behaviour of a system controlled by a modular SFC programme can be translated into finite automata utilising the translation procedure for SBDs. At the end of Section 4.1, we formally introduce the unification operator, which is necessary to illustrate the global closed-loop behaviour. In Section 4.2, we revisit the results in Chapter 3 to show that the compositional verification method is applicable for modular SFC programmes as well, where the global behaviour is restricted by $\mathcal{S}(\mathcal{U}(\cdot))$ instead of $\mathcal{S}(\cdot)$. The current chapter is closed by a case study in Section 4.3 with some concluding remarks.

4.1 Correlating SFCs with SBDs

To apply compositional non-blockingness verification to modular SFC programmes, in this section, we exploit the SBD semantics introduced in Chapter 2 to interpret SFC semantics, which lays the foundation of translating SFCs into finite automata. In particular, through syntax mapping from SFCs to SBDs and comparing the subtle differences between their semantics, SFC translation can be handled by modifying and extending the translation procedure for SBD introduced in Section 2.2. This procedure is represented in detail in the following.

4.1.1 Syntax mapping from SFCs to SBDs

Generally, SFCs are syntactically extended from Petri-nets where firing transitions are guarded by specified conditions and each place, a.k.a. *step* in an

SFC, specifies a sequence of actions to execute. Thus, to semantically interpret an SFC as an SBD, we first briefly introduce the syntactic elements in SFC and introduce their intended semantics by mapping them into SBD components. Before starting, we shall first mention that hierarchical structures similar to *invocation* in SBDs are not defined for SFCs. Although the terminology of "hierarchical SFC" is utilised in some contexts, e.g. in (Bauer, Huuck et al., 2004), this is in fact more closely related to *activation*, i.e. an SFC step can activate another programme, which may again be an SFC. The difference between invocation and activation is that the step which activates another SFC simply proceeds its operation or token propagation without waiting for the termination of the activated programme. This is beyond the scope of the current thesis and we only consider modular SFCs in the following.

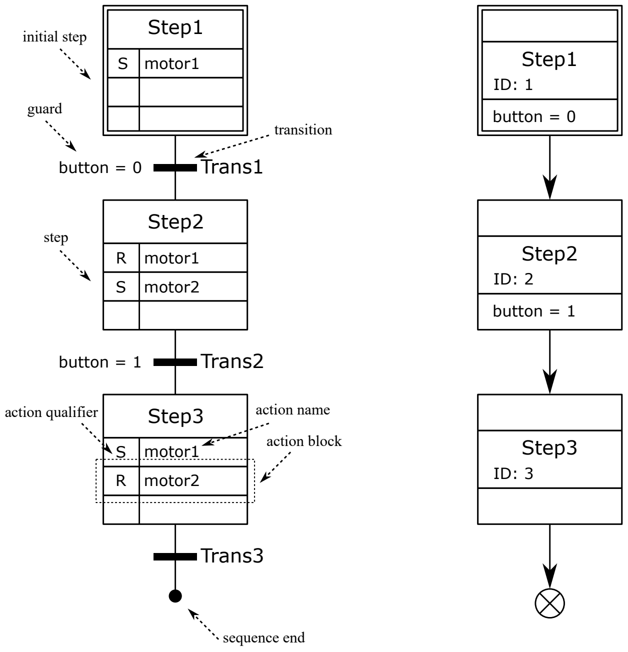

Figure 46: Syntactic interpretation of an SFC (left) as an SBD (right)

In Figure 46, a compact example SFC is given on the left side where most basic SFC elements are illustrated. The SBD resulting from the syntax mapping is demonstrated on the right side of Figure 46. By taking the convention that tokens generally propagate from top to bottom in an SFC, arrows are omitted in SFCs. In the following, we briefly introduce the syntax mapping of each component demonstrated in Figure 46.

Steps and transitions Basically, an SFC consists of alternatively connected *steps* and *transitions*, which are synonymous to places and transitions from a Petri-net perspective. Thus, it is natural to map an SFC step into an SBD process. Besides, each SFC transition corresponds to an SBD hyper-edge. For the situation in Figure 46, each SFC transition is directly mapped into a single SBD edge since no branching structures are present.

Initial step Each SFC can define a unique *initial step*, in which a token is placed when the SFC programme is activated. We can indeed match an SFC initial step analogously into an SBD *initial process*, which is a dedicated SBD process owning a token upon activation and can have zero or one predecessor. In Figure 46, the initial step Step1 of the SFC is mapped into an initial process Step1 with ID = 1 in the SBD, which, similar to an initial step, is denoted with a doubled contour.

Sequence ends Some SFC derivatives also suggest terminals for SFCs which eliminate tokens, e.g. *sequence end* as in GRAPH from Siemens TIA. Such elements can be directly mapped into SBD terminal nodes.

Guards For each SFC transition, a *guard* is specified (which can be trivially true) so that the transition can fire only if its guard evaluates true. An SFC guard can either be mapped to the precondition or the postcondition of an SBD process, since they both guards the firing of hyper-edges. Technically, we choose to map an SFC guard into the postcondition of the preceding SBD process, which benefits handling merged flows, as we will see in the following. As for the case in Figure 46, the guard of the transition Trans1 is mapped into the postcondition of the SBD process Step1 in the resulting SBD.

Actions For each SFC step, a list of *actions* this step executes is specified. Each action is given in the form of an *action block* which typically consists of an *action qualifier* and an *action name*. The action name of an action block specifies the variable which should be manipulated, typically an output bit or a memory bit. On the other hand, the action qualifier specifies the type of the action. For simplicity, we assume that an action name always refers to a binary variable, while only the two most basic qualifiers are considered, namely S for "setting a bit to 1" and R for "resetting a bit to 0". Basically, actions associated with a step are executed from top to bottom. This can be reflected in an SBD by assigning immediate instructions introduced in Section 2.3.3. As for the SBD in Figure 46, a possible assignment is

$$\mathsf{immediate}(1, \mathsf{motor1}, 1) = 1;$$

$$\text{immediate}(2, \text{motor1}, 0) = 1;$$
$$\text{immediate}(2, \text{motor2}, 1) = 2;$$
$$\text{immediate}(3, \text{motor1}, 1) = 1;$$
$$\text{immediate}(3, \text{motor2}, 0) = 2.$$

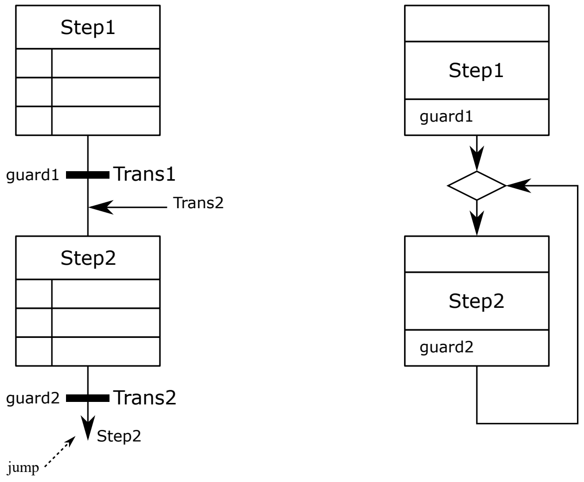

Figure 47: Mapping a jump with a preceding transition

We now consider further essential SFC components SFC which are not involved in Figure 46.

Sequence loops To realise cyclic executions, SFC supports looping structures which is the only situation that a token can be propagated from bottom to top. This is also referred to as a *jump* in GRAPH, as we can specify a transition to jump to an arbitrary target step. If the target step does not have any preceding transition (i.e. an initial step), such a jump can directly be mapped to an SBD edge; otherwise, a merge is necessary before the SBD process mapped from the target step; see Figure 47.

Divergences and convergences SFC divergences and convergences are synonymous to branches and merges in SBDs, respectively. Note that for an SFC divergence, guards of all successive transitions are mapped to the corresponding branch condition (instead of postcondition of some preceding process) in the resulting SBD; see Figure 48.

Simultaneous divergences and simultaneous convergences SFC simultaneous divergences and simultaneous convergences are synonymous to forks

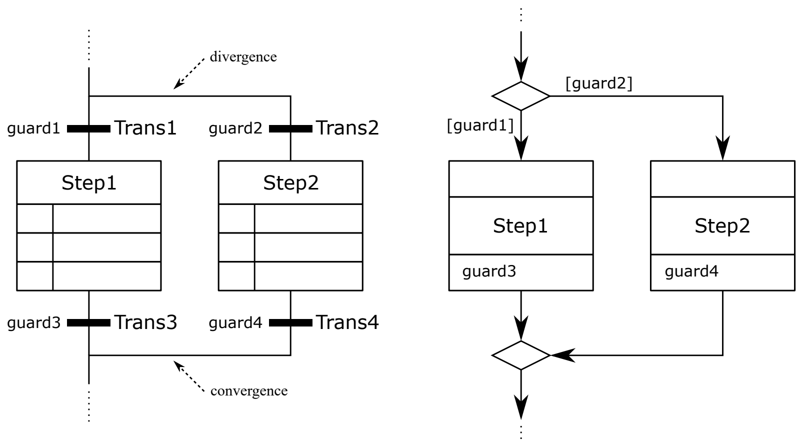

Figure 48: Mapping a divergence and a convergence

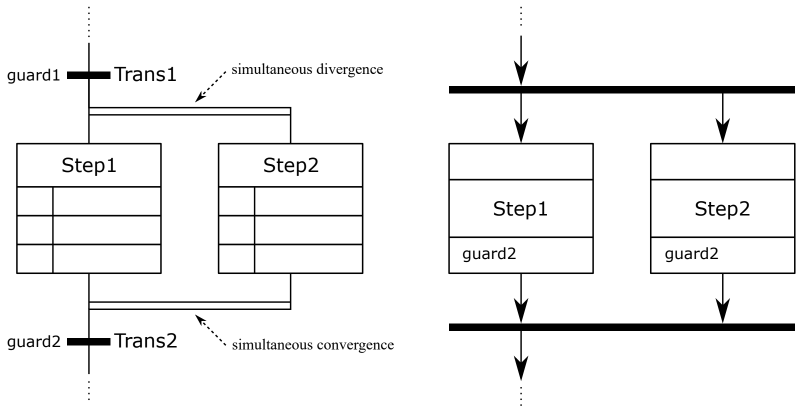

Figure 49: Mapping a simultaneous divergence and a simultaneous convergence

and joins in SBDs, respectively. Note that for a simultaneous convergence, if only one successive transition exists, its guard is copied to the postcondition of all preceding SBD processes; see Figure 49. Otherwise, the simultaneous convergence must be followed directly by a divergence, in which situation we shall map successive guards into SBD branch conditions.

4.1.2 Dense-time SFC semantics

Based on the syntax mapping, we attempt to translate a given modular SFC programme into finite automata by properly extending and modifying the translation procedure for SBDs introduced in Section 2.2. This requires a careful discussion of the subtle semantic differences between SFCs and SBDs. As clarified in Section 2.1.2, SBD semantics is based on the discrete dense time axis $\mathbb{N}_0 \times \mathbb{N}_0$ (which can also be simplified to the one-dimensional logic time axis $\mathbb{N}_0$). Reaction upon the occurrence of any input event is assumed instantaneous, which leads to the following two relevant effects:

(F1) Token propagation (i.e. firing hyper-edges) must happen immediately as soon as it becomes possible;

(F2) Token propagation is triggered by events.

In comparison with SBDs, SFCs are defined specifically for PLCs which is operated over the physical time axis. In particular, the operation of PLCs follow the so-called *PLC-cycle* which must last for a positive duration of physical time.[1] In each PLC-cycle, the following four procedures are sequentially executed:

(C1) Read values of PLC inputs;

(C2) Execute all specified actions in all active steps;

(C3) Figure out the set of enabled SFC transitions[2] and fire these transitions;

(C4) Set values to PLC outputs.

Note that the above cycle applies to a family of modular SFCs as well, i.e. for multiple SFCs running in parallel, a PLC cycle will first execute actions in all SFCs, then fire enabled transitions in all SFCs.

Consider the situation illustrated in Figure 50, where we focus on the system behaviour over PLC cycles based on the physical time axis. In particular, the topmost axis includes a complete PLC cycle lasting from ι_0 to ι_5. Ideally, the value of $\iota_5 - \iota_0$ should be as low as possible in a high-performance PLC, but can never be reduced to zero. Most prominently, after all input values have been read, the PLC becomes somewhat "blind" until the beginning of the next PLC-cycle. Consider the value changes of three binary sensors as illustrated in Figure 50, where we interpret each positive or negative edge in input bits as an event. This indicates that four events in_1, in_2, in_3 and in_4 sequentially happen in the same PLC cycle. In this case, although the occurrence of in_1 at ι_1 may enable some transitions, such transitions cannot be fired at ι_1 since the input

[1] A similar issue exists when comparing SFC with Grafcet defined in the IEC 60848 standard as well; see also (Provost, J.-M. Roussel et al., 2011).

[2] An SFC transition is enabled if all its preceding steps are currently active and its guard evaluates true

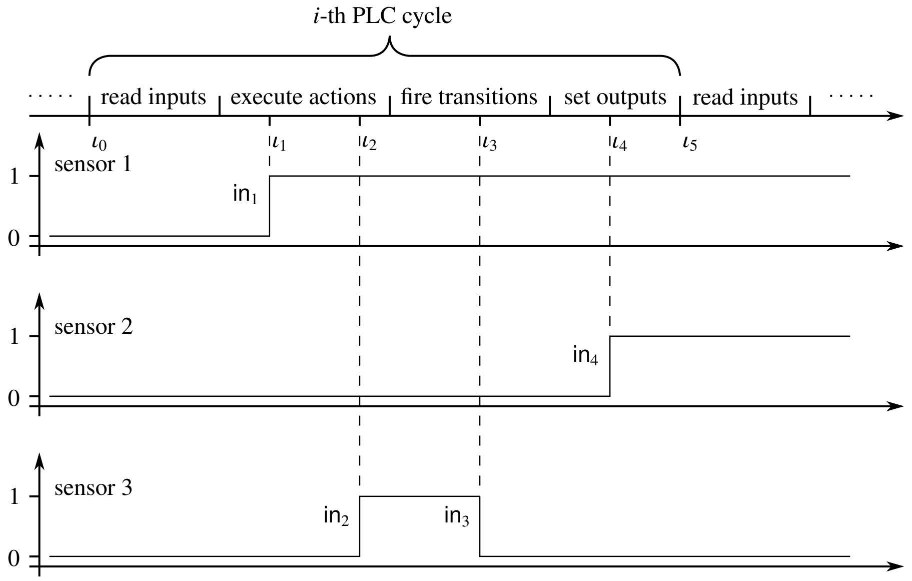

Figure 50: PLC cycles on the physical time axis

value change of sensor 1 can only be detected after ι_5, i.e. the start of the next PLC-cycle. In addition, the input event in_4, which occurs in the current PLC cycle as well, may also disable such transitions. Moreover, for sensor 3, the positive edge in_2 and the negative edge in_3 occur in the same PLC cycle. In this situation, since the bit value of sensor 3 appears the same at the beginning and end of this PLC cycle, both events in_2 and in_3 will be missed by the PLC.

