Compositional Modelling and Verification of Self-Adaptive Cyber Physical Systems

Aimee Borda
bordaa@tcd.ie

under the supervision of Dr. Vasileios Koutavas

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thesis submitted to the School of Computer Science and Statistics in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science

October 22, 2019

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidates own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgments.

Aimee Borda

The possession of knowledge does not kill the sense of wonder and mystery. There is always more mystery.

—*Anais Nin*

Abstract

Cyber-Physical Systems (CPSs) must often self-adapt to respond to changes in their operating environment. However, providing assurances of critical requirements through formal verification techniques can be computationally intractable due to the large state space of self-adaptive CPSs. In this thesis we propose a methodology to support assurances of such systems, which employs a novel modelling language, Adaptive CSP, enabling *compositional reasoning* for tractable verification. The process language extends Communicating Sequential Processes (CSP) with constructs to reduce the effort required to model and compositionally verify the adaptation options and events of self-adaptive CPSs.

Our methodology allows system designers to identify (a subset of) the CPS components that can affect satisfaction of each given requirement. An adaptation procedure for these components can then be created to preserve the requirement in the face of changes to the systems operating environment. Although manual, the task of identifying components relevant to a requirement can be guided by topological relationships, such as containment and connectivity between system components. Adaptive CSP can then be used to model the system including potential self-adaptation procedures. We propose a modular structure for adaptation procedures which, together with topology-guided adaptation, allows system designers to compare alternative adaptation procedures, potentially involving different sets of CPS components.

We show that with this approach, when different requirements involve disjoint sets of components, verification of each requirement can be performed against only its relevant components, in isolation from the rest of the system and other adaptation procedures. When components are relevant for multiple requirements, we develop a theory of compositionality to identify cases where interference between adaptation procedures is not possible. We prove that in such cases requirement verification can still be performed in isolation from other adaptation procedures and the rest of the system. In the other cases, the system designer must additionally verify that requirement satisfaction is preserved when components with corresponding adaptation procedures from interdependent requirements are composed.

Our methodology has the benefit of leveraging existing formal verification tools to check requirement satisfaction. We illustrate this through the use of FDR—an existing refinement checker for CSP, taking advantage of its advanced model minimisation and refinement checking functionality. The soundness of using FDR for verification relies on an adequate translation from a subset of Adaptive CSP to the language of FDR. We also alleviate the onus of modelling and verifying components in our framework further by providing a concrete syntax to Adaptive CSP, where we augment the process language with convenient idioms and macros from functional languages, together with a tool which translates Adaptive CSP code to FDR.

We demonstrate the feasibility of our methodology using a substantive motivating example of a smart art gallery. We further evaluate it with a case study of a smart stadium. Our results show that our methodology reduces the computational complexity of verifying self-adaptive CPSs and

can effectively support the design of adaptation procedures in such systems.

Acknowledgements

First of all, I would like to thank my supervisor Vasileios Koutavas. I can't be more grateful for his precious guidance throughout the PhD. His keen eye for details, everlasting patience and technical knowledge left me astounded a good number of times. This work is the product of hours of discussions and encouragement from him and so I thank Vassilis for all his help in trying to understand technical matters and the thoughts in my head. I am forever grateful to have been one of his students.

To Liliana Pasquale and Bashar Nuseibeh, the unofficial co-supervisors of this work from whom I learnt a lot. I extend my gratitude for the time spent in lengthy discussions and their knowledgeable feedback. Their insight on the domain and cheerful support kept me motivated throughout this journey. I like to acknowledge the Lero Research Centre and Trinity College Dublin as the sponsors of this work.

I like to extend a massive thank you to my family and friends back home. I love each and every one very dearly. In particular, a special thanks goes to my sister which from Day -9 months has been by side every step of the way. She has been my ultimate travelling buddy, laughing buddy, whining buddy and Crossfit buddy. There is nothing I enjoy more in life than drinking a tea and americano in some small, sunny coffee shop located in the remotest of places talking for hours all things Crossfit and life plans. I extend my love to all the people I hold close to my heart back home, which there are too many to mention by name, that remind me everyday what is really important in life.

I thank the brilliant and witty people that I shared the office with over the last four years. In the early years, Colm Bhandal has been a steady stream of interesting mathematical puzzles and Carlo Spaccasassi provided interesting discussions on religion and philosophy. These were slowly replaced by the co-owners of the coffee club: Artur Gomes and Daniel Flynn, with whom everyday at 11am I laughed and had to be explained Irish slangs, with the classic remaining "the man with the one with the yoke" and the ever lovely Marian Reeves. I am very grateful to have met such wonderful people through this PhD.

Lastly, I thank the Dublin University Sub-Aqua Club, through which I met some of the most amazing people in Dublin and filled my thirst for adventure. The jelly-babies fuelled adventures on a small boat around the coast of Ireland will forever put a smile on my face.

Aimee Borda, Trinity College Dublin, October 22, 2019

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Computational and communication capabilities are being increasingly embedded into physical entities and processes, resulting in a proliferation of Cyber-Physical systems (CPSs) [18]. Notable examples include smart buildings (e.g., [48, 121, 120, 52, 128]) and autonomous vehicles (e.g., [23, 16, 112, 78, 99, 128]). CPSs exhibit a sophisticated interplay of digital (cyber) processes with a physical operational environment. An aim for such systems is to be *flexible* and *resilient* by effectively responding to a wide range of changes in their context through dynamic adaptation of their behaviour [86, 87]. A change in the context of a CPS can be anything from a foreseen spike in resource usage to a catastrophic natural disaster. CPSs cannot afford to have a manual process, with a human in the loop, to counteract all changes in their context, as this would be too disruptive and error-prone. Therefore CPSs often rely on built-in *self-adaptation* functionality to fulfill their goals in the face of a changing operational environment.

Self-adaptation is a system's capability to autonomously, without human input, detect when the operational environment changes and deploy counter-measures to guarantee the continued satisfaction of requirements. To engineer a SA system, a system designer needs to automate the inference of correct functionality in all possible contexts. Only then can an SA system decide autonomously, at runtime, an optimal behaviour. Moreover, as SACPSs are given autonomy over a wide range of actions that can affect the physical world, it is important that they are designed with high reliability in mind, and concrete assurances can be provided to their users. For this reason, it is crucial we understand the structure of self-adaptive systems and create techniques for building them and reasoning about their properties and runtime behaviour.

A roadmap produced by the self-adaptive community identifies two main *open challenges* that need to be addressed to better understand how to build effective self-adaptive systems [42, 32].

Systematic Software-Engineering Processes: Current research aims to provide a common framework and systematic software engineering techniques to effectively model SA systems [8]. A widely accepted view of SA systems is to distinguish their functionality in two parts—the *adaptation procedure* and the *base system*. The former encapsulates all the adaptation functionality: when, what and how to adapt; the base system contains the system functionality that may be adapted. This leads to a centralized SA system, where a single adaptation procedure has a global view of the system and determines all adaptations. In CPSs however, components may potentially be dispersed over a large area, or the CPS may need to operate in an open environment. This means that a centralized adaptation procedure that requires global and absolute knowledge of the entire

system may be impractical [127, 126, 42]. Recent research has been aiming at identifying systematic software engineering techniques for decentralized SA systems [107, 42]. In a decentralized setting, adaptation is going to emerge from the implicit composition of adaptation procedures operating on different parts of the system. A key challenge in this approach is dealing with the case where system components may need to be adapted by multiple adaptation procedures. In such a case adaptation procedures may interfere with each other leading to the violation of requirements. Effective and systematic approaches to localize adaptation to small parts of the systems may reduce the potential of interference. Yet, approaches to explicitly handle the interference between adaptation procedures are still needed [127, 126, 42].

Practical Verification Techniques: Self-adaptive CPSs increasingly support critical services. Errors can indeed be catastrophic. Thus it is important that high-level assurances are provided for these systems, guaranteeing that key requirements are satisfied in the presence of self-adaptation. The golden standard for providing such assurances is formal verification. The highly dynamic nature of SA CPSs, however, makes verification difficult due to the large state space of SA CPSs models that needs to be verified [107, 42, 32]. There are two challenges that need to be addressed to attain a tractable verification technique for such systems.

- Firstly, verifying self-adaptive CPSs using explicit state model checking of the entire system may suffer from the state explosion problem and be computationally infeasible for large-scale systems (e.g., [120]). Techniques where properties can be verified against a small part of the system, knowing that, once verified, these properties hold for the whole system are still lacking. Due to the versatility of CPSs, different groupings of concurrent cyber and physical components may need to be considered for each property.

- Secondly, verification needs to incorporate a large class of requirements that together define the overall behaviour of the system. Most existing verification techniques (e.g., [95, 28, 29, 71, 55, 54, 23, 94]) employ a utility function to quantify the satisfaction of multiple requirements. However, during the early design phase, we may not yet know appropriate weights and probabilities to define an effective utility function. A practical qualitative verification technique for SA CPSs with multiple requirements is therefore needed [5, 126, 26].

In this thesis, we propose to tackle the complexity of modelling and verifying the satisfaction of safety properties in SA CPSs through systematic and *compositional* techniques. Compositionality allows us to prove that the correctness of a verification task on a small part of the system holds for the entire system. Our approach is based on three key principles.

Topology-driven Modelling: Firstly we use a modelling methodology guided by the topological layout of CPSs and topological relations, such as containment and connectivity [102]. Consider, for example a smart building comprising rooms and sensors inside these rooms. In our model, we map the rooms and sensors to components, whereas the connectivity and containment relation between the identified components map to the interface between components.

We propose to model the components that result from such an architecture in a novel adaptation-aware process algebra, where connectivity is encoded as named events and containment as named locations. We call this language Adaptive CSP (ACSP), and we obtain it by extending Communicating Sequential Processes (CSP) [70] with locations and self-adaptation functionality. Being process based (as e.g., [89, 70]) and able to directly express self-adaptation, ACSP can readily support the definition of decentralized adaptation procedures at different levels of granularity in a system, as well as compositional reasoning for the system. In this thesis, the term granularity refers

to a group of CPS components over which an adaptation procedure aimed to ensure a requirement is defined whereas levels of granularity imply different groupings of components. Our use of the term granularity differs from the standard definition in computer science research literature [100].

Requirement-driven Adaptation: Secondly, we propose a methodology for modelling requirement-driven adaptation. The decentralization of adaptation logic in our model is guided by the topology and requirements. We utilize the topology and topological relations to systematically explore different grouping of components (levels of granularity) that affect the satisfaction of each requirement. We then model and verify an adaptation procedure over the selected group of components that aims to ensure the satisfaction of the requirement in the face of changes to the operational environment.