In order to adopt SBD semantics as a framework to formalise SFC semantics, we recall the two-dimensional dense time axis $\mathbb{N}_0 \times \mathbb{N}_0$ from Section 2.1.2, where we utilised the horizontal axis $\mathbb{N}_0$ to represent the progress in physical time while the vertical axis $\mathbb{N}_0$ enables finitely stacking ordered sequences of events that occur at the same physical time instance. To abstractly describe the behaviour of an SFC over the dense time axis, we impose the following two assumptions considering the horizontal progress on the dense time axis:

(A1) PLC operations are instantaneous, i.e. upon detecting any input event, procedures (C2) – (C4) all take place at the same physical time instance;

(A2) In each PLC cycle, at most one input event can happen.

By assuming that PLCs react sufficiently rapidly upon the occurrence of any sensor event, assumption (A1) enables stacking (finitely many) PLC cycles vertically on the dense time axis. This is a reasonable simplification since the

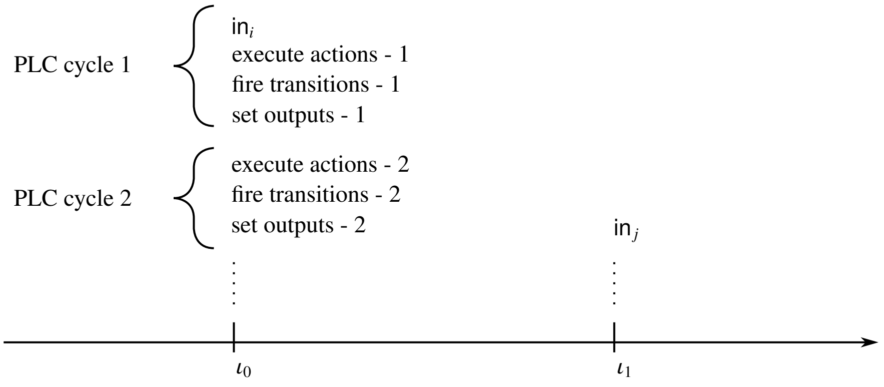

Figure 51: PLC cycles on the dense time axis

actual duration of a PLC cycle is generally available a-priori. This information can be utilised to validate that the additional delay of action manipulation will not affect the operation of the physical plant. For relatively simple plants, assumption (A2) is proposed from a similar motivation: since input events can generally be seen as a physical consequence of some preceding output edges, a minimum positive duration in between is conceivable (which can be checked a-priori as well). This can be further compared with the duration of a PLC cycle to validate whether (A2) is fulfilled. Indeed, for some large-scale systems, this assumption is somewhat vulnerable if multiple independent input events await. This situation is beyond the scope of the current dissertation.

We are now in the position to take a deeper look into vertical event stacks on the dense time axis, where we specifically mentioned that only a *finite* stack is allowed. Figure 51 illustrates the behaviour of some closed-loop systems controlled by SFCs with the dense-time interpretation, where assumptions (A1) and (A2) are already taken into consideration. As for (A2), the two input events in_i and in_j occur at different physical time instances ι_0 and ι_1. Moreover, upon the occurrence of in_i, internal actions involved in possibly multiple PLC cycles are vertically stacked below in_i. However, for the vertical line-up, an infinite number of cycles need to be stacked from (A1). Technically, a infinite stack can be avoided by requiring SFCs to always attain *stable states* after a finite count of transitions, i.e. states in which only input events cause changes in token configurations and/or variable evaluations. This can be guaranteed by assuming the following:

(A3) Given the set of active steps in an SFC and the corresponding variable evaluation, executing the actions in active steps always results in the same subsequent variable evaluation;

(A4) There is a finite upper bound of the number of SFC transitions that can be fired without any changes of input variables.

Assumption (A3) effectively allows us to avoid repetition of action executions in a stable state, i.e. when in a stable state, repeating action executions in subsequent PLC cycles does not change the evaluation of variables (unless some input event occurs) and thus can be ignored. This is a commonly assumed prerequisite for logic control programmes, which is comparable with *determinism* in (Bauer, Huuck et al., 2004). Furthermore, together with assumption (A4), only a finite number of transitions and action executions will be stacked on any physical time instance in the dense time axis. In particular, at the end of each stack, a stable state is reached.

Remark 4.1.1. *Typically, a PLC has a set of specific internal memory locations which are associated with the output bits. Within one PLC cycle, different values may be assigned to the memory multiple times while only the final value will be actually brought to the output bit. Nevertheless, this feature is irrelevant to non-blockingness verification of the closed-loop behaviour, thus not explicitly reflected in our model.*

Remark 4.1.2. *For a more detailed SFC semantics, one may be interested in the order of action execution (if multiple steps are currently active) and transition firing within each PLC cycle. Since the action execution order is not addressed by the original IEC* 61131*, we envisage that actions are executed in a shuffled order (which preserves the "local order" of each step), i.e. we consider all possible resulting variable evaluations. From this perspective, all events corresponding to action executions can be assigned with the same priority. As for diverging transitions, IEC* 61131 *stipulated that transitions sharing a same preceding step shall have different priorities, i.e. if multiple diverging transitions are enabled, the PLC shall deterministically choose one of them to fire. This feature can easily be reflected, as we will see below in Section 4.1.3.*

4.1.3 Translating SFCs into automata

With the dense-time SFC semantics as well as the syntax mapping, we are in the position to translate SFCs into finite automata. By exploiting the translation procedure for SBDs introduced in Section 2.2 and 2.3, a modular SFC programme can be translated into synchronised finite automata where several modifications and extensions due to the semantic features carried by PLC cycles are necessary. In particular, since SFC steps do *not* have process states as in SBDs, translation procedures concerning process states are generally discarded.

We recall from Figure 11 that the local closed-loop behaviour of an SBD is constructed by synchronising (through synchronous composition) a couple of automata. In the following, we concisely introduce the construction of the automata to synchronise when translating an SBD mapped from an SFC:

Reachability automaton extended with controlled variables The construction of a reachability automaton remains unchanged as introduced in Section 2.2.1. In particular, since the concept of SBD process states is dropped for SFCs, the extension regarding termination flags as mentioned in Remark 2.2.2 is ignored. Nevertheless, we still need the concept of controlled variables, i.e. in each state, self-loops of actions specified in active steps are appended.

Constraint automata The construction of condition automata remains unchanged as introduced in Section 2.2.2 by interpreting each input bit, memory bit and output bit as a binary variable. Particularly, the modified construction of variable automata suggested in Figure 19 is adopted as well. Since process states are not considered, the concept of termination condition is dropped in the translation and process state automata are not constructed either.

Immediate instructions As proposed in Section 4.1.1, actions assigned to a step are mapped into immediate instructions of an SBD process. The representation of immediate instructions mapped from an SFC is slightly simplified from (84) in Section 2.3.3, i.e. for each SBD process n, we generate

$$(\,\Sigma_{\mathsf{prio}}^{*} \cdot \Sigma_{n}^{\mathsf{TARGET}} \cdot P_n \cdot \Sigma_{n}^{\mathsf{SOURCE}}\,)^{*}. \qquad (151)$$

Two details in (151) are worth noting:

(i) Recall from (85) that $\Sigma_{\mathsf{prio}} := \{\sigma_{v,l}, \sigma_{v,l,n'} \mid v \text{ is utilised in } \mathsf{precond}(n)\} \cup \{\sigma_{v,l,n} \mid (v,l) \in \mathsf{CVariables}(n)\}$. Since syntax mapping never generates precondition for any process, it is equivalent to write

$$\Sigma_{\mathsf{prio}} := \{\sigma_{v,l,n} \mid (v,l) \in \mathsf{CVariables}(n)\}. \qquad (152)$$

(ii) Comparing with the original construction in (84), (151) removes the $\Sigma_{\mathsf{prio}}^{*}$ term after P_n. The reason for this adjustment is that action execution is explicit in SFCs, i.e. instead of “having the access to do so”, an SFC step “explicitly does so”. This adjustment guarantees that the guard evaluation afterwards indeed evaluates variables *after* the action sequence.

By constructing the above three types of automata as well as plant automata, we are in the position to review Section 2.2.3, i.e. the synchronous composition of all constructed automata is to perform. This results in an intermediate translation result with three classes of events, i.e. action events (resulting from output/memory manipulation) Σ_{ACT}, transition events (resulting from firing transitions) Σ_{TRANS} and sensor events (resulting from input bit edges) Σ_{SEN}. Furthermore, a subtle post-processing is necessary regarding the priority of diverging transitions (similar to branches in SBDs). Recall from Remark 4.1.2 that diverging transitions must have specified priority so that a deterministic choice among simultaneously enabled diverging transitions can always be taken. Since transition events are all private (as hierarchy is not considered), this feature can be conveniently reflected by removing lower-priority transition events in the composition, which is clearly legit following the intuition of Lemma 3.2.2.

Regarding the priority assignment, we recall that upon the occurrence of some sensor event, before a subsequent sensor event occurs, the current PLC cycle will first execute all necessary actions and then fire all enabled transitions. With this notion, we propose that

$$\mathsf{prio}(\sigma_{\mathsf{ACT}}) < \mathsf{prio}(\sigma_{\mathsf{TRANS}}) < \mathsf{prio}(\sigma_{\mathsf{SEN}}) \tag{153}$$

holds for any $\sigma_{\mathsf{ACT}} \in \Sigma_{\mathsf{ACT}}$, $\sigma_{\mathsf{TRANS}} \in \Sigma_{\mathsf{TRANS}}$ and $\sigma_{\mathsf{SEN}} \in \Sigma_{\mathsf{SEN}}$, respectively. In particular, from Remark 4.1.2, we globally let all action events to have the same priority. In addition, we propose that all transition events have the same priority as well. Note that transition divergence has already locally been resolved in the previous step. Thus, the global order of firing transitions is inessential for the system dynamics. By also letting all sensor events to have the same priority, at the current stage, the suggested priority assignment is well functional if at most one transition is enabled in each PLC cycle. However, if multiple transitions are enabled in one PLC cycle, critical errors may occur w.r.t. the higher priority of action events over transition events. This issue is discussed in detail in the following.

We now consider describing the global closed-loop behaviour by recalling Section 2.2.4. As an example, we translate the two SFCs depicted in Figure 52 into G_1 and G_2 as shown in Figure 53 with the translation procedure suggested so far. The value of the output bit motor, whose initial value is 0, is set to 1 and 0 by executing events on and off, respectively. Since none of the processes

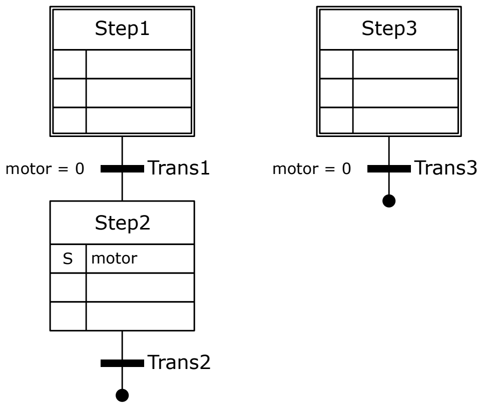

Figure 52: Two SFCs with simultaneously enabled transitions

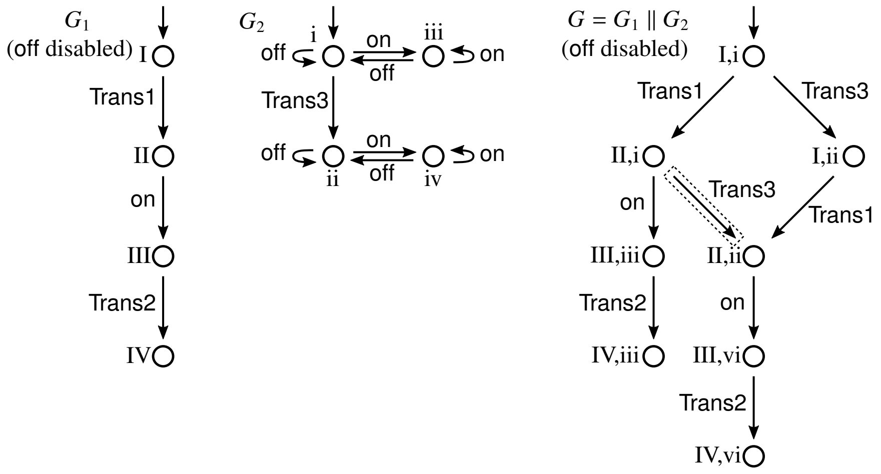

Figure 53: Intermediate translation results G_1 and G_2 (the transition $(\mathrm{II}, \mathrm{i}) \xrightarrow{\mathsf{Trans3}} (\mathrm{II}, \mathrm{ii})$ will be removed in $\mathcal{S}(G_1 \parallel G_2)$)

is specified with an action block R – motor, the event off is globally disabled. At the current stage, by letting $G = G_1 \parallel G_2$, we may expect that the global closed-loop behaviour complies with $\mathcal{S}(G)$, which contradicts the definition of PLC cycles. In the first PLC cycle, both transitions Trans1 and Trans2 are enabled and should be fired in the current PLC cycle, while in the next PLC cycle, on should be executed since Step2 has become active. However, if the transition $(\mathrm{I}, \mathrm{i}) \xrightarrow{\mathsf{Trans1}} (\mathrm{II}, \mathrm{i})$ is executed, the subsequent transition event, i.e.

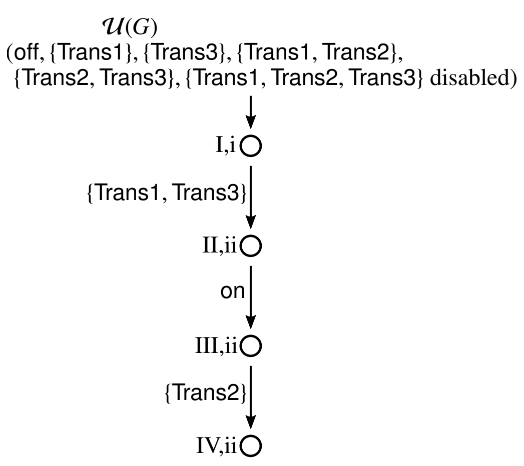
$\mathcal{U}(G)$
(off, {Trans1}, {Trans3}, {Trans1, Trans2}, {Trans2, Trans3}, {Trans1, Trans2, Trans3} disabled)
I,i
{Trans1, Trans3}
II,ii
on
III,ii
{Trans2}
IV,ii

Figure 54: Unifying transition labels Trans1 and Trans3 through the unification operator $\mathcal{U}(\cdot)$ (unreachable states are removed)

Trans3, will be preempted by the action event on according to the priority assignment suggested in (153). Note that by executing on, the value of motor is set to 1 which invalidates the guard of Trans1. This issue motivates us to *unify* the transition labels Trans1 and Trans3 into a single event $\{\mathsf{Trans1}, \mathsf{Trans3}\}$, by executing which state $(\mathrm{II}, \mathrm{ii})$ is directly reached; see Figure 54.