Compositional Verification: Topology-driven modelling and requirement-driven adaptation can already lead to models where adaptation functionality is distributed throughout the system. A theory of compositional reasoning guarantees that some verification tasks can be localised, without considering the entire system, drastically reducing the state explosion problem. When adaptation procedures are defined over disjoint parts of the system, their composition preserves the satisfaction of requirements because of the semantic properties of the modelling language, thus requiring no additional verification at the time of composition. When adaptation procedures have overlapping scope, there is potential of interference between them. Here, we need to verify the satisfaction of requirements after composing potentially interfering adaptation procedures. As we show, however, for certain types of overlaps, interference can be theoretically ruled out (e.g., when adaptation procedures only monitor, but not adapt, common components), thus allowing us to skip certain verification tasks at the time of composition. Our definition of compositional verification differs from the traditional meaning, e.g., [12, 76, 34], where the term is often used for approaches that construct system-level correctness proofs hierarchically from proofs of component-level properties. In this thesis, we leverage compositional verification techniques to safely localize verification tasks to a small part of the system. Moreover, our approach allows us to leverage existing, well-developed verification tools. In this dissertation, we use the refinement checker FDR [59]; however, our technique is general enough to be used with other verification tools such as (bi-)simulation [89, 108], testing preorders [43, 35], and modal logics [69, 11]..

1.2 Motivating Example: An Art Gallery

Throughout this dissertation we use a motivating example of an art gallery building. The two-floor plan of the gallery is shown in Figure 1.1. *Floor 1* includes a corridor and an exhibition area (rooms A, B, and D) where paintings are displayed. *Floor 2* includes a *Restoration Area* where maintenance and restoration of artwork is carried out. A heating, ventilation, and air conditioning system ($HVAC$) maintains a predefined target temperature and humidity level in the *Restoration Area*. A wireless access point in the *Computer Room* provides internet connectivity to the devices located in *Floor 2*. It is also connected to the $HVAC$ allowing the latter to be monitored and controlled remotely.

The art gallery would like to promote to the public the skilled work performed in the restoration area. Nevertheless, a critical security requirement for the art gallery is:

Requirement 1. Visitors should not interfere with the restoration process. ◊

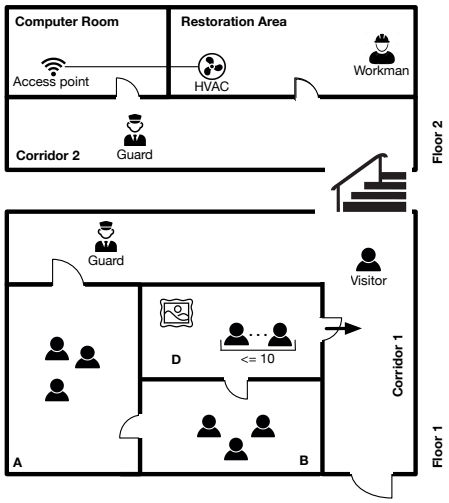


Figure 1.1: Two-floors plan of the Art Gallery.

This requirement may be violated when a visitor is in the *Restoration Area* without a staff member present in *Floor 2*. However, controlling access to the *Restoration Area* based on the presence of restoration staff might not always be desirable. For example, when restoration work is performed, access to the restoration area should be free of controls to allow efficient movement of workers. This can lead to the situation where visitors can access the *Restoration Area* without the presence of a staff member and interfere with the restoration work.

To avoid violating Req. 1 a system designer can introduce adaptation procedures at different granularities. Here we examine two such procedures.

Adaptation Procedure 1.1. Adaptation is applied at the granularity of the second floor, allowing free access to the *Restoration Area* only in the presence of a guard. ■

Ad. Proc. 1.1 and all adaptation procedures defined in this section f match the definition of a self-adaptive system presented before, where the system can autonomously determine if visitors should be precluded from entering the restoration area by monitoring the movement of the guard. However, this adaptation procedure can obstruct the movement of workers by locking the restoration area door when a guard is not present. The following adaptation procedure avoids this by using a coarser granularity which includes *Corridor 1* and the *Stairs*.

Adaptation Procedure 1.2. The guard from *Corridor 1* should escort visitors to *Floor 2* as soon as they start climbing the stairs. This ensures that visitors always reach the *Restoration Area* accompanied by a guard. ■

A very important painting at the core of the current exhibition is located in *Room D*. Visitors can enter this room from *Room B* and exit to *Corridor 1*. To maintain integrity of the painting, the following requirement should be satisfied.

Requirement 2. No more than ten visitors should be in *Room D* at the same time. ◊

To satisfy this requirement an adaptation can be used at the granularity of *Room D*.

Adaptation Procedure 2.1. Visitor entrance to *Room D* is allowed only when there are fewer than ten people in it. ■

However, if there is the possibility of multiple visitors entering *Room D* simultaneously (e.g., by tailgating), then the above adaptation may lead to a violation of Req. 2. An adaptation procedure at a coarser granularity, including *Room B* in its scope, can avoid this.

Adaptation Procedure 2.2. The total number of visitors in B, and D does not exceed the maximum number of people allowed in *Room D* alone; a guard is also located in *Corridor 1* (to prevent tailgating into *Room A*). ■

Back in *Floor 2*, a malicious visitor may connect a device to the wireless network and take control of the *HVAC*, exploiting security vulnerabilities of the wireless protocol. The following requirement is designed to prevent this.

Requirement 3. The HVAC should not be controlled remotely by unauthorised users. ◊

To satisfy this requirement an adaptation procedure can be designed at the granularity of *Floor 2*, which protects the *HVAC* from attacks while still allowing visitors to connect to the wireless network.

Adaptation Procedure 3.1. The *HVAC* is disconnected from the network when an untrusted device in the second floor is connected to the *Access Point*. ■

Additional requirements for the art gallery may involve overcrowding and emergency scenarios.

Requirement 4. No more than thirty visitors should be in the entire exhibition area at the same time. ◊

To satisfy this requirement, an adaptation procedure can monitor the movement of people into *Room A* and out of *Room D*—the entrance and exit, respectively, of the exhibition area.

Adaptation Procedure 4.1. A visitor is allowed to enter *Room A* from *Corridor 1* only when the people who have entered *Room A* from *Corridor 1* minus the people who have exited *Room D* into *Corridor 1* are no more than thirty. ■

This adaptation procedure allows all thirty people to be in the same room of the exhibition area. Although this is sufficient for Req. 4, the system designer may decide to refine this adaptation procedure in a way that spreads the people across the different rooms of the area.

Adaptation Procedure 4.2. Monitor visitors in rooms A, B and D and allow visitor movement between adjacent rooms only when the sum of people in adjacent rooms is less than fifteen. For example, movement from *Room A* to B is allowed when the number of visitors in *rooms B* and D is less than fifteen. Similarly, movement from *Corridor 1* to *room A* is only allowed when the number of visitors in *rooms A* and B is less than fifteen. ■

In case of an emergency the following requirement should also be satisfied

Requirement 5. The people in the building should be able to reach the nearest emergency exit. ◊

This requirement could be enforced at the granularity of the entire system, but this would require verification for the satisfaction of the requirement over the entire system. Instead, the following adaptation procedure is sufficient to guarantee the requirement.

Requirement 5 (revised)**.** At each room, movement in the case of an emergency is not restricted—all doors are open. ◊

Finally, the following requirement should also be satisfied.

Requirement 6. The *HVAC* is reconnected to the *access point* when updates are needed. ◊

Adaptation Procedure 6.1. To satisfy this requirement, when a new update is available, the *HVAC* should connect to the *Access Point* and install it. This means that visitors should be disconnected from the access point so the *HVAC* can connect and install the update. This adaptation procedure controls the *access point*, which is also controlled in the adaptation procedure designed to satisfy requirement 3. ■

1.3 Challenges

In this thesis, we aim to achieve a general tractable verification of the satisfaction of security requirements in SA CPSs such as the art gallery presented in the preceding section, and a stadium discussed in Chapter 9. We aim to achieve our goal through *compositional* verification, where verification tasks proven over a small part of the system also hold for the whole system. For instance, the verification of Req. 1 can be localized over the *restoration area* and through compositionality we have the guarantee the satisfaction of the requirement holds for the whole art gallery. We thus require a modelling framework and verification approach that supports compositionality. In our art gallery, we want to verify the satisfaction each requirement by considering only a small subset of the rooms and digital components knowing that it suffices to verify the satisfaction for the art gallery.

An important aspect for successfully exploiting compositional verification is identifying which components are important to include and which to ignore. Requirements for SA CPSs include a fusion of computation capabilities influenced by the physical layout of the system. Thus, identifying which components to include in verification tasks is not straightforward from the requirements. For example, even though both descriptions of Requirements 3 and 6 include the *HVAC*, their adaptation procedures focus solely on the *access point* that has the potential to connect unauthorized users to the *HVAC*.

Self-adaptation aims to evert as much as possible degradation in the level of service provided by an CPS, by adjusting the functionality with respect to a requirement after disruptive changes. Degradation in functionality may not always be avoidable [48] and the task of encoding effective adaptation procedures is hard and error prone. Designers of these systems must decide at which system execution points should adaptation be invoked in order for the system to operate efficiently and satisfy its requirements. We refer to the execution points where adaptation must be invoked as the *adaptation pattern*. These points can be as simple as timing events (e.g., every X seconds), can be triggered by specific system events, or even have a complex logic taking account the execution history and state of the system. Existing modelling approaches for SA systems often distinguish the adaptation decision process from the base system, but encode adaptation points directly in one, or both, of these components (e.g., [10, 64, 79, 47, 132, 115, 77]). The drawback of this approach is that the pattern of adaptation events becomes a cross-cutting across the system model. Modifying this pattern in search for a correct and optimal model often requires significant changes to the description of the base system and/or the adaptation decision process, which has to be done in an ad-hoc manner. However, modularising adaptation patterns is not easy [58]. Although some systems may be able to support such events at every state, others must disallow them during critical sections of their execution. Moreover, adaptation decision processes may be able to operate only at a subset of the system states.

At a global level, satisfying requirements in the presence of multiple adaptation procedures introduces another challenge. Adaptation procedures defined over common parts of a CPS may potentially interfere with each other and if adaptation is arbitrary, the interference may violate key requirements. We need to guarantee that the satisfaction of requirements is preserved when

all adaptation procedures are composed together. We need to explicitly and effectively address potential interference between overlapping adaptation procedures. For example, consider Requirements 2 and 4 from section 1.2 asserting the allowed number of visitors in *room D* and *exhibition area* respectively. The adaptation procedures overlap, yet we can verify that the composition of the adaptation procedures preserves the satisfaction of both Requirements 2 and 4. Consider now we compose an adaptation procedure that aims to ensure the satisfaction of Req. 5 requiring visitors to have a clear exit from the building in the case of an emergency. This adaptation procedure may allow any number of visitors in *room D* and the *exhibition area*, which may potentially violate Requirements 2 and 4. Here, we need to revise the adaptation procedures and potentially the requirements to resolve the conflict. In this example, we only apply Requirements 2 and 4 in a non-emergency situation and only apply Req. 5 in an emergency. In some cases, interference is not possible, e.g., when adaptation procedures are defined over disjoint components or the adaptation procedures only monitor, without adapting, common components. In these cases, we should be able to show that the satisfaction of requirements is preserved through compositionality theorems.

The challenge of attaining a compositional modelling framework for the tractable verification of SA CPSs is because of the complexity of SA CPSs.

1.4 Research Question

The main research question that we address in this thesis is

How can we tackle the complexity of modelling and verifying SA CPSs?