Technically, transition unification is achieved by applying the *unification operator* $\mathcal{U}(\cdot)$ which unifies the active transitions labelled by *unifiable events*. We first consider all SFC transitions as a set of *unifiable symbols* $\mathfrak{U}$. By recalling the notation of the event universe $\mathfrak{E}$, we propose that

$$\mathfrak{U} \cap \mathfrak{E} = \emptyset \tag{154}$$

shall hold. The set of all unifiable events $\Psi \subseteq \mathfrak{E} - \Upsilon$ is obtained through excluding the empty set from the power set of $\mathfrak{U}$, i.e.

$$\Psi = 2^{\mathfrak{U}} - \{\emptyset\}, \tag{155}$$

where we additionally require a specific priority value $\mathfrak{u} \in \mathbb{N}$ so that for any $\sigma \in \mathfrak{E} - \Upsilon$,

$$\sigma \in \Psi \quad \Leftrightarrow \quad \mathsf{prio}(\sigma) = \mathfrak{u}. \tag{156}$$

As for the $\Rightarrow$-part of the above requirement, all unifiable events shall have the same priority, which is reasonable since unifying any unifiable events should result in a new event with the same priority. For the $\Leftarrow$-part, no non-unifiable event shall be at priority $\mathfrak{u}$, which is an acceptable restriction from (153) in the context of SFC verification. In addition, we introduce the following notations for brevity:

- unifiable events within some alphabet Σ
  $\Sigma^{\mathsf{u}} := \Sigma \cap \Psi$;
- augmentation of an alphabet Σ through event unification
  $\mathsf{aug}(\Sigma) := \Sigma \cup \{\psi \in \Psi \mid \exists \Phi \subseteq \Sigma^{\mathsf{u}}.\, \psi = \cup_{\phi \in \Phi} \phi\}$;
- active unifiable event set in the state x of an automaton G
  $G^{\mathsf{u}}(x) := G(x) \cap \Psi$.

The set of unifiable symbols corresponds to the set of all SFC transitions. In comparison, we modify the set of transition events as such that $\Sigma_{\mathsf{TRANS}} \subseteq \Psi$ holds, i.e. each transition event corresponds to firing *a set of* SFC transitions. From this set-up, we recall the slight abuse of the unifiable symbols $\mathsf{Trans1}$ and $\mathsf{Trans2}$ as translation labels in Figure 53, which should have been replaced by singletons $\{\mathsf{Trans1}\}$ and $\{\mathsf{Trans2}\}$.

With the notion of unifiable events, we are now in the position to formally introduce the unification operator.

Definition 4.1.1. *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be an arbitrary automaton. The* unification operator *$\mathcal{U}(\cdot)$ is defined as such that $\mathcal{U}(G) := \langle Q, \mathsf{aug}(\Sigma), \rightarrow^{\mathcal{U}}, Q^\circ, M^{\mathcal{U}}\rangle$ where*

$$M^{\mathcal{U}} := \{\Sigma' \in 2^{\mathsf{aug}(\Sigma)} \mid \exists \Omega \in M.\, \mathsf{aug}(\Omega) = \Sigma'\} \tag{157}$$

and $x \overset{\alpha}{\rightarrow}{}^{\mathcal{U}}\, y$ if and only if either of the following statements holds:

(i) $\alpha \in A - \Psi$ *and* $x \overset{\alpha}{\rightarrow} y$, or

(ii) $G^{\mathsf{u}}(x) \neq \emptyset$, $\alpha = \cup_{\psi \in G^{\mathsf{u}}(x)} \psi$, $G^{\mathsf{u}}(y) \cap G^{\mathsf{u}}(x) = \emptyset$ *and* $x \xrightarrow{\psi_1 \psi_2 \cdots \psi_k} y$ *where* $\{\psi_1, \psi_2, \dots, \psi_k\} = G^{\mathsf{u}}(x)$.

In the unified automaton $\mathcal{U}(G)$, the alphabet is augmented as such that all possible unified transition labels are considered. The marking set is extended in the same fashion. This guarantees that if any $\psi \in \Sigma^{\mathsf{u}}$ appears in some $\Omega \in M$, then executing any unifiable event containing ψ is counted as executing ψ. Consider the following example.

Example 4.1.1. *Consider the automaton G given in Figure 52 again. The alphabet as well as the marking set of G are*

$$\Sigma = \{\, \mathsf{on}, \mathsf{off}, \{\mathsf{Trans1}\}, \{\mathsf{Trans2}\}, \{\mathsf{Trans3}\}\,\} \tag{158}$$

and

$$M = \{\, \{\mathsf{on}, \{\mathsf{Trans1}\}\},\\ \{\{\mathsf{Trans2}\}\}\,\}, \tag{159}$$

respectively. By applying the unification operator on G, the alphabet and the marking set of $\mathcal{U}(G)$ are

$$\begin{aligned}\mathsf{aug}(\Sigma) = \{\, &\mathsf{on}, \mathsf{off},\\ &\{\mathsf{Trans1}\}, \{\mathsf{Trans2}\}, \{\mathsf{Trans3}\},\\ &\{\mathsf{Trans1}, \mathsf{Trans2}\}, \{\mathsf{Trans1}, \mathsf{Trans3}\}, \{\mathsf{Trans2}, \mathsf{Trans3}\},\\ &\{\mathsf{Trans1}, \mathsf{Trans2}, \mathsf{Trans3}\}\,\}\end{aligned} \tag{160}$$

and

$$\begin{aligned}M^{\mathcal{U}} = \{\, &\{\mathsf{on}, \{\mathsf{Trans1}\}, \{\mathsf{Trans1}, \mathsf{Trans2}\}, \{\mathsf{Trans1}, \mathsf{Trans3}\},\\ &\quad \{\mathsf{Trans1}, \mathsf{Trans2}, \mathsf{Trans3}\}\},\\ &\{\{\mathsf{Trans2}\}, \{\mathsf{Trans1}, \mathsf{Trans2}\}, \{\mathsf{Trans2}, \mathsf{Trans3}\}, \{\mathsf{Trans1},\\ &\quad \mathsf{Trans2}, \mathsf{Trans3}\}\}\,\},\end{aligned} \tag{161}$$

respectively.

Remark 4.1.3. *Although the alphabet after unification is of exponential order, most of the unified events are unnecessary to be maintained in the alphabet if they only contain private unifiable events (which is indeed the case of SFC translations) and do not label any transition in the current automaton.*

As for the unified transition relation $\rightarrow^{\mathcal{U}}$, active unifiable events in each state (which correspond to all enabled SFC transitions in one PLC cycle) is unified into a single transition. Note that SFC transitions which potentially become enabled in the subsequent PLC cycle are not unified. Consider the following example.

Example 4.1.2. *Consider the automaton G given in Figure 55, which results from translating both SFCs in Figure 52 while ignoring all actions and guards. By Definition 4.1.1, we shall only unify both active unifiable events* $\{\mathsf{Trans1}\}$ *and* $\{\mathsf{Trans3}\}$ *in state* I*, while* $\{\mathsf{Trans2}\}$ *should be excluded since* Trans2 *can only be fired in the next PLC cycle. Note that there is no transition from* I *to* II *in $\mathcal{U}(G)$ since $G^{\mathrm{u}}(\mathrm{I}) \cap G^{\mathrm{u}}(\mathrm{II}) \neq \emptyset$. This indicates that* II *is unreachable in $\mathcal{U}(G)$.*

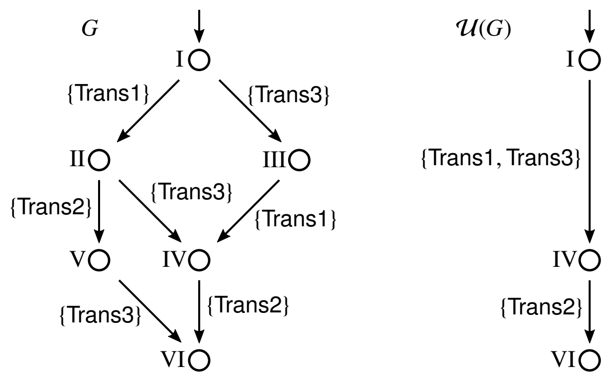

Figure 55: Unifying active unifiable events in one state (disabled events in $\mathcal{U}(G)$ are dropped)

With the unification operator, we are finally in the position to describe the global closed-loop behaviour of a system controlled by a modular SFC programme. For k SFCs which are correspondingly translated into $G_1, G_2, \dots, G_k$, the global closed-loop behaviour can be monolithically represented by

$$\mathcal{S}_{\mathcal{U}}(G_1 \parallel G_2 \parallel \cdots \parallel G_k) \tag{162}$$

where we concisely write $\mathcal{S}_{\mathcal{U}}(G) := \mathcal{S}(\mathcal{U}(G))$.

4.2 Compositional verification of modular SFC programmes

With the translation procedure clarified, the non-blockingness verification problem of modular SFC programmes is addressed in the current section by exploiting the compositional verification approach introduced in Chapter 3. A major technical change comparing with Chapter 3 is that, instead of $\mathcal{S}(\cdot)$, the global closed-loop behaviour of a modular system is now organised by the operator $\mathcal{S}_{\mathcal{U}}(\cdot)$, i.e. for a modular system whose global behaviour is represented by $\mathcal{S}_{\mathcal{U}}(G_1 \parallel G_2 \parallel \cdots \parallel G_k)$, we are interested in properly abstracting e.g. G_1 into G_1' so that

$$\begin{aligned}&\mathcal{S}_{\mathcal{U}}(G_1 \parallel G_2 \parallel \cdots \parallel G_k) \text{ is non-blocking}\\ &\qquad\qquad\Leftrightarrow \quad \mathcal{S}_{\mathcal{U}}(G_1' \parallel G_2 \parallel \cdots \parallel G_k) \text{ is non-blocking.}\end{aligned}$$

This subtle change in the problem statement first motivates us to adjust the definition of *non-conflictingness*, as given in Definition 3.1.5, into *non-$\mathcal{U}$-conflictingness*.

Definition 4.2.1 (adjusted from Definition 3.1.5). *A family $(G_i)_{1\leq i\leq k}$ of automata is* non-$\mathcal{U}$-conflicting *if and only if $\mathcal{S}_{\mathcal{U}}(G_1 \parallel G_2 \parallel \cdots \parallel G_k)$ is nonblocking.*

By revisiting Section 3.1.3, we first handle transition hiding through adjusting the definition of *hidable transition*, which was given in Definition 3.1.7, to adapt the definition of $\mathcal{U}$-conflictingness.

Definition 4.2.2 (adjusted from Definition 3.1.7). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ and $H = \langle Q_H, \Sigma_H, \rightarrow_H, Q^\circ_H, M_H\rangle$ be two automata. A transition $t \in \rightarrow_G$ in G is* $\mathcal{U}$-hidable w.r.t. *H if and only if*

$$G \text{ and } H \text{ are non-}\mathcal{U}\text{-conflicting} \quad \Leftrightarrow \quad G/_t \text{ and } H \text{ are non-}\mathcal{U}\text{-conflicting.} \tag{163}$$

With Definition 4.2.2, we figure out the set of $\mathcal{U}$-hidable transitions of a given automaton by revisiting Proposition 3.3.1 in the following. Particularly, we assert that transitions labelled by a unifiable event should be excluded from transition hiding, although, in the context of SFC verification, unifiable events must be private. The reason for this assertion is that a unifiable event potentially causes a *synchronous* step in a synchronous composition after unification, i.e. $\mathcal{U}(G \parallel H)$ for some automata G and H. More precisely, suppose no silent transitions exist in G and H, executing a private unifiable event in G also potentially causes H to change its state, while the key property of a silent transition is that executing a silent transition will *not* change the state of the rest part. Consider the situation in Figures 53 and 54 again. For the transition $\text{I} \xrightarrow{\mathsf{Trans1}} \text{II}$ in G_1, although it is labelled by a private event $\mathsf{Trans1}$ (w.r.t. G_2), it still results in a synchronous transition $(\text{I}, \text{i}) \xrightarrow{\{\mathsf{Trans1},\mathsf{Trans3}\}}_{\mathcal{U}} (\text{II}, \text{ii})$ in $\mathcal{U}(G_1 \parallel G_2)$ where a transition from G_2 is taken synchronously. Recall from (156) that only unifiable events are at priority $\mathfrak{u}$. Thus, we assume that, *in the scope of the current section,* the silent event $\tau_{(\mathfrak{u})}$ should never appear as transition label in any automaton.

Assumption 1. *For any automaton $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ in the current section, $x \overset{\alpha}{\rightarrow} y$ implies $\alpha \neq \tau_{(\mathfrak{u})}$.*

With the assumption above, we adapt Proposition 3.3.1 as follows.

Proposition 4.2.3 (adjusted from Proposition 3.3.1). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ be a Υ-shaped automaton and $H = \langle Q_H, \Sigma_H, \rightarrow_H, Q^\circ_H, M_H\rangle$ be an arbitrary automaton. Let $t = (\bar{x}_G, \sigma, \bar{y}_G) \in \rightarrow_G$ with $\sigma \in \Sigma_G - \Sigma_H - \Psi$ be* *such that for all $\Omega_G \in M_G$ so that $\sigma \in \Omega_G$, there exists $\sigma' \in \Omega_G - \Sigma_H - \Psi$ so that the following two statements hold:*

(i) $\mathsf{prio}(\sigma') \leq \mathsf{prio}(\sigma)$;

(ii) $\bar{y}_G \xRightarrow[\Delta:\sigma]{\epsilon} \bar{z}_G \xrightarrow[\Delta:\sigma']{\sigma'}$ *for some* $\bar{z}_G \in Q_G - \{\bar{x}_G\}$ *and* $\Delta = \Sigma(\bar{x}_G)$.

It holds that t is $\mathcal{U}$-hidable w.r.t. H.

Proof. By uniformly replacing each shaping operator $\mathcal{S}(\cdot)$ with the shaped unification operator $\mathcal{S}_{\mathcal{U}}(\cdot)$ and replacing each transition superscript $(\cdot)^{\mathcal{S}}$ with $(\cdot)^{\mathcal{S}_{\mathcal{U}}}$ (which denotes the existence of a transition in $\mathcal{S}_{\mathcal{U}}(G)$ for some automaton G), the proof of Proposition 3.3.1 applies to the current proposition. □

Based on transition hiding, we define the *$\mathcal{U}$-conflict equivalence* which is modified from Definition 3.1.9.

Definition 4.2.4 (adjusted from Definition 3.1.9). *Two automata G_1 and G_2 are $\mathcal{U}$-*conflict equivalent*, denoted $G_1 \simeq^{\mathcal{S}_{\mathcal{U}}} G_2$, if for any automaton T, it holds that*

$$G_1 \text{ and } T \text{ are non-}\mathcal{U}\text{-conflicting} \quad \Leftrightarrow \quad G_2 \text{ and } T \text{ are non-}\mathcal{U}\text{-conflicting}.$$

With the notion of $\mathcal{U}$-conflict equivalence, we say an abstraction of G, say G', is a *$\mathcal{U}$-conflict-preserving abstraction of G* if $G' \simeq^{\mathcal{S}_{\mathcal{U}}} G$. In the following, we explore whether the abstraction rules developed in Section 3.2 are all $\mathcal{U}$-conflict preserving and revisit the compositional verification procedure introduced in Section 3.3.

$\mathcal{U}$-conflict preserving abstraction rules

To review the abstraction rules introduced in Section 3.2, we first recall that all abstraction rules require that the automaton to abstract must be preprocessed by Υ-shaping, which itself is conflict-preserving. In order to apply $\mathcal{U}$-conflict-preserving abstractions on a Υ-shaped automaton, we shall first discuss whether Υ-shaping is $\mathcal{U}$-conflict-preserving. Obviously, this is indeed the case from Lemma 3.2.2.