We propose a modelling and verification methodology that enables a system designer to tractably verify the satisfaction of security requirements in realistic SA CPSs. In particular we focus on smart buildings, where the layout is static i.e., components, like rooms or access points, are not added and removed dynamically. This is achieved by proposing a framework that support compositionality. The methodology , despite being manual, comprises software engineering techniques that guides a system designer to systematically and effectively attain a compositional model for SA CPSs. As we shall see in later chapters, the modelling process is based on topology and topological relations. We also propose a verification process that decomposes the task of verifying the satisfaction of security requirements in SA CPSs to reduce the complexity of realizing the verification. Finally, the proposed modelling methodology allow us to leverage existing verification tools and therefore avail of the years of experience in optimizing and advancing the verification technique.

1.5 Contributions

We answer our research question through two high-level contributions. To achieve the high-level contributions, we also present a number of technical sub-contributions.

A modelling methodology for SA CPSs This methodology is derived through software engineering techniques to tackle the complexity of defining a compositional model for SA CPSs. From this perspective, technical contributions include

1. A novel process language called Adaptive CSP (ACSP) to *compositionally* model SA CPSs. The process language extends CSP with high-level constructs for specifying adaptation events and procedures of CPSs. The process language ACSP reduces the efforts required to verify the satisfaction of requirements over a subset of the CPS. This extension to CSP also enable us

to compose adaptation procedures to propose an iterative verification process for effectively proving the satisfaction of a number of requirements.

2. We also define a concrete representation for ACSP, which we call $ACSP_M$. The syntax for $ACSP_M$ is inspired from CSP_M [59], the machine readable dialect of CSP [70].

3. A requirement-driven adaptation approach that aims to ensures the satisfaction of safety requirements. In our framework, adaptation emerges from the implicit composition of adaptation procedures enforcing each requirement.

4. A CPS topology-driven technique to systematically identify the smallest set of components that affect the satisfaction of a requirement, over which an adaptation procedure enforcing the requirement would operate. This work has been published in [17].

5. A modular encoding for adaptation procedures that helps a system designer experiment and compare alternative adaptation procedures. We also propose generic properties an adaptation procedure (in our encoding) should possess. This work has been published in [16].

6. An evaluation of our framework through a second case-study: a smart stadium [92].

A practical verification process One of the persistent challenges faced by the SA systems community is the lack of a practical verification techniques [107, 42, 32]. In this thesis, we present a compositional verification process to verify the satisfaction of security requirements in SA CPSs using existing verification tools. The following technical contributions have been implemented to achieve this goal,

1. A translation for a subset of ACSP to CSP, the input language to FDR. We show this translation is adequate by showing the translation is a strong bisimulation and that it models an interesting class of SA CPSs by encoding the art gallery and smart stadium case-studies.

2. A theory of compositionality that guarantees verification results proven over different subsets of an CPS which localizes adaptation are preserved for the whole system.

3. An extension of this theory to derive a technique to systematically verify the satisfaction of requirements by the composition of adaptation procedures with potentially overlapping scope. We identify cases where interference is not possible and prove that requirement satisfaction in such cases is preserved by composition, thus allowing us to avoid re-verifying the composed system.

4. The automation of the translation from ACSP to CSP with a prototype tool. This tool integrates FDR to present a complete verification process. The input language for the tool is the concrete syntax of ACSP, $ACSP_M$. The concrete syntax augments ACSP with idioms inspired from functional programming languages to alleviate the task of encoding realistic problems.

1.6 Limitations and Assumptions

In this thesis, only requirements that can be expressed in the modelling formalism employed by the methodology can be verified. Our framework does not take into account that the probability of certain events occurring is known. The utilization of known probabilities can improve the effectiveness of the verification process. For example, a SA security system might have established

statistics about the possibility of a hacking or burglar attempt and the probability of bypassing specific security measures like breaking a password. There are other interesting and relevant classes of requirements that would be worth exploring as a future research direction of this thesis. This include time-related requirements, e.g., requiring HVAC updates to be installed within 3 hours, and constraints on physical parameters, e.g., verifying that the temperature and energy consumption in a room is always within a pre-defined range.

We also make a number of other assumptions. Firstly, we assume that the structure of the system, i.e., the rooms and assets, is static. The verification of systems where components can be added and/or deleted autonomously at runtime is left as future work. Secondly, the mapping between requirements and specifications cannot be verified in our methodology. The requirement is the english description of the intended behaviour, whereas the specification is the formal processes encoding the intended behaviour. There is always the risk that the specification does not faithfully encode the requirement. It is the responsibility of the system designer to ensure that the specifications and requirements are close to each other as much as possible such that the mapping from the requirements to the specifications is clear. Lastly, we assume that requirements describe the intended behaviour of a small part of the system only and this enables the effective application of compositional reasoning that scales our verification approach. The effective verification of global requirements is left as future work.

1.7 Organisation of the Dissertation

The dissertation is structured as follows.

Chapter 2: In this chapter, we first overview the literature about SA systems, where we discuss related work regarding the design-time and run-time verification of SA systems. In the second part of the chapter, we present the main syntax and semantic models of the process language Communicating Sequential process (CSP) [70].

Chapter 3: We present a high-level model for centralized SA systems, which decouples adaptation patterns from the descriptions of base systems and adaptation procedure. A distinct automaton pinpoints when adaptation must happen. Using this framework system designers can experiment with different adaptation patterns, without modifying the base system or adaptation procedure to discover correct and efficient patterns. We provide an adequate translation into FDR and also prove that self-adaptive systems correspond to standard models of computation. We illustrate the use of our framework through a use case of a self-adaptive system of autonomous search-and-rescue rovers.