Lemma 4.2.5 (adjusted from Lemma 3.2.2). *For any two automata G_1 and G_2, it holds that*

$$\mathcal{S}_{\mathcal{U}}(G_1 \parallel G_2) = \mathcal{S}_{\mathcal{U}}(\mathcal{S}_{\Upsilon}(G_1) \parallel G_2). \tag{164}$$

Based on Lemma 4.2.5, we are in the position to discuss whether the abstraction rules introduced in Section 3.2 are all $\mathcal{U}$-conflict-preserving. Fortunately, the result turns out to be positive and most relevant statements with their proofs only need straightforward uniform substitutions. Thus, an explicit review of the contents in Section 3.2 is moved to Appendix B.

Compositional verification

At the end of the current section, we briefly introduce the complete compositional verification procedure for modular SFC programmes by revisiting Algorithm 2. Generally, since all abstraction rules introduced in Section 3.2 are $\mathcal{U}$-conflict-preserving, Algorithm 2 can directly be utilised to check non-$\mathcal{U}$-conflictingness by only replacing $\textsc{IsNonBlocking}(\mathcal{S}(G))$ with $\textsc{IsNonBlocking}(\mathcal{S}_{\mathcal{U}}(G))$ in Line 19 (note that transition hiding is now performed by checking Proposition 4.2.3 instead of Proposition 3.3.1). Nevertheless, a conceivable improvement w.r.t. the unification operator can be applied due to the fact that for SFC translation results, unifiable events in all automata are private. Thus, similar to the $\mathcal{S}_{\Pi}(\cdot)$-operation in Algorithm 2, transition unification through $\mathcal{U}(\cdot)$ can be performed locally as well.

Lemma 4.2.6. *Let* $G_1 = \langle Q_1, \Sigma_1, \rightarrow_1, Q_1^\circ, M_1\rangle$ *and* $G_2 = \langle Q_2, \Sigma_2, \rightarrow_2, Q_2^\circ, M_2\rangle$ *be two automata so that* $\Sigma_1^{\mathrm{u}} \cap \Sigma_2^{\mathrm{u}} = \emptyset$. *It holds that*

$$\mathcal{S}_{\mathcal{U}}(G_1 \parallel G_2) = \mathcal{S}_{\mathcal{U}}(\mathcal{U}(G_1) \parallel G_2). \tag{165}$$

From Lemma 4.2.6, we can insert $G \leftarrow \mathcal{U}(G)$ and $H \leftarrow \mathcal{U}(H)$ after Lines 5 and 12 in Algorithm 2, respectively.

4.3 Case study

In this section, we envisage a use-case similar to that in Figure 44 where k conveyor belts (CB) and an exit slide (XS) are concatenated after an stack feeder (SF). Each CBi, $1 \leq i \leq k$ as well as the SF (i.e. CB0) is controlled by a single SFC; see Figure 56, where $\mathsf{WPS}i$ and $\mathsf{BM}i$ are input/output bits corresponding the workpiece sensor and the belt motor of CBi, respectively. XS (i.e. CB$k+1$) is not controlled by any SFC and is not equipped with a belt motor. Thus, for the SFC controlling CBk, the equality proposition $\mathsf{BM}k{+}1 = 0$ in the guard of $\mathsf{Trans}k\_3$ should be removed. Besides, similar to the "send event" in Figure 45, we utilise a memory bit $\mathsf{Send}i$ to synchronise the workpiece delivery from CBi to CB$i+1$. More precisely, CBi sets $\mathsf{Send}i$ (to 1) to start the delivery, while CB$i{+}1$ resets $\mathsf{Send}i$ (to 0) to terminate the delivery. We also associate each SFC

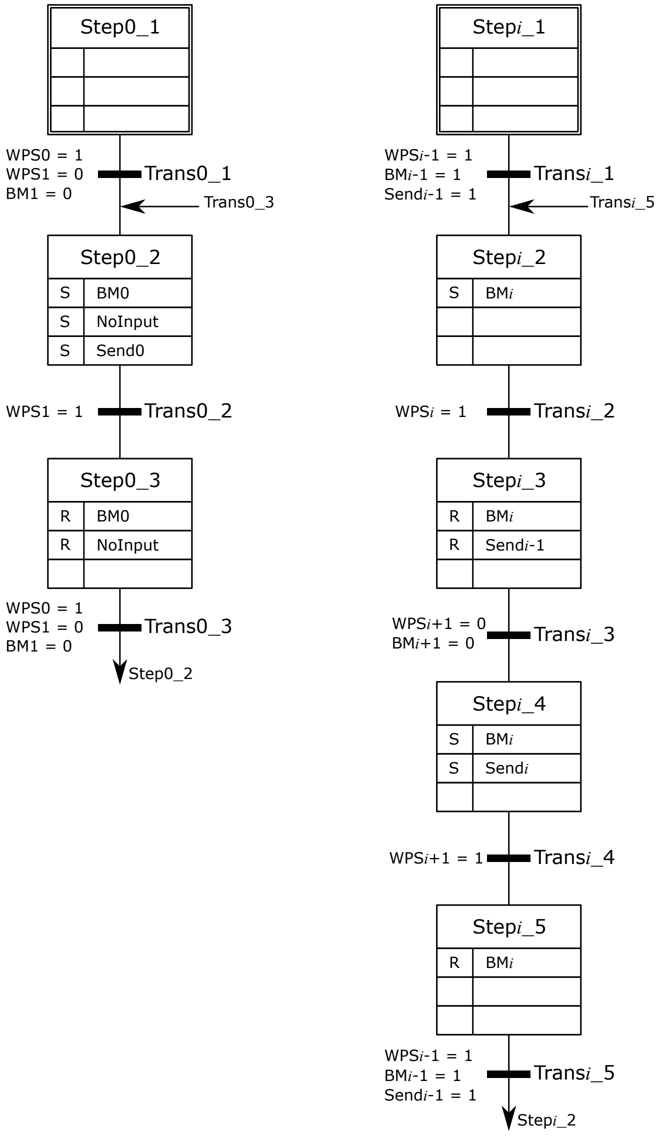

Figure 56: SFCs controlling CB0 (left) and CBi for $1 \leq i \leq k$ (right)

with a plant model. As for $1 \leq i \leq k$, the corresponding plant model for CBi is the synchronous composition of G_i and C_i given in Figure 45 where the event ar_i (or lv_i) corresponds to a positive (or negative) edge in the input bit WPSi while the event on_i (or off_i) sets (or resets) the output bit BMi, respectively. For CB0, we utilise an output bit NoInput to block the reception of a workpiece at CB0 when set to 1. This restriction is considered as a plant feature which

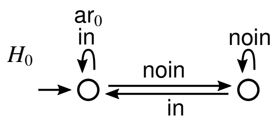

Figure 57: Workpiece input block when NoInput $= 0$

is represented by the automaton H_0 in Figure 57, where events noin and in denote the positive and negative edge of NoInput, respectively. Composing H_0 in Figure 57 with G_0 and C_0 in Figure 45 yields the plant model for CB0. Note that an explicit plant model for XS, i.e. CB$k+1$, is unnecessary, since XS only has a workpiece sensor whose behaviour is already included in C_k.

With $k+1$ SFCs, the global closed-loop behaviour is represented by $k+1$ automata $E_0, E_1, \dots, E_k$. To apply compositional verification as suggested in Section 4.2, we take the following conventions:

- All transition events in each E_i are renamed to the same event name, e.g. t_1 for all transition events in E_1. This is legit since all transition events are private and, as will be shown below, we do not put transition events into marking sets.
- Based on the event renaming above, the marking set of each E_i is set to

$$M_i = \{\{\mathsf{t}_i\}\} \tag{166}$$

for all $i \in \{0, 1, \dots, k\}$. In addition, the input order of the automata for the verification is

$$E_0, E_1, \dots, E_k. \tag{167}$$

- As for the function ConflictPreservingAbstraction in Algorithm 2, we only utilise PWB as the single abstraction rule and skip all other rules. From various tests of different rule combinations, the special structure of SFCs leads to only minor state reduction from other abstraction rules. In other words, the state reduction resulting from abstraction rules other than PWB does not pay off the cost of computing the abstraction.

Similar to Section 3.4.2, we apply compositional verification for closed-loop systems with different conveyor belt counts. The state count as well as the elapsed time for verification are listed in Table 7. It can be observed that compared with the monolithic construction of the entire system (where we still iteratively shape and unify w.r.t. local events), compositional verification does generally reduce the overall state space and the time needed for verification.

Table 7: State count and elapsed time (SFC verification)

k	mono. state cnt.	mono. time	abst. state cnt.	abst. time
5	4.8×10^3	9.3s	1.2×10^3	8.3s
6	1.3×10^4	29.9s	2.6×10^3	21.9s
7	3.6×10^4	92.3s	5.7×10^3	61.5s
8	9.6×10^4	309.9s	1.2×10^4	167.1s
9	2.5×10^5	857.1s	2.6×10^4	463.9s

However, drastic reduction as in Section 3.4.2 unfortunately does not apply to the case of SFC verification. The major reason is that transitions labelled by unifiable events, i.e. SFC transition events, are not hidable. This implicates that the synchronous composition of reachability automata of all SFCs is completely preserved in each iteration. Thus, the overall exponential growth of the total state count cannot be avoided.

Concluding remarks

In the current chapter, we have exploited SBD semantics and the compositional verification approach introduced in Chapters 2 and 3 to address the non-blockingness verification problem of modular SFC programmes. The physical-time based SFC semantics has been adapted onto the dense logic time axis, which has enabled us to represent SFCs as finite automata. Particularly, the semantic features carried by PLC cycles bring out the challenge that simply restricting the global behaviour by the shaping operator does not yield a faithful representation of the global behaviour. In this context, the unification operator has been introduced which solves this issue by unifying simultaneously enabled transition events into a single event. This again allows us to apply compositional verification to modular SFC programmes. However, comparing with former results in Section 3.4.2, the state reduction resulting from compositional verification is relatively mild for SFC verification. This is majorly caused by the fact that transition events are never hidable, since they potentially synchronise transition events from other modules even when they are private.

5 Conclusions and future prospects

In the current dissertation, we have formally addressed the non-blockingness verification problem of manufacturing systems represented by finite automata. Generally, when the system behaviour is represented monolithically by a single automaton, its non-blockingness can be checked by directly performing backward reachability search. However, when the system is represented by a family of synchronised automata, such an approach is normally infeasible since the state space of the monolithic representation of a modular system grows exponentially w.r.t. the count of modules. To mitigate this issue, in the current dissertation, we exploited the approach of compositional verification for the non-blockingness verification problem. The basic idea is to iteratively (i) perform conflict-preserving abstractions on each module and (ii) compose strategically chosen modules to form a subsystem. The iteration terminates when there is only one module left, which typically has fewer states compared with the monolithic representation of the original modular systems. In addition, since all applied abstractions are conflict-preserving, verifying the non-blockingness of the final module is equivalent to verifying the non-blockingness of the monolithic representation. In the current dissertation, we have attempted to apply the compositional verification approach to verify large-scale systems controlled by SBDs and SFCs. However, existing results w.r.t. compositional verification are not directly applicable, since the global behaviour in our use-cases are additionally restricted by event priorities and transition unifications. Thus, various modifications and extensions w.r.t. the framework of compositional verification as well as individual abstraction methods have been investigated and tested on different examples.

One major open research topic in the future is the automatic plant model generation. As we envisage the scenarios where engineers directly utilise either SBD or SFC to construct control programmes (which can be directly translated into automata), plant models should still be pre-designed by experts specialised in discrete event system modelling. To further automate the verification procedure, directly generating plant automata from some abstract model is of great practical value. One possible way to address this problem is to exploit the other two types of diagrams defined in IML, namely the Functional Structure (which organises the hierarchy of system functions and the hardware realising the functions) and the Interaction Structure (which describes the interaction between hardware components). In this regard, we expect that IML has the potential to enable fully automated closed-loop behaviour verification.

Bibliography

Aho, A., J. Hopcroft, J. Ullman (1974): *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company.

Akers, S. (1978): Binary Decision Diagrams. In: *IEEE Transactions on Computers* C-27.6, 509–516.

Alpern, B., F. Schneider (1985): Defining liveness. In: *Information Processing Letters* 21.4, 181–185.

– (1987): Recognizing Safety and Liveness. In: *Distributed Computing* 2, 117–126.

Baeten, J., J. Bergstra, J. Klop (1986): Syntax and defining equations for an interrupt mechanism in process algebra. In: *Fundamenta Informaticae* 9.2, 127–168.

Bauer, N., S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Pereira Remelhe, O. Stursberg (2004): Verification of PLC Programs Given as Sequential Function Charts. In: *Integration of Software Specification Techniques for Applications in Engineering* 3147, 517–540.

Bauer, N., R. Huuck, B. Lukoschus, S. Engell (2004): *A Unifying Semantics for Sequential Function Charts*. In: *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, 400–418.

Blech, J. O., S. Ould Biha (2011): *Verification of PLC Properties Based on Formal Semantics in Coq*. In: *Software Engineering and Formal Methods*, 58–73.

Blom, S., S. Orzan (2003): Distributed State Space Minimization. In: *Electronic Notes in Theoretical Computer Science* 80, 109–123.

Brandin, B., W. M. Wonham (1994): Supervisory control of timed discrete-event systems. In: *IEEE Transactions on Automatic Control* 39 (2), 329–342.

Brecher, C., M. Obdenbusch, D. Özdemir, J. Flender, A. R. Weber, L. Jordan, M. Witte (2016): Interdisciplinary Specification of Functional Structurres for Machine Design. In: *IEEE International Symposium on Systems Engineering (ISSE)*.

Brinksma, H., A. Rensink, W. Vogler (1995): *Fair Testing*. In: *CONCUR'95 Concurrency Theory*, 313–327.

Buzhinsky, I., V. Vyatkin (2017): Automatic Inference of Finite-State Plant Models From Traces and Temporal Properties. In: *IEEE Transactions on Industrial Informatics* 13.4, 1521–1530.

Cassandras, C. G., S. Lafortune (2008): *Introduction to Discrete Event Systems*. Second. Springer.

Clarke, E. M., O. Grumberg, D. A. Peled: (2001): *Model Checking*. MIT Press.

Clarke, E., D. Long, K. McMillan (1989): *Compositional model checking*. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, 353–362.

Cleaveland, R., G. Lüttgen, V. Natarajan (2007): Priority and abstraction in process algebra. In: *Information and Computation* 205.9, 1426–1458.

Daniele, M., F. Giunchiglia, M. Y. Vardi (1999): *Improved Automata Generation for Linear Temporal Logic*. In: *Computer Aided Verification*, 249–260.

Daw, Z., R. Cleaveland (2015a): *An Extensible Operational Semantics for UML Activity Diagrams*. In: *Software Engineering and Formal Methods*, 360–368.

De Giacomo, G., M. Vardi (2013): Linear temporal logic and Linear Dynamic Logic on finite traces. In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, 854–860.

Dijkstra, E. (1971): Hierarchical ordering of sequential processes. In: *Acta Informatica* 1, 115–138.

Eker, J., J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong (2003): Taming heterogeneity - the Ptolemy approach. In: *Proceedings of the IEEE* 91.1, 127–144.

Eshuis, H. (2002): Semantics and verification of UML activity diagrams for workflow modelling. PhD thesis. University of Twente.

Eshuis, R., R. Wieringa (2003): Comparing Petri Net and Activity Diagram Variants for Workflow Modelling – A Quest for Reactive Petri Nets. In: *Petri Net Technology for Communication-Based Systems* 2472, 321–351.

Eshuis, R. (2006): Symbolic model checking of UML activity diagrams. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15, 1–38.

Fabian, M., A. Hellgren (1998): PLC-based implementation of supervisory control for discrete event systems. In: *Proceedings of the 37th IEEE Conference on Decision and Control* 3, 3305–3310.

Fanti, M. P., G. Iacobellis, G. Rotunno, W. Ukovich (2013): *A simulation based analysis of production scheduling in a steelmaking and continuous casting plant*. In: *2013 IEEE International Conference on Automation Science and Engineering (CASE)*, 150–155.

Fernandez, J.-C. (1989): An implementation of an efficient algorithm for bisimulation equivalence. In: *Science of Computer Programming* 13, 13–219.

Flender, J., S. Storms, W. Herfs, M. Witte (2019): *Model-based Engineering of modern Automation Structures with the Interdisciplinary Modeling Language (IML)*. In: *2019 IEEE International Systems Conference (SysCon)*, 1–8.

Flordal, H., R. Malik (2006): *Modular nonblocking verification using conflict equivalence*. In: *2006 8th International Workshop on Discrete Event Systems*, 100–106.

– (2009): Compositional Verification in Supervisory Control. In: *SIAM Journal on Control and Optimization* 48, 1914–1938.

Gerber, C., S. Preuße, H.-M. Hanisch (2010): *A complete framework for controller verification in manufacturing*. In: *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, 1–9.

Harel, D., A. Naamad (1996): The STATEMATE semantics of statecharts. In: *ACM Transactions on Software Engineering and Methodology* 5, 293–333.

Hart, P. E., N. J. Nilsson, B. Raphael (1968): A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, 100–107.

Herfs, W., J. Flender, S. Storms, M. Witte (2018): *Data-Consistent Toolchain for a Requirements-Based Specification with the Interdisciplinary Modeling Language (IML)*. In: *2018 IEEE 22nd International Conference on Intelligent Engineering Systems (INES)*, 219–224.

Hering de Queiroz, M., J. Cury, W. Wonham (2005): Multitasking Supervisory Control of Discrete-Event Systems. In: *Discrete Event Dynamic Systems* 15, 375–395.

Jarraya, Y., M. Debbabi, J. Bentahar (2009): *On the Meaning of SysML Activity Diagrams*. In: *2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 95–105.

Kimura, S., E. Clarke (1990): *A parallel algorithm for constructing binary decision diagrams*. In: *Proceedings., 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 220–223.

Köhler, H., U. Nickel, J. Niere, A. Zündorf (2000): *Integrating UML diagrams for production control systems*. In: *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, 241–251.

Leduc, R. (2002a): Hierarchical Interface-based Supervisory Control. PhD thesis. Department of Electrical and Computer Engineering, University of Toronto.

Lennartson, B., X. Liang, M. Noori-Hosseini (2020): *Efficient Temporal Logic Verification by Incremental Abstraction*. In: *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, 894–899.

Lima, L., A. Didier, M. Cornélio (2013): *A Formal Semantics for SysML Activity Diagrams*. In: *Formal Methods: Foundations and Applications*, 179–194.

Liu, Y., P. Irudayaraj, F. Zhou, R. J. Jiao, J. N. Goodman (2014): *SysML-based Model Driven Discrete-Event Simulation*. In: *Moving Integrated Product*

Development to Service Clouds in the Global Economy - Proceedings of the 21st ISPE Inc. International Conference on Concurrent Engineering. Vol. 1, 617–626.

Lüttgen, G. (1998): Pre-emptive Modeling of Concurrent and Distributed Systems. PhD thesis. Universität Passau.

Malik, R., D. Streader, S. Reeves (2004): *Fair Testing Revisited: A Process-Algebraic Characterisation of Conflicts*. In: *Automated Technology for Verification and Analysis*, 120–134.

Malik, R. (2015): *Advanced selfloop removal in compositional nonblocking verification of discrete event systems*. In: *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, 819–824.

Malik, R., R. Leduc (2008): *Generalised nonblocking*. In: *2008 9th International Workshop on Discrete Event Systems*, 340–345.

– (2013): Compositional Nonblocking Verification Using Generalized Nonblocking Abstractions. In: *IEEE Transactions on Automatic Control* 58.8, 1891–1903.

Malik, R., M. Teixeira (2016): *Modular supervisor synthesis for extended finite-state machines subject to controllability*. In: *2016 13th International Workshop on Discrete Event Systems*, 91–96.

Malik, R., S. Ware (2020): On the computation of counterexamples in compositional nonblocking verification. In: *Discrete Event Dynamic Systems* 30, 301–334.

Michon, J.-F., J.-M. Champarnaud (1998): *Automata and binary decision diagrams*. In: *International Workshop on Implementing Automata*. Springer, 178–182.

Milner, R. (1989): *Communication and Concurrency*. Prentice-Hall, Inc.

Mohajerani, S., S. Lafortune (2020): Transforming Opacity Verification to Nonblocking Verification in Modular Systems. In: *IEEE Transactions on Automatic Control* 65.4, 1739–1746.

Mohajerani, S., R. Malik, M. Fabian (2014): A Framework for Compositional Synthesis of Modular Nonblocking Supervisors. In: *IEEE Transactions on Automatic Control* 59.1, 150–162.

– (2016): A framework for compositional nonblocking verification of extended finite-state machines. In: *Discrete Event Dynamic Systems* 26, 33–84.

– (2017): Compositional synthesis of supervisors in the form of state machines and state maps. In: *Automatica* 76, 277–281.

Moor, T. (2022): CompileDES: Executable-Code Generation from Synchronised libFAUDES Automata. In: *https://fgdes.tf.fau.de/compiledes, Accessed: 27.06.2022*.

Moor, T., K. Schmidt, S. Perk (2008): *libFAUDES — An open source C++ library for discrete event systems*. In: *2008 9th International Workshop on Discrete Event Systems*, 125–130.

Natarajan, V., R. Cleaveland (1995): *Divergence and fair testing*. In: *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming*, 648–659.

Object Management Group (2017a): *OMG System Modeling Language*. An OMG Systems Modeling Language™ Publication.

– (2017b): *OMG Unified Modeling Language*. An OMG Unified Modeling Language™ Publication.

Paige, R., R. Tarjan (1987): Three Partition Refinement Algorithms. In: *SIAM Journal on Computing* 16, 973–989.

Pilbrow, C., R. Malik (2015): An algorithm for compositional nonblocking verification using special events. In: *Science of Computer Programming* 113, 119–148.

Preuße, S., H.-C. Lapp, H.-M. Hanisch (2012): *Closed-loop system modeling, validation, and verification*. In: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, 1–8.

Provost, J., J.-M. Roussel, J.-M. Faure (2011): *A formal semantics for Grafcet specifications*. In: *2011 IEEE International Conference on Automation Science and Engineering*, 488–494.

Qamsane, Y., T. Abdelouahed, A. Philippot (2016): A synthesis approach to distributed supervisory control design for manufacturing systems with Grafcet implementation. In: *International Journal of Production Research* 55, 1–21.

Ramadge, P., W. Wonham (1989): The control of discrete event systems. In: *Proceedings of the IEEE* 77.1, 81–98.

Ramadge, P., W. Wonham (1987): Supervisory control of a class of discrete event systems. In: *SIAM Journal on Control and Optimization* 25, 206–230.

Schmidt, K., M. H. de Queiroz, J. E. R. Cury (2007): *Hierarchical and decentralized multitasking control of discrete event systems*. In: *2007 46th IEEE Conference on Decision and Control*, 5936–5941.

Störrle, H. (2004): *Semantics of Control-Flow in UML 2.0 Activities*. In: *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, 235–242.

Stursberg, O., S. Lohmann, S. Engell (2005): Improving Dependability of Logic Controllers by Algorithmic Verification. In: *IFAC Proceedings Volumes* 38.1, 104–109.

Su, R., J. H. van Schuppen, J. E. Rooda, A. T. Hofkamp (2010): Nonconflict check by using sequential automaton abstractions based on weak observation equivalence. In: *Automatica* 46.6, 968–978.

Tabuada, P. (2009): *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer US.

Tang, Y., T. Moor (2024): Compositional non-blockingness verification of finite automata with prioritised events. In: *Discrete Event Dynamic Systems* 34, 1–37.

Tarjan, R. (1972a): Depth-First Search and Linear Graph Algorithms. In: *SIAM Journal on Computing* 1, 146–160.

Vardi, M. Y. (1996): *An automata-theoretic approach to linear temporal logic*. In: *Proceedings of the VIII Banff Higher Order Workshop Conference on Logics for Concurrency*, 238–266.

Ware, S., R. Malik (2012): Conflict-preserving abstraction of discrete event systems using annotated automata. In: *Discrete Event Dynamic Systems* 22, 451–477.

Ware, S., R. Malik (2013): Compositional verification of the generalized non-blocking property using abstraction and canonical automata. In: *International Journal of Foundations of Computer Science* 24, 1183–1208.

Own Publications

Tang, Y., T. Moor (2021): *Compositional Verification of Finite Automata under Event Preemption*. In: *2021 60th IEEE Conference on Decision and Control (CDC)*, 301–308.

– (2022): Compositional Verification of Non-Blockingness with Prioritised Events. In: *16th IFAC Workshop on Discrete Event Systems (WODES)* 55.28, 236–243.

– (2024): Compositional non-blockingness verification of finite automata with prioritised events. In: *Discrete Event Dynamic Systems* 34, 1–37.

Student Works

Li, Z. (2019): Fallstudie zur Spezifikation einer Automatisierungseinrichtung durch Sequential Behavior Diagrams. MA thesis. Lehrstuhl für Regelungstechnik, FAU Erlangen-Nürnberg.

Lu, Z. (2020): Effiziente Verifikation durch Komposition und Abstraktion. Research internship. Lehrstuhl für Regelungstechnik, FAU Erlangen-Nürnberg.

Appendix

A Plant models of the production line example

In this section, plant models utilised in the SBD example in Section 2.4 are introduced. Recall from Figure 23 that for the four modules $\mathrm{M1}-1$, $\mathrm{M1}-2$ M1 and M2, four automata G_{1-1}, G_{1-2}, G_1 and G_2 are respectively required as plant models to describe the global closed-loop behaviour. We first list all events utilised in the plant automata in Table 8, which are based on SBD variables introduced in Section 2.4 (internal variables are considered irrelevant to plant models). For each variable in Table 8, events and their corresponding target values are arranged in the same order, e.g. for the variable CB1_BM, cb1_off is the event which changes its value to 0.

Module M1-1

The plant behaviour of this module is represented by the synchronous composition of the three automata in Figure 58. In particular, G_{1-1_1} indicates that the positioning motor can only move between the south most and the north most position by turning on the motor. In addition, G_{1-1_2} is redundant in that a same copy will be generated while translating the corresponding SBD; see Section 2.2.2.

Module M1-2

The plant behaviour of this module is represented by the synchronous composition of the three automata in Figure 59, where G_{1-2_1} and G_{1-2_2} synonymously describe the behaviour of each conveyor belt. Besides, G_{1-2_3} describes the physical coupling between CB1 and CB2, which essentially specifies that CB2 can get a workpiece only if CB1 has sent one to it. In addition, G_{1-2_3} also illustrates that sending more than one workpiece from CB1 without CB2 having received one in between is considered illegal. This is indicated by a dedicated blocking state which can be reached by e.g. the illegal sequence cb1_lv · cb1_lv.

Module M1

This is a high-level module which coordinates $\mathrm{M1}-1$ and $\mathrm{M1}-2$. Thus, low-level behaviour is omitted in this module and only the button represents the plant behaviour, as being depicted in Figure 60.

Table 8: Variable list of the production line example with event names

variable	description	values	events
CB1_BM	belt motor	$\{\underline{0}, 1\}$	{cb1_off, cb1_on}
CB1_WPS	workpiece sensor	$\{\underline{0}, 1\}$	{cb1_lv, cb1_ar}
CB2_BM	belt motor	$\{\underline{0}, 1\}$	{cb2_off, cb2_on}
CB2_WPS	workpiece sensor	$\{\underline{0}, 1\}$	{cb2_lv, cb2_ar}
PM_PM	positioning motor (1 = to s., 0 = stop, -1 = to n.)	$\{-1, \underline{0}, 1\}$	{pm_p-, pm_p0, pm_p+}
PM_PS+	south position sensor	$\{\underline{0}, 1\}$	{pm_lv+, pm_ar+}
PM_PS-	north position sensor	$\{0, \underline{1}\}$	{pm_lv-, pm_ar-}
PM_MOP	processing machine	$\{\underline{0}, 1\}$	{pm_stp, pm_op}
PM_MRD	ready to start processing machine	$\{0,\underline{1}\}$	{pm_bs, pm_rd}
OP1	operation button	$\{\underline{0}, 1\}$	{op1_rl, op1_pr}
OP2	operation button	$\{\underline{0}, 1\}$	{op2_rl, op2_pr}
RB_BM	belt motor	$\{\underline{0}, 1\}$	{rb_off, rb_on}
RB_WPS	workpiece sensor	$\{\underline{0}, 1\}$	{rb_lv, rb_ar}
RB_RM	rotation motor (1 = cw., 0 = stop, -1 = ccw.)	$\{-1, \underline{0}, 1\}$	{rb_r-, rb_r0, rb_r+}
RB_SCW	orientation sensor, north-south position	$\{\underline{0}, 1\}$	{rb_lv+, rb_ar+}
RB_SCCW	orientation sensor, west-east position	$\{0, \underline{1}\}$	{rb_lv-, rb_ar-}
XS_WPS	workpiece sensor	$\{\underline{0}, 1\}$	{xs_lv, xs_ar}

Module M2

The plant behaviour of this module is represented by the synchronous composition of the six automata in Figure 61. The rotation of RB (described by G_{2_1}) is similar to the positioning motor in Module M$1-1$, i.e. RB can only rotate 90° between both orientations. As the behaviour represented by G_{2_2} and G_{2_3} is relatively clear, special care should be taken to G_{2_4} and G_{2_5}, which are intended to describe the coupling between RB and XS as well as CB2 and RB.

Similar to G_{1-2_3}, G_{2_4} specifies that XS can only receive a workpiece if RB has sent one, while sending a workpiece from RB is disallowed if XS has not received the former one yet. In addition, rb_lv shall not happen as well if RB is not in the west-east orientation; see Figure 20 in Section 2.4. To reject such undesired behaviour, we set a dedicated blocking state which can be reached by executing e.g. rb_lv— · rb_lv. Similarly, the sequence rb_lv · rb_lv— is undesired as well since RB shall not rotate if RB has sent a workpiece but XS has not received it yet. Besides, G_{2_5} is intended to specify the coupling between CB2 and RB.[1] There are two possible cases for RB to receive a workpiece, i.e. either from CB2 or from SF2. Both cases correspond to the two orientations of RB, respectively. Clearly, sending a workpiece from CB2 when RB is not in the west-east orientation is illegal and thus leads to the dedicated blocking state.