In the remaining parts of the dissertation, we present a methodology for compositional modelling and verification of SA CPS.

Chapter 4: We first overview our methodology to *compositionally* model and verify the satisfaction of security requirements in SA CPSs. In this chapter, we summarize the overall process and expanded on in subsequent chapters]in subsequent chapters we focus on individual steps. We also present the main syntax and semantic models for our novel process language, which we call the Adaptive CSP process language. The process language is inspired from CSP but extended with higher-order communication constructs to concisely model adaptation.

Chapter 5: We investigate how the topological layout and relations between components of a CPS guide a system designer to systematically explore different levels of granularities (i.e., grouping of components) for satisfying a requirement. We also present a systematic technique for encoding adaptation procedures that aims to ensure the satisfaction of a requirement. We motivate this technique through the art gallery example presented in section 1.2 by providing an encoding in Adaptive CSP.

Chapter 6 We show how the satisfaction of requirements can be verified using FDR by considering only a relevant subset of the components. In this chapter, we present an adequate translation where a small subset of the system encoded in Adaptive CSP is translated to CSP, the input language of FDR to verify the satisfaction of each requirement. We also present the theory of compositionality for our verification technique that guarantees verification tasks proven over small, disjoint subsets of CPS components hold for the entire system.

Chapter 7: In this chapter we discuss the verification of requirements when adaptation procedures are composed together. We extend our theory of compositionality results to investigate potential interference between different types of adaptation procedure overlaps. In some cases, interference is theoretically not possible—e.g., when adaptation procedures monitor, without changing, overlapping components—thus allowing us to skip re-verification tasks for the composition. We present a systematic approach to compose adaptation procedures, and verify where needed the relevant system requirements.

Chapter 8: We discuss the development of a tool that implements the translation from $ACSP_M$, a concrete syntax for Adaptive CSP, to CSP_M, the input language of FDR. The concrete syntax extends Adaptive CSP with functional programming languages constructs to improve the usability of our framework in realistic examples.

Chapter 9: We evaluate the applicability of our modelling methodology for SA CPSs and the practicality of the verification approach through a different case study, that of a smart stadium [92].

Chapter 10: We conclude the dissertation by summarizing the main results and briefly outline future research directions that would strengthen the contributions of the thesis.

Chapter 2

Background

2.1 Process Languages

Compositionality entails the divide-and-conquer or decomposition of complex problems into smaller more-manageable sub-problems. This techniques has been widely-utilized by the software verification community to achieve scalable verification. Process algebras like CCS, CSP or π-calculus have been proposed to compositionally model software and different verification techniques like refinement-checking [59] and behavioural equivalence [122, 44] have been proposed to compositional verify process languages. CCS [90], CSP [20] and π-calculus [108] were introduced to investigate small concurrent systems. The communication of behaviour cannot be concisely encoded in such languages. This motivated the introduction of a new class of process languages, known as higher-order process languages, e.g., CHOCS [119] and HOπ [109]. These languages are very powerful and expressive, but we lack tools to reason about higher-order processes. In this thesis, we identify a strict subclass of higher-order processes in our novel process language that can be encoded in the first-order process language CSP.

Here, we overview *Communicating Sequential Processes* (CSP) and refinement-based verification techniques.

$$\textsc{CHid}\ \frac{P \xrightarrow{e} P' \qquad e \in X}{P \setminus X \xrightarrow{\tau} P' \setminus X} \qquad \textsc{CTrue}\ \frac{b = \texttt{tt}}{\text{if } b \text{ then } P \text{ else } Q \xrightarrow{\tau} P} \qquad \textsc{CPar}\ \frac{P \xrightarrow{e} P' \qquad e \notin X}{P \underset{X}{\parallel} Q \xrightarrow{e} P' \underset{X}{\parallel} Q}$$

$$\textsc{CSyn}\ \frac{P \xrightarrow{e} P' \qquad e \in X \qquad Q \xrightarrow{e} Q'}{P \underset{X}{\parallel} Q \xrightarrow{e} P' \underset{X}{\parallel} Q'} \qquad \textsc{CInt}\ \frac{Q \xrightarrow{e} Q'}{P \triangle Q \xrightarrow{e} Q'} \qquad \textsc{CRdc}\ \frac{P \xrightarrow{e} P'}{P \triangle Q \xrightarrow{e} P' \triangle Q}$$

$$\textsc{CPrf}\ e \to P \xrightarrow{e} P \qquad \textsc{CChx}\ \frac{P \xrightarrow{e} P'}{P \,\square\, Q \xrightarrow{e} P} \qquad \textsc{CIcx}\ P \sqcap Q \xrightarrow{\tau} P \qquad \textsc{CEsc}\ \frac{P \xrightarrow{e} P' \qquad e \notin X}{P \setminus X \xrightarrow{e} P' \setminus X}$$

$$\textsc{CLet}\quad \text{let } X(\vec{y}) = P \text{ within } X(\vec{e}) \xrightarrow{\tau} P[\vec{e}/\vec{y}][\text{let } X(\vec{y}) = P \text{ within } X/X]$$

Figure 2.1: CSP Operational Semantics (omitting symmetric rules)

2.1.1 Communicating Sequential Processes (CSP)

CSP, introduced in 1978 by CAR Hoare, formally captures the notion of concurrent behaviour, where the low-level scheduling details of interleaving are abstracted over [20]. The main focus of the language is the communication between a process and its environment, which is defined in terms of events over a predefined alphabet. Here, we consider only a subset of the language. We let P, Q range over CSP Processes, x, y range over variables and e range over events. We let Σ represent the set of all events. defined as follows:

$$P, Q ::= e \to P \mid P \,\Box\, Q \mid P \underset{A}{\parallel} Q \mid \mathit{SKIP} \mid \mathit{STOP} \mid \text{if}\, b \,\text{then}\, P \,\text{else}\, Q$$
$$\mid \text{let}\, x(\vec{y}) = P \,\text{within}\, Q \mid P[\![e_2/e_1]\!] \mid P \triangle Q \mid P \setminus A \mid P \sqcap Q \mid x(\vec{e})$$