[1] Indeed, CB2 does not belong to module M2. Thus, as few events from CB2 as possible should be utilised in the plant model of M2 from a design perspective.

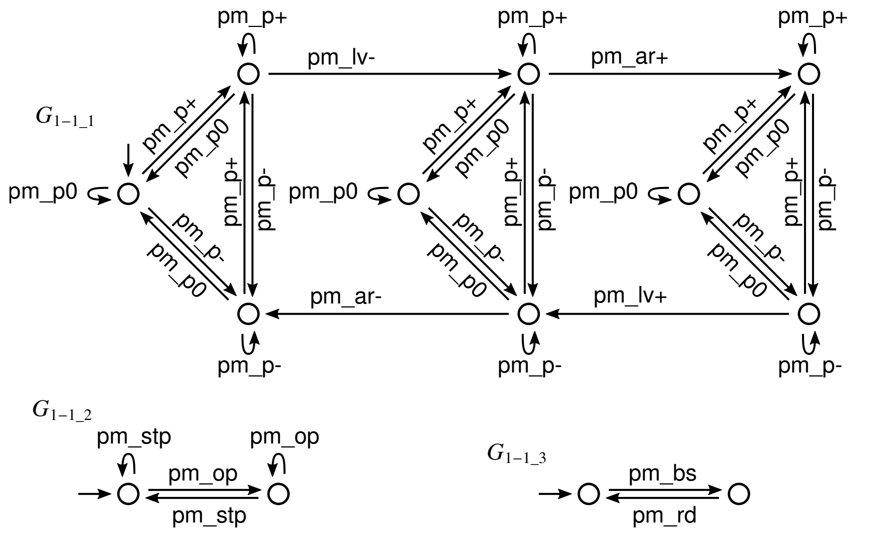

Figure 58: Automata representing the plant behaviour of M1 − 1

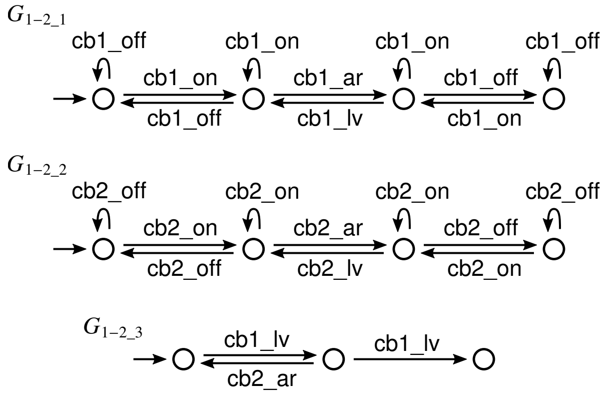

Figure 59: Automata representing the plant behaviour of M1 − 2

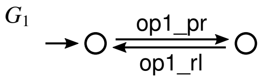

Figure 60: The plant behaviour of M1

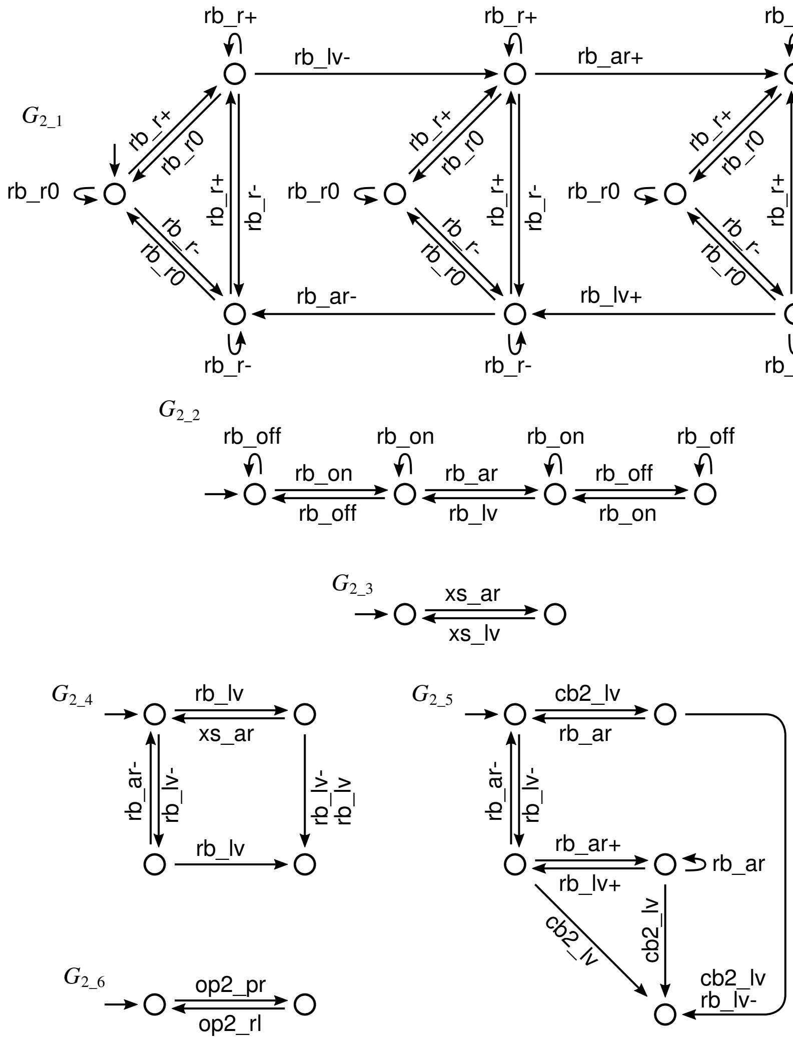

Figure 61: Automata representing the plant behaviour of M2

B $\mathcal{U}$-conflict-preserving abstraction rules

In this section, we explicitly review all abstraction rules introduced in Section 3.2 and show that, through straightforward substitutions in the corresponding statements, they can all be applied as $\mathcal{U}$-conflict-preserving abstraction rules. In particular, since the proofs of the relevant statements do not require major modifications, we avoid explicitly reformulating the proofs. Instead, we suggest the following uniform substitutions in proofs:

- Replace each $\mathcal{S}(\cdot)$ operator with the $\mathcal{S}_{\mathcal{U}}(\cdot)$ operator;
- Replace each transition superscript $(\cdot)^{\mathcal{S}}$ with $(\cdot)^{\mathcal{S}_{\mathcal{U}}}$, which denotes the existence of a transition in $\mathcal{S}_{\mathcal{U}}(G)$;
- Replace each $\Sigma_{T\setminus G}$ with $\mathsf{aug}(\Sigma_{T\setminus G})$ (since we only use the alphabet $\Sigma_{T\setminus G}$ when discussing synchronised behaviour);
- Replace each $\Sigma_G \cup \Sigma_T$ with $\mathsf{aug}(\Sigma_G \cup \Sigma_T)$;
- Replace each $M_G \cup M_T$ with $M^{\mathcal{U}} := \{E \in 2^{\mathsf{aug}(\Sigma)} \mid \exists \Omega \in M_G \cup M_T.\, \mathsf{aug}(\Omega) = E\}$.

In the remainder, when utilising terms like *the proof of Proposition A analogously applies to proving proposition B*, we refer to proving Proposition B by performing the aforementioned five uniform substitutions to the proof of Proposition A. Some proofs may need several additional substitutions, which will be explicitly mentioned. It is also worth mentioning that e.g. for automata G and T with individual alphabets Σ_G and Σ_T, we say a transition $(x_G, x_T) \xrightarrow{\alpha}^{\mathcal{U}} (y_G, y_T)$ in $\mathcal{U}(G \parallel T)$ is *driven by* G if $\alpha \in \mathsf{aug}(\Sigma_G \cup \Sigma_T) - \mathsf{aug}(\Sigma_{T\setminus G})$, i.e. if any event from Σ_G participates in this transition.

Prioritised weak bisimulation

We first review abstractions through constructing PW-bisimilar automata, including quotient automaton construction w.r.t. PWB and redundant silent loop removal. By adjusting Proposition 3.2.8 and Theorem 3.2.9, it turns out that two PW-bisimilar automata are also $\mathcal{U}$-conflict equivalent. We first adjust Proposition 3.2.8 as follows.

Proposition B.1 (adjusted from Proposition 3.2.8). *Let $G_1 = \langle Q_1, \Sigma_G, \rightarrow_1, Q_1^\circ, M_G\rangle$ and $G_2 = \langle Q_2, \Sigma_G, \rightarrow_2, Q_2^\circ, M_G\rangle$ be two Υ-shaped automaton so that $G_1 \approxeq G_2$. For any automaton $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q_T^\circ, M_T\rangle$, any transition $(x_1, x_T) \xrightarrow{\alpha}^{\mathcal{S}_{\mathcal{U}}} (y_1, y_T)$ in $\mathcal{S}_{\mathcal{U}}(G_1 \parallel T)$ and any $x_2 \in Q_2$ so that $x_1 \approxeq x_2$, there* *exists some* $y_2 \in Q_2$ *so that* $(x_2, x_T) \overset{\mathsf{p}(\alpha)}{\Longrightarrow}^{\mathcal{S}_\mathcal{U}} (y_2, y_T)$ *in* $\mathcal{S}_\mathcal{U}(G_2 \parallel T)$ *and* $y_1 \approxeq y_2$.

Proof. The proof of Proposition 3.2.8 applies analogously to the current proposition. Note that for the proof in Case 2, i.e. $(x_1, x_T) \xrightarrow{\alpha}^{\mathcal{S}_\mathcal{U}} (y_1, y_T)$ being not driven by G_1, one shall consider more carefully when $\alpha \in \Psi$. Note that $G^{<\alpha}_{1,\text{slnt}}(x_1) = \emptyset$. From the definition of the unification operator, we can also conclude that $G^{\mathsf{u}}_1(x_1) = \emptyset$. From the proof of Case 2 in Proposition 3.2.8, for all $x_2 \in Q_2$ so that $x_1 \approxeq x_2$, there exists some $y_2 \in Q_2$ so that $x_1 \approxeq y_2$ and

$$(x_2, x_T) \overset{\epsilon}{\Rightarrow} (y_2, x_T) \xrightarrow{\alpha} (y_2, y_T) \tag{168}$$

in $G_2 \parallel T$. Based on the proof of Case 2 in Proposition 3.2.8, it suffices to check whether $G^{\mathsf{u}}_2(y_2) = \emptyset$ holds, since if not, the transition $(y_2, x_T) \xrightarrow{\alpha}^{\mathcal{U}} (y_2, y_T)$ no longer exists in $\mathcal{U}(G_2 \parallel T)$. To prove by contradiction, we suppose that there exists some $\psi_2 \in G^{\mathsf{u}}_2(y_2)$. Since $x_1 \approxeq y_2$, from (P1), either $\psi_2 \in G^{\mathsf{u}}_1(x_1)$ or $G^{\leq\alpha}_{1,\text{slnt}}(x_1) \neq \emptyset$ holds. Note that the latter statement implies $G^{<\alpha}_{1,\text{slnt}}(x_1) \neq \emptyset$ from Assumption 1. However, from $(x_1, x_T) \xrightarrow{\alpha}^{\mathcal{S}_\mathcal{U}} (y_1, y_T)$, it can be easily implied that $G^{\mathsf{u}}_1(x_1) = G^{<\alpha}_{1,\text{slnt}} = \emptyset$, which contradicts both possibilities. □

With Proposition 3.2.8 adjusted as above, the applicability of PWB as a $\mathcal{U}$-conflict-preserving abstraction follows immediately.

Theorem B.2 (adjusted from Theorem 3.2.9). *Let* $G_1 = \langle Q_1, \Sigma_G, \rightarrow_1, Q^\circ_1, M_G\rangle$ *and* $G_2 = \langle Q_2, \Sigma_G, \rightarrow_2, Q^\circ_2, M_G\rangle$ *be two* Υ*-shaped automata so that* $G_1 \approxeq G_2$. *It holds that* $G_1 \simeq^{\mathcal{S}_\mathcal{U}} G_2$.

Proof. The proof of Theorem 3.2.9 applies analogously to the current theorem by replacing Proposition 3.2.8 with Proposition B.1. □

Redundant silent step rule

The redundant silent step rule can obviously be applied as a $\mathcal{U}$-conflict-preserving abstraction, since for a redundant silent step $x \xrightarrow{\tau} y$, no regular event is active in x at all. This indicates that for the proof of Proposition 3.2.19, there is no opportunity for a private transition in T to unify with a transition in G. With this observation, we adapt Propositions 3.2.19 and 3.2.20 as follows.

Proposition B.3 (adjusted from Proposition 3.2.19). *Let* $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ *be a* Υ*-shaped automaton and the equivalence* $\sim \subseteq Q_G \times Q_G$ *is* *induced by the redundant silent step* $\bar{x}_G \xrightarrow{\tau} \bar{x}'_G$*. Let* $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^\circ_T, M_T\rangle$ *be any automaton. For all* $\bar{x}_T \in Q_T$ *so that* $T^{\leq\tau}_{\mathrm{prvt}}(\bar{x}_T) \neq \emptyset$*,* $(\bar{x}_G, \bar{x}_T)$ *is not reachable in* $\mathcal{S}_{\mathcal{U}}(G \parallel T)$*.*

Proof. The proof of Proposition 3.2.19 applies analogously to the current proposition. □

Proposition B.4 (adjusted from Proposition 3.2.20)**.** *Let* $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ *be a* Υ*-shaped automaton and the equivalence* $\sim \subseteq Q_G \times Q_G$ *is induced by the redundant silent step* $\bar{x}_G \xrightarrow{\tau} \bar{x}'_G$*. Let* $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^\circ_T, M_T\rangle$ *be any automaton.*

(i) For any transition $([x_G], x_T) \xrightarrow{\alpha}^{\mathcal{S}_{\mathcal{U}}} ([y_G], y_T)$ *in* $\mathcal{S}_{\mathcal{U}}(G/{\sim} \parallel T)$*, at least one of the following two statements is true for any* $x'_G \in [x_G]$*:*

  a) There exists some $y'_G \in [y_G]$ *so that* $(x'_G, x_T) \overset{\mathsf{p}(\alpha)}{\Longrightarrow}^{\mathcal{S}_{\mathcal{U}}} (y'_G, y_T)$ *in* $\mathcal{S}_{\mathcal{U}}(G \parallel T)$*, or*

  b) (x'_G, x_T) *is not reachable in* $\mathcal{S}_{\mathcal{U}}(G \parallel T)$*.*

(ii) For any transition $(x_G, x_T) \xrightarrow{\alpha}^{\mathcal{S}_{\mathcal{U}}} (y_G, y_T)$ *in* $\mathcal{S}_{\mathcal{U}}(G \parallel T)$*, at least one of the following two statements is true:*

  a) $([x_G], x_T) \xrightarrow{\mathsf{p}(\alpha)}^{\mathcal{S}_{\mathcal{U}}} ([y_G], y_T)$ *in* $\mathcal{S}_{\mathcal{U}}(G/{\sim} \parallel T)$*, or*

  b) (x_G, x_T) *is not reachable in* $\mathcal{S}_{\mathcal{U}}(G \parallel T)$*.*

Proof. The proof of Proposition 3.2.20 applies analogously to the current proposition by replacing Proposition 3.2.19 with Proposition B.3. □

With Propositions B.3 and B.4, adjusting Theorem 3.2.21 turns out to be straightforward as follows.