The prefix construct $e \to P$ means that the process P is guarded by event e. External (deterministic) choice $P \,\Box\, Q$ allows the environment to choose between synchronising with P or Q. In the case where both choices are enabled, the choice is resolved non-deterministically. The internal (non-deterministic) choice is written as $P \sqcap Q$. The processes $\text{let}\, x(\vec{y}) = P \,\text{within}\, Q$ and $x(\vec{e})$ represent the let and application constructs respectively, e.g.,

$$\text{let}\, x(y) = e_y \to x(y+1) \,\text{within}\, x(1)$$

The processes $\text{if}\, b$ then P else P, $P \setminus A$ and $P[\![e_2/e_1]\!]$ represent conditional, action hiding, and event renaming from e_1 to e_2. The process and let declaration. The processes *SKIP* and *STOP* represent a successful termination and the deadlocked process, respectively. Interleaving $P \underset{A}{\parallel} Q$ requires P and Q to synchronise on actions in the set A but requires no synchronisation on events not in A. An interrupt process, written $P \triangle Q$, propagates any event from P without affecting the interrupt, but if Q ever performs a visible event then this removes the interrupt and P, and the entire process behaves as Q.

In fig. 2.1, we outline the operational semantic rules for CSP. The rule CHid hides an event e from its environment; in rule CEsc such an event is propagated to the environment. The condition rule CTrue and its (omitted) symmetric rule evaluates a conditional; CPar and its (omitted) symmetric rule propagates an event e of P over a parallel composition $P \underset{E}{\parallel} Q$, provided $e \notin E$. The rule CSyn synchronizes such an event. In rule CInt, a process P is interrupted by a process Q; in rule CRdc the interrupt is propagated forward. CPrf annotates the transition by an event e, CChx can transition to either a process in P or Q. Finally, rule CLet unfolds a recursion by an internal transition, during which the formal parameters of the recursive process $\vec{y}$ are replaced by the actual parameters $\vec{e}$, and the recursive variable X is replaced with the recursive process itself.

$$\begin{aligned} &\text{let}\, x(y) = e_y \to x(y+1) \,\text{within}\, x(1) \\ \xrightarrow{\tau}\; &e_y \to x(y+1)[^1/_y][^{\text{let}\, X(y) = e_y \to x(y+1) \,\text{within}\, x}/_x] \\ =\; &e_1 \to \text{let}\, x(y) = e_y \to x(y+1) \,\text{within}\, x(1+1) \end{aligned}$$

From the parallel construct $P \underset{A}{\parallel} B$ we derive the alphabetized parallel construct. We write $P \;{}_A\|_B\; Q$ to mean P and Q can only perform the events in A and B respectively and they

synchronize on the set of events $A \cap B$,

$$P \ {}_A\|_B \ Q = \left(P \underset{Events \setminus A}{\|} STOP \right) \underset{A \cap B}{\|} \left(Q \underset{Events \setminus B}{\|} STOP \right)$$

If $ev(P) \subseteq A$ and $ev(Q) \subseteq B$ then the following statement holds,

$$P \ {}_A\|_B \ Q = P \underset{A \cap B}{\|} Q$$

2.1.2 Semantic Models

The three main semantic models of CSP are trace (`T`), stable failure (`F`) and failure-divergence (`FD`) models.

Trace Semantic Model (`T`) Let a trace be a potentially infinite sequence of events denoted by $t = \langle e_1, e_2, \ldots \rangle$ where $\langle\rangle$ is the empty trace. We say a process P has a trace $t = \langle e_1, e_2, e_3, \ldots \rangle$ written as $P \xrightarrow{t}$ iff it can communicate all the events in the trace in order i.e., e_1 followed by e_2 and so on. Formally, $P \xrightarrow{t}$ is the reflexive and transitive closure of $P \xrightarrow{e_1} \xrightarrow{e_2} \ldots$. We define the set of traces T for a process P as

$$traces(P) = \{t \mid P \xrightarrow{t} \}$$

The set of traces allows us to define the first semantic model for CSP. We say Q refines P by the trace model, written as

$$P \sqsubseteq_{\texttt{T(CSP)}} Q \quad \text{iff} \quad trace(Q) \subseteq traces(P)$$

This is useful for specifying safety properties: *nothing bad ever happens*, i.e., all the traces of implementation Q are in the traces of the specification P.

Failure Semantic Model (`F`) was introduced to verify liveness properties, where the violation is the absence of a suffix leading to a desired state. Here, we also check that both the process and the specification deadlock on the same state. Consider the following two processes where $STOP$ denotes a deadlock termination rather than a successful termination (i.e., $SKIP$),

$$P = \text{let } X = a \rightarrow X \text{ within } X$$
$$Q = (\text{let } X = a \rightarrow X \text{ within } X) \ \sqcap \ STOP$$

The trace model does *not* distinguish the two processes as both define the language a^*. However, the failure semantic model does distinguish the processes, because Q can deadlock on $STOP$ whereas P never reaches a deadlock state.