Theorem B.5 (adjusted from Theorem 3.2.21)**.** *Let* $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ *be a* Υ*-shaped automaton and the equivalence* $\sim \subseteq Q_G \times Q_G$ *is induced by the redundant silent step* $\bar{x}_G \xrightarrow{\tau} \bar{x}'_G$*. It holds that* $G \simeq^{\mathcal{S}_{\mathcal{U}}} (G/{\sim})$*.*

Proof. The proof of Theorem 3.2.21 applies analogously to the current theorem by replacing Proposition 3.2.20 with Proposition B.4. □

Abstraction rules based on incoming equivalence

Both the active events rule and the silent continuation rule are based on the incoming equivalence, whose key property is the redirectability. Since redirectability is defined over the shaped synchronous composition, an adjustment to embed unification operator is necessary in the current context. This results in the following definition of *$\mathcal{U}$-redirectability*.

Definition B.6 (adjusted from Definition 3.2.22). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ be a Υ-shaped automaton. An equivalence $\sim \subseteq Q_G \times Q_G$ is $\mathcal{U}$-*redirectable *if and only if for any automaton $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^\circ_T, M_T\rangle$, $y_G \in Q_G$, $y_T \in Q_T$ and $s_T \in (\mathsf{aug}(\Sigma_{T\setminus G}))^*$, the following two statements hold:*

(R1') $(x_G, x_T) \xrightarrow{\sigma}{}^{\mathcal{S}} \overset{s_T}{\Longrightarrow}{}^{\mathcal{S}_\mathcal{U}} (y_G, y_T)$ *in $\mathcal{S}_\mathcal{U}(G \parallel T)$ for any $x_G \in Q_G$, $x_T \in Q_T$ and $\sigma \in \mathsf{aug}(\Sigma_G \cup \Sigma_T) - \mathsf{aug}(\Sigma_{T\setminus G})$ implies that for all $y'_G \in [y_G]$, $(x_G, x_T) \overset{\sigma s_T}{\Longrightarrow}{}^{\mathcal{S}_\mathcal{U}} (y'_G, y_T)$ holds.*

(R2') $\mathcal{S}_\mathcal{U}(G \parallel T) \overset{s_T}{\Longrightarrow}{}^{\mathcal{S}_\mathcal{U}} (y_G, y_T)$ *implies that for all $y'_G \in y_G$, $\mathcal{S}(G \parallel T) \overset{s_T}{\Longrightarrow}{}^{\mathcal{S}_\mathcal{U}} (y'_G, y_T)$.*

The key property of redirectability which was stated in Proposition 3.2.23 can be adapted in a straightforward manner. Nevertheless, a minor supplement w.r.t. Lemma 3.2.5 is essential for proving the following proposition. In the proof of Proposition 3.2.23, the implication *if $([x_G], x_T) \xrightarrow{\alpha}{}^{\mathcal{S}} ([y_G], y_T)$ in $\mathcal{S}(G/{\sim} \parallel T)$, then there exists $x'_G \in [x_G]$ and $y'_G \in [y_G]$ so that $(x'_G, x_T) \xrightarrow{\alpha}{}^{\mathcal{S}} (y'_G, y_T)$ in $\mathcal{S}(G \parallel T)$* is clearly true from Lemma 3.2.5. However, in the current context where we intend to replace $\mathcal{S}(\cdot)$ with $\mathcal{S}_\mathcal{U}(\cdot)$, the implication is invalidated if two equivalent states have different non-empty sets of active unifiable events. This issue can be solved by additionally requiring $\sim_{\text{ae}}$ or $\sim_{\text{sc}}$ on the equivalence.

Lemma B.7. *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq Q_G \times Q_G$ on G so that either $\sim \subseteq \sim_{ae}$ or $\sim \subseteq \sim_{sc}$. For any arbitrary automaton $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^\circ_T, M_T\rangle$ and any transition $([x_G], x_T) \xrightarrow{\alpha}{}^{\mathcal{S}_\mathcal{U}} ([y_G], y_T)$ in $\mathcal{S}_\mathcal{U}(G/{\sim} \parallel T)$, there exists $x'_G \in [x_G]$ and $y'_G \in [y_G]$ so that $(x'_G, x_T) \xrightarrow{\alpha}{}^{\mathcal{S}_\mathcal{U}} (y'_G, y_T)$ in $\mathcal{S}_\mathcal{U}(G \parallel T)$*

Proof. If $([x_G], x_T) \xrightarrow{\alpha}{}^{\mathcal{S}_\mathcal{U}} ([y_G], y_T)$ is not driven by G, then the statement is obviously true due to Lemma 3.2.5.(ii). In particular, if $\alpha \in \mathsf{aug}(\Sigma^{\mathrm{u}}_{T\setminus G})$, then for all $x'_G \in [x_G]$, we have $G^{\mathrm{u}}(x'_G) = \emptyset$. Thus, we consider the case in which $([x_G], x_T) \xrightarrow{\alpha}^{\mathcal{S}_{\mathcal{U}}} ([y_G], y_T)$ is driven by G. Clearly, from Lemma 3.2.5, it suffices to consider the case in which $\alpha \in \Psi$. We prove by contradiction in the following: if the statement does not hold, then there must exist two distinct states $x'_G, x''_G \in [x_G]$ so that $G^{\mathrm{u}}(x'_G) \neq \emptyset$, $G^{\mathrm{u}}(x''_G) \neq \emptyset$ and $G^{\mathrm{u}}(x'_G) \neq G^{\mathrm{u}}(x''_G)$. This contradicts the definitions of both $\sim_{\mathrm{ae}}$ and $\sim_{\mathrm{sc}}$. In particular, if $\sim \subseteq \sim_{\mathrm{sc}}$, then $G^{\mathrm{u}}(x_G) \neq \emptyset$ implies that $[x_G]$ is a singleton for any $x_G \in Q_G$ due to Assumption 1. □

With Lemma B.7, we are in the position to adjust Proposition 3.2.23 as follows.

Proposition B.8 (adjusted from Proposition 3.2.23). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ be a Υ-shaped automaton with a $\mathcal{U}$-redirectable equivalence $\sim \subseteq Q \times Q$ on G so that either $\sim \subseteq \sim_{ae}$ or $\sim \subseteq \sim_{sc}$. For any automaton $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^\circ_T, M_T\rangle$, the following two statements hold:*

(i) For any trace

$$([x_{G0}], x_{T0}) \xrightarrow{\alpha_1}^{\mathcal{S}_{\mathcal{U}}} ([x_{G1}], x_{T1}) \xrightarrow{\alpha_2}^{\mathcal{S}} \cdots \xrightarrow{\alpha_k}^{\mathcal{S}} ([x_{Gk}], x_{Tk}) \tag{169}$$

in $\mathcal{S}_{\mathcal{U}}(G/{\sim} \parallel T)$ where $k \geq 1$, $\alpha_1 \in \mathsf{aug}(\Sigma_G \cup \Sigma_T) - \mathsf{aug}(\Sigma_{T\setminus G})$ and $\alpha_i \in \mathsf{aug}(\Sigma_G \cup \Sigma_T) \cup \Upsilon$ for all $i \in \{2, \cdots, k\}$, there exist $x'_{G0} \in [x_{G0}]$ and $x'_{Gk} \in [x_{Gk}]$ so that $(x'_{G0}, x_{T0}) \xRightarrow{\mathsf{p}(\alpha_1 \cdots \alpha_k)}^{\mathcal{S}_{\mathcal{U}}} (x'_{Gk}, x_{Tk})$ in $\mathcal{S}_{\mathcal{U}}(G \parallel T)$;

(ii) If $\mathcal{S}_{\mathcal{U}}(G/{\sim} \parallel T) \xRightarrow{s}^{\mathcal{S}_{\mathcal{U}}} ([x_G], x_T)$ for some $s \in (\mathsf{aug}(\Sigma_G \cup \Sigma_T))^*$, then there exists $x'_G \in [x_G]$ so that $\mathcal{S}_{\mathcal{U}}(G \parallel T) \xRightarrow{s}^{\mathcal{S}_{\mathcal{U}}} (x'_G, x_T)$.

Proof. The proof of Proposition 3.2.23 applies analogously to the current proposition through replacing Lemma 3.2.5 with Lemma B.7. □

Following Section 3.2.2, we are now in the position to state that the conjunction of an incoming equivalence with either an active-event equivalence or an silent-continuation equivalence is $\mathcal{U}$-redirectable. This conceivably requires adjustments in Lemmata 3.2.29 and 3.2.30 as well as Propositions 3.2.31 and 3.2.32. We first consider adjusting Lemma 3.2.29.

Lemma B.9 (adjusted from Lemma 3.2.29). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ be a Υ-shaped automaton. Let $\sim \subseteq Q \times Q$ be an equivalence on G so that either $\sim \subseteq \sim_{ae}$ or $\sim \subseteq \sim_{sc}$. For any automaton $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^\circ_T, M_T\rangle$ and any trace*

$$(x_G, x_{T0}) \xrightarrow{\tau_1}^{\mathcal{S}_{\mathcal{U}}} (x_G, x_{T1}) \xrightarrow{\tau_2}^{\mathcal{S}_{\mathcal{U}}} \cdots \xrightarrow{\tau_k}^{\mathcal{S}_{\mathcal{U}}} (x_G, x_{Tk}) \tag{170}$$

in $\mathcal{S}_{\mathcal{U}}(G \parallel T)$ where $k \geq 0$ and $\tau_i \in \mathsf{aug}(\Sigma_{T \setminus G})$ for all $i \in \{1, \dots, k\}$, it holds that for any $x'_G \in [x_G]$, a trace

$$(x'_G, x_{T0}) \xrightarrow{\tau_1}^{\mathcal{S}_{\mathcal{U}}} (x'_G, x_{T1}) \xrightarrow{\tau_2}^{\mathcal{S}_{\mathcal{U}}} \dots \xrightarrow{\tau_k}^{\mathcal{S}_{\mathcal{U}}} (x'_G, x_{Tk}) \tag{171}$$

must exist in $\mathcal{S}_{\mathcal{U}}(G \parallel T)$ as well.

Proof. The proof of Lemma 3.2.29 applies analogously to the current lemma. □

In the following, Lemma 3.2.30 as well as Propositions 3.2.31 and 3.2.32 are to adjust. These statements show properties of *asynchronous traces*, which are defined as such that all events appearing on such traces are *private*. In Chapter 3, *asynchronous* and *private* are synonymous concepts. This is clearly not the case when the unification operator is taken into consideration, since unifying private unifiable transitions results in a synchronous transition. Hence, we slightly strengthen the definition of an asynchronous trace in $\mathcal{S}_{\mathcal{U}}(G \parallel T)$ as such that all events on the trace is in either $\mathsf{aug}(\Sigma_{T \setminus G})$ or Υ. This extension enables the adjustments in the sequel.

Lemma B.10 (adjusted from Lemma 3.2.30). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^{\circ}_G, M_G \rangle$ and $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^{\circ}_T, M_T \rangle$ be two arbitrary automata and*

$$(x_G, x_{T0}) \xrightarrow{\tau_1}^{\mathcal{S}_{\mathcal{U}}} (x_G, x_{T1}) \xrightarrow{\tau_2}^{\mathcal{S}_{\mathcal{U}}} \dots \xrightarrow{\tau_k}^{\mathcal{S}_{\mathcal{U}}} (x_G, x_{Tk}) \xrightarrow{\tau_{k+1}}^{\mathcal{S}_{\mathcal{U}}} (y_G, x_{Tk}) \tag{172}$$

be an asynchronous trace in $\mathcal{S}_{\mathcal{U}}(G \parallel T)$ so that $k \geq 0$ and for all $i \in \{1, \cdots, k\}$, $(x_G, x_{Ti-1}) \xrightarrow{\tau_j}^{\mathcal{S}_{\mathcal{U}}} (x_G, x_{Tj})$ is driven by T and $(x_G, x_{Tk}) \xrightarrow{\tau_{k+1}}^{\mathcal{S}_{\mathcal{U}}} (y_G, x_{Tk})$ is driven by G. It holds that $\mathsf{prio}(\tau_{k+1}) \geq \mathsf{lo}(\{\tau_1, \cdots, \tau_k\})$.

Proof. The proof of Lemma 3.2.30 applies analogously to the current lemma. □

Proposition B.11 (adjusted from Proposition 3.2.31). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^{\circ}_G, M_G \rangle$ and $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^{\circ}_T, M_T \rangle$ be two arbitrary automata and*

$$(x_{G0}, x_{T0}) \xrightarrow{\tau_1}^{\mathcal{S}_{\mathcal{U}}} (x_{G1}, x_{T1}) \xrightarrow{\tau_2}^{\mathcal{S}_{\mathcal{U}}} \dots \xrightarrow{\tau_k}^{\mathcal{S}_{\mathcal{U}}} (x_{Gk}, x_{Tk}) \tag{173}$$

be an asynchronous trace in $\mathcal{S}_{\mathcal{U}}(G \parallel T)$ so that $k \geq 1$ and the last transition $(x_{Gk-1}, x_{Tk-1}) \xrightarrow{\tau_k}^{\mathcal{S}_{\mathcal{U}}} (x_{Gk}, x_{Tk})$ is driven by G.

(i) Let $n = \mathsf{lo}(\{\tau_1, \cdots, \tau_k\})$. It holds that $T^{<n}_{\mathsf{prvt}}(x_{Tk}) = \emptyset$.