We define the set of refusal of a process P to be the set of events that are not enabled from process P. This allows us to define the set of failures of a process P. A failure is a pair (t, X) where t is a trace and X is a set of first-order events refused after P performs t.

$$refusal(P) = \{X \subseteq \Sigma \mid P \not\xrightarrow{a} \text{ and } a \in X\}$$
$$failures(P) = \{(t, X) \mid P \xrightarrow{t} Q \text{ and } X \in refusal(Q)\}$$

We say that P refines Q by the failure model iff the failures and traces of Q are contained in the

failures and traces of P

$$P \sqsubseteq_{\mathtt{F(CSP)}} Q \qquad \text{iff} \qquad failures(Q) \subseteq failures(P) \text{ and } traces(Q) \subseteq traces(P)$$

Failure-Divergence Semantic Model (FD) is the last semantic model we consider for CSP. The failure-divergence model also distinguishes between livelock and deadlock process. A process is livelocked or divergent if the process can perform an infinite sequence of internal events τ but no observable event.

$$Q_0 \Uparrow = \text{ for all } n \in N. \exists\, Q_{n+1}.\ ; Q_n \xrightarrow{\tau} Q_{n+1}$$
$$div(P) = \{t\,;t' \mid P \xrightarrow{t} Q \text{ and } Q \Uparrow\}$$

$$P \sqsubseteq_{\mathtt{FD(CSP)}} Q = failures(Q) \subseteq failures(P) \text{ and } div(Q) \subseteq div(P)$$

The advantages of all three semantic models presented above are that they are congruent and transitive. Congruence means that for any context C and semantic model $\mathtt{M} \in \{\mathtt{T},\mathtt{F},\mathtt{FD}\}$,

$$P \sqsubseteq_{\mathtt{M(CSP)}} Q \qquad \text{implies} \qquad C\,[P] \sqsubseteq_{\mathtt{M(CSP)}} C\,[Q]$$

while through transitivity, we infer that

$$Spec \sqsubseteq_{\mathtt{M(CSP)}} Q \text{ and } Q \sqsubseteq_{\mathtt{M(CSP)}} Impl \quad \text{implies} \quad Spec \sqsubseteq_{\mathtt{M(CSP)}} Impl$$

This allows us to have a potentially iterative verification approach from the specification down to the implementation also known as *stepwise refinement*.

2.1.3 Algebraic Laws

From [105], we list some of the algebraic laws founded in CSP, that are relevant to this thesis. Equivalence between two processes means that an external observer cannot distinguish between the two processes. This is known as strong bisimulation. In CSP, the notion of divergence is important especially for the failure-divergence semantic model and thus weak bisimulation is not applicable. In this dissertation, we refer to strong bisimulation as bisimulation.

Definition 2.1.1 (Bisimulation relation in CSP)**.** The relation $\mathcal{R}$ on CSP processes P_1 and P_2 is said to be a (strong) bisimulation iff $P_1 \mathcal{R} P_2$ and

1. If $P_1 \xrightarrow{a} P_1'$ implies there is a P_2' such that $P_2 \xrightarrow{a} P_2'$ and $P_1' \mathcal{R} P_2'$
2. If $P_2 \xrightarrow{a} P_2'$ implies there is a P_1' such that $P_1 \xrightarrow{a} P_1'$ and $P_1' \mathcal{R} P_2'$

$\Diamond$

Two processes are said to be bisimilar $P \sim Q$, iff there is a bisimulation relation $\mathcal{R}$ that related them $(P, Q) \in \mathcal{R}$. The relation $\sim$ is the largest bisimulation.

Definition 2.1.2. In later chapters, we assume the following list of equivalence laws for CSP

$$P = (P \underset{A}{\parallel} P) \qquad \langle \parallel - \texttt{Identity} \rangle$$

provided $ev(P) \subseteq A$

$$(P \ {}_X\|_Y \ Q) = (Q \ {}_Y\|_X \ P) \qquad \langle {}_A\|_B - \texttt{symm} \rangle$$

$$(P \ {}_X\|_Y \ Q) \ {}_{X \cup Y}\|_Z \ R = P \ {}_X\|_{Y \cup Z} \ (Q \ {}_Y\|_Z \ R) \qquad \langle {}_A\|_B - \texttt{assoc} \rangle$$

$$(P \ {}_A\|_B \ Q) \setminus Z = (P \setminus Z \cap A) \ {}_A\|_B \ (Q \setminus Z \cap B) \qquad \langle \texttt{hide} - {}_A\|_B - \texttt{dist} \rangle$$

provided $A \cap B \cap Z = \emptyset$

$$\left(P \underset{A}{\parallel} Q\right) \setminus Z = (P \setminus Z) \underset{A}{\parallel} (Q \setminus Z) \qquad \langle \texttt{hide} - \underset{A}{\parallel} - \texttt{dist} \rangle$$

provided $A \cap Z = \emptyset$

$\Diamond$

In Section 7.2 in the book [105], Roscoe shows that the relation $=$ is a strong bisimulation

2.1.4 Tools Available

A number of tools have been developed to reason about CSP processes. Here, we provide a non-exhaustive overview of some of the tools based on CSP:

FDR [59] is an automatic refinement tool for the machine readable dialect of CSP—CSP$_M$ [110]. CSP$_M$ has been introduced to encourage the use of CSP to verify real-world systems. CSP$_M$ extends CSP with constructs and shorthand idioms from functional programming languages, like Haskell and Miranda, to alleviate the complexity of encoding real-world processes. We can check that an implementation refines a specification according to a semantic model. We can also verify if an implementation is deterministic, deadlock free or livelock free (divergence free). We use this tool in subsequent chapters to verify our examples.

Process Analysis Toolkit (PAT) [117] is a framework for reasoning about concurrent processes in CSP, probabilistic concurrent processes (probabilistic CSP