(ii) Let $n_G = \mathsf{lo}(\{\tau_i \mid (x_{Gi-1}, x_{Ti-1}) \xrightarrow{\tau_i}{}^{\mathcal{S}} (x_{Gi}, x_{Ti})$ *is driven by* $G\})$ *and* $n_T = \mathsf{lo}(\{\tau_i \mid (x_{Gi-1}, x_{Ti-1}) \xrightarrow{\tau_i}{}^{\mathcal{S}_{\mathcal{U}}} (x_{Gi}, x_{Ti})$ *is driven by* $T\})$*. It holds that* $n_G \geq n_T$*.*

Proof. The proof of Proposition 3.2.31 applies analogously to the current proposition by replacing Lemma 3.2.30 with Lemma B.10. □

Proposition B.12 (adjusted from Proposition 3.2.32). *Let* $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^{\circ}_G, M_G\rangle$ *be a* Υ*-shaped automaton and*

$$(x_{G0}, x_{T0}) \xrightarrow{\tau_1}{}^{\mathcal{S}_{\mathcal{U}}} (x_{G1}, x_{T1}) \xrightarrow{\tau_2}{}^{\mathcal{S}_{\mathcal{U}}} \cdots \xrightarrow{\tau_k}{}^{\mathcal{S}_{\mathcal{U}}} (x_{Gk}, x_{Tk}) \tag{174}$$

be an asynchronous trace in $\mathcal{S}(G \parallel T)$ *where* $k \geq 0$ *and let* $n = \mathsf{lo}(\{\tau_i \mid (x_{Gi-1}, x_{Ti-1}) \xrightarrow{\tau_i} (x_{Gi}, x_{Ti})$ *is driven by* $G\})$*. Let*

$$x'_{G0} \xrightarrow{\tau'_1} x'_{G1} \xrightarrow{\tau'_2} \cdots \xrightarrow{\tau'_{k'}} x'_{Gk'} \tag{175}$$

with $k' \geq 0$ *be a trace in* G *so that all events on this trace are silent,* $\mathsf{lo}(\{\tau'_1, \cdots, \tau'_{k'}\}) = n$ *and for all* $i' \in \{1, \cdots, k'-1\}$, $G^{<\tau'_i}_{\mathrm{rglr}}(x'_{Gi'}) = \emptyset$*. The following two statements hold:*

(i) For the trace given in (126), if $k \geq 1$ *and the last transition* $(x_{Gk-1}, x_{Tk-1}) \xrightarrow{\tau_k}{}^{\mathcal{S}_{\mathcal{U}}} (x_{Gk}, x_{Tk})$ *is driven by* G*, then* $(x'_{G0}, x_{T0}) \xRightarrow{\mathsf{p}(\tau_1\cdots\tau_k)}{}^{\mathcal{S}_{\mathcal{U}}} (x'_{Gk'}, x_{Tk})$ *in* $\mathcal{S}(G \parallel T)$ *where the last transition is driven by* G*.*

(ii) Let $\sim \subseteq Q_G \times Q_G$ *be an equivalence on* G *so that either* $\sim \subseteq \sim_{ae}$ *or* $\sim \subseteq \sim_{sc}$*. If* $x_{Gk} \sim x'_{Gk'}$*, then* $(x'_{G0}, x_{T0}) \xRightarrow{\mathsf{p}(\tau_1\cdots\tau_k)}{}^{\mathcal{S}_{\mathcal{U}}} (x'_{Gk'}, x_{Tk})$ *in* $\mathcal{S}(G \parallel T)$*.*

Proof. The proof of Proposition 3.2.32 applies analogously to the current proposition by replacing Lemma 3.2.29 and Proposition 3.2.31 with Lemma B.9 and Proposition B.11, respectively. Note that for proving statement (i), when constructing an asynchronous trace, transition unification can never happen due to Assumption 1. More precisely, for the silent trace given in (175), it is implicitly guaranteed that for all $i' \in \{1, \cdots, k'-1\}$, $G^{\mathrm{u}}(x'_{Gi'}) = \emptyset$. □

With Proposition B.12, we are prepared to declare that the conjunction of $\sim_{\mathrm{inc}}$ with either $\sim_{\mathrm{ae}}$ or $\sim_{\mathrm{sc}}$ is indeed $\mathcal{U}$-redirectable.

Proposition B.13 (adjusted from Proposition 3.2.28). *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M\rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq Q \times Q$ on G be such that either $\sim \subseteq \sim_{inc} \cap \sim_{ae}$ or $\sim \subseteq \sim_{inc} \cap \sim_{sc}$. It holds that $\sim$ is redirectable.*

Proof. The proof of Proposition 3.2.28 applies analogously to the current proposition by replacing Lemma 3.2.29 and Proposition 3.2.32 with Lemma B.9 and Proposition B.12, respectively. □

We are now finally at the stage to prove that the active events rule and the silent continuation rule are both $\mathcal{U}$-conflict-preserving. This requires adjustments of Proposition 3.2.33, Lemma 3.2.34 as well as both Theorems 3.2.35 and 3.2.36.

Proposition B.14 (adjusted from Proposition 3.2.33). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq Q_G \times Q_G$ on G so that either $\sim \subseteq \sim_{ae}$ or $\sim \subseteq \sim_{sc}$ holds. For any arbitrary automaton $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^\circ_T, M_T\rangle$ and any transition $(x_G, x_T) \xrightarrow{\alpha}^{\mathcal{S}_\mathcal{U}} (y_G, y_T)$ in $\mathcal{S}_\mathcal{U}(G \parallel T)$, it holds that $([x_G], x_T) \xrightarrow{\mathsf{p}(\alpha)}^{\mathcal{S}_\mathcal{U}} ([y_G], y_T)$ in $\mathcal{S}_\mathcal{U}(G/\!\sim \parallel T)$.*

Proof. The proof of Proposition 3.2.33 applies analogously to the current proposition. □

Lemma B.15 (adjusted from Lemma 3.2.34). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq \sim_{ae}$. Then for any arbitrary automaton $T = \langle Q_T, \Sigma_T, \rightarrow_T, Q^\circ_T, M_T\rangle$, if $([x_G], x_T) \xRightarrow{s_T\mathsf{p}(\alpha)}^{\mathcal{S}_\mathcal{U}}$ in $\mathcal{S}_\mathcal{U}(G/\!\sim \parallel T)$ for some $x_G \in Q_G$, $x_T \in Q_T$, $s_T \in (\mathsf{aug}(\Sigma_{T\setminus G}))^*$ and $\alpha \in (\mathsf{aug}(\Sigma_G \cup \Sigma_T) - \mathsf{aug}(\Sigma_{T\setminus G})) \cup \Upsilon$, then for all $x'_G \in [x_G]$, $(x'_G, x_T) \xRightarrow{s_T\mathsf{p}(\alpha)}^{\mathcal{S}_\mathcal{U}}$ in $\mathcal{S}_\mathcal{U}(G \parallel T)$.*

Proof. The proof of Lemma 3.2.34 applies analogously to the current lemma by replacing Lemma 3.2.29 with Lemma B.9. □

Theorem B.16 (adjusted from Theorem 3.2.35). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q^\circ_G, M_G\rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq \sim_{ae} \cap \sim_{inc}$ on G. It holds $G \simeq^{\mathcal{S}_\mathcal{U}} (G/\!\sim)$.*

Proof. The proof of Theorem 3.2.35 applies analogously to the current theorem through the following uniform substitutions:

- Replace Lemma 3.2.34, Propositions 3.2.33, 3.2.23 and 3.2.28 with Lemma B.15, Propositions B.14, B.8 and B.13, respectively;

- Replace *redirectable* with *$\mathcal{U}$-redirectable*;
- Replace $\sigma \in \Sigma_G - \Omega$ and $\sigma' \in \Sigma_G$ with $\sigma \in \mathsf{aug}(\Sigma_G \cup \Sigma_T) - \mathsf{aug}(\Sigma_{T \setminus G}) - \Omega$ and $\sigma' \in \mathsf{aug}(\Sigma_G \cup \Sigma_T) - \mathsf{aug}(\Sigma_{T \setminus G})$ in Case 2. □

Theorem B.17 (adjusted from Theorem 3.2.36). *Let $G = \langle Q_G, \Sigma_G, \rightarrow_G, Q_G^\circ, M_G \rangle$ be a Υ-shaped automaton with an equivalence $\sim \subseteq Q_G \times Q_G$ on G so that $\sim \subseteq \sim_{inc} \cap \sim_{sc}$. It holds $G \simeq^{\mathcal{S}_\mathcal{U}} (G/\sim)$.*

Proof. The proof of Theorem 3.2.36 applies analogously to the current theorem through the following uniform substitutions:

- Replace Propositions 3.2.33, 3.2.23 and 3.2.28 with Propositions B.14, B.8 and B.13, respectively;
- Replace *redirectable* with *$\mathcal{U}$-redirectable*;
- Replace $\sigma \in \Sigma_G$ with $\sigma \in \mathsf{aug}(\Sigma_G \cup \Sigma_T) - \mathsf{aug}(\Sigma_{T \setminus G})$ in Case 1;
- Replace $\alpha \notin \Sigma_G$ with $\alpha \notin \mathsf{aug}(\Sigma_G \cup \Sigma_T) - \mathsf{aug}(\Sigma_{T \setminus G})$ in Case 3. □

Further abstraction rules

We have also introduced three abstraction rules in Section 3.2.3, i.e. the only silent incoming rule, the only silent outgoing rule and the certain conflicts rule. Recall that the first two rules originate from combining PWB and silent continuation rule, while the latter rule is simply inspired by the fact that blocking behaviour of an automaton can be merged without caring about its explicit structure. All these rules are obviously $\mathcal{U}$-conflict-preserving. For consistency, we subtly adjust Theorems 3.2.37, 3.2.38 and 3.2.39 as follows, where we only uniformly substitute each $\simeq^{\mathcal{S}}$ relation with $\simeq^{\mathcal{S}_\mathcal{U}}$.

Theorem B.18 (adjusted from Theorem 3.2.37). *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$ be a Υ-shaped automaton and let $\bar{x} \in Q$ be such that $\bar{x}$ is not in any live-lock, $\tau_{(1)} \in G(\bar{x})$ and $y \xrightarrow{\alpha} \bar{x}$ implies $\alpha = \tau_{(1)}$. Then for the automaton $G' = \langle Q, \Sigma, \rightarrow', Q^\circ, M \rangle$ with*

$$\rightarrow' = \{(x, \alpha, y) \mid x \xrightarrow{\alpha} y \, \textit{and} \, y \neq \bar{x}\} \cup \{(x, \alpha, y) \mid x \xrightarrow{\tau} \bar{x} \xrightarrow{\alpha} y\}, \qquad (176)$$

it holds that $G \simeq^{\mathcal{S}_\mathcal{U}} G'$.

Theorem B.19 (adjusted from Theorem 3.2.38). *Let $G = \langle Q, \Sigma, \rightarrow, Q^\circ, M \rangle$ be a Υ-shaped automaton and let $\bar{x} \in Q$ be such that $\bar{x}$ is not in any live-lock* *and $G(\bar{x}) = \{\tau_{(1)}\}$. Let $\bar{Q} := \{y \in Q \mid \bar{x} \xrightarrow{\tau_{(1)}} y\}$, then for the automaton $G' = \langle Q - \{\bar{x}\}, \Sigma, \to', Q^{\circ\prime}\rangle$ with*

$$Q^{\circ\prime} = \begin{cases} Q^\circ & \text{if } \bar{x} \notin Q^\circ \\ (Q^\circ - \{\bar{x}\}) \cup \bar{Q} & \text{if } \bar{x} \in Q^\circ \end{cases}; \tag{177}$$

$$\to' = \{(x,\alpha,y) \mid x \xrightarrow{\alpha} y \text{ and } \bar{x} \notin \{x,y\}\} \cup \{(x,\alpha,y) \mid x \xrightarrow{\alpha} \bar{x} \text{ and } y \in \bar{Q}\}, \tag{178}$$

it holds that $G \simeq^{\mathcal{S}_\mathcal{U}} G'$.

Theorem B.20 (adjusted from Theorem 3.2.39). *Let $G = \langle Q, \Sigma, \to, Q^\circ, M\rangle$ be a Υ-shaped automaton. Let $Q_\text{c} \subseteq Q$ be the set of co-reachable states in G and $Q_\text{uc} := Q - Q_\text{c}$ the set of non-co-reachable states in G. Define two transition sets as*

$$\to_1 := \{x \xrightarrow{\alpha} y \mid x \in Q_\text{c}, \alpha \in A, y \in Q \text{ and } \exists y' \in Q_\text{uc}, \tau \in \Upsilon.\ G^{<\tau}_\text{rglr}(x) = \emptyset \wedge x \xrightarrow{\tau} y'\}; \tag{179}$$

$$\to_2 := \{x \xrightarrow{\sigma} y \mid x \in Q_\text{c}, \sigma \in \Sigma, y \in Q_\text{c}, G^{<\sigma}_\text{rglr}(x) = \emptyset \text{ and } \exists y' \in Q_\text{uc}.\ x \xrightarrow{\sigma} y'\} \tag{180}$$

and let $G' = \langle Q, \Sigma, \to - (\to_1 \cup \to_2), Q^\circ, M\rangle$. It holds that $G \simeq^{\mathcal{S}_\mathcal{U}} G'$.

C Tables of symbols

Important symbols utilised in the current dissertation are listed in the following tables. Note that in different chapters, we sometimes use a same symbol to refer to as elements in different sets. For instance, n is referred to as a *node* in Chapter 2 while a *priority value* in Chapters 3 and 4.

General symbols

symbol	description	page
$i, j, k \in \mathbb{N}_0$	indices	15
Σ	non-silent event set	37, 70
$\sigma \in \Sigma$	non-silent event	37, 70

Symbols in Chapter 2

symbol	description	page
$S, T \in \mathsf{SBDP}$	SBDs	15
$n, m \in \mathsf{Nodes}$	nodes in SBDs	15
$\iota \in \mathbb{N}_0$	logic time instance	21
$h \in \mathsf{HEs}$	hyper-edge	21
$v \in \mathsf{Variables}$	variable	31
$l \in \mathsf{range}(v)$	value of the variable v	39

Symbols in Chapter 3

symbol	description	page
$\mathfrak{E}$	universe of events	69
Υ	silent event set	69

$n, m \in \mathbb{N}$	priority value	69
$A \subseteq \mathfrak{E}$	event set	69, 70
$A^{<n}$, $A^{\leq n}$	events with priority higher than (or not lower than) n in A	69
$\alpha \in \mathfrak{E}$	event	69
$x, y, z \in Q$	state	70
$\tau \in \Upsilon$	silent event	69
$\Sigma_{T \setminus G} \subseteq \mathfrak{E} - \Upsilon$	regular private event set in the test automaton T	81
$\tau \in \Sigma_{T \setminus G}$	regular private event in the test automaton T	81
prio	priority assignment function	69
lo	lowest priority of a given set of events	69
hide	hiding map	70
p	natural projection	70
$G(x)$	active events in the state x	71
$G/_t$	hiding transition t in G	75
$\mathcal{S}$	shaping operator	73
$\mathcal{S}_\Upsilon$	Υ-shaping operator	76
$\rightarrow$	transition relation	70
$\Rightarrow$	abstract transition relation	71
$\xrightarrow[\Delta:n]{}$, $\xRightarrow[\Delta:n]{}$, $\underset{n}{\rightarrow}$	extended transition relations	81
$\Rrightarrow$	extended transition relation (for defining APWB only)	87
$\underset{!}{\rightarrow}$, $\underset{n}{\hookrightarrow}$	extended transition relations (for defining incoming equivalence only)	96
$\simeq^{\mathcal{S}}$	conflict equivalence	76
$\approxeq$	PWB (over two automata)	82
$\approx$	PWB (over one automaton)	84
$\approx^*$	APWB	87
$\sim_{\text{inc}}$	incoming equivalence	97
$\sim_{\text{ae}}$	active-event equivalence	97
$\sim_{\text{sc}}$	silent-continuation equivalence	98

Symbols in Chapter 4

symbol	description	page
$\mathfrak{U}$	universe of unifiable symbols	144
Ψ	unifiable event set	144
$\psi \in \Psi$	unifiable event	145
aug	event set augmentation (w.r.t. unifiable events)	145
$\mathcal{U}$	unification operator	145
$\mathcal{S}_{\mathcal{U}}$	shaped unification	147
$\simeq^{\mathcal{S}_{\mathcal{U}}}$	$\mathcal{U}$-conflict equivalence	149

In recent decades, discrete-event modelling has been widely utilised to address control engineering problems. Comparing with conventional dynamic system modelling where physical behaviour is explicitly to describe, discrete-event modelling focuses on a more abstract level where logical behaviour is of interest. In this dissertation, we focus on the formal verification of the logical closedloop behaviour of control systems. To satisfy safety and/or liveness requirements according to given technical specifications, we exploit the formal semantics of control programmes to represent the entire closed-loop behaviour in a discrete-event model, from which the properties of interest can be formally verified through an efficient method.

ISBN 978-3-96147-743-2

9 783961 477432

FAU UNIVERSITY PRESS 2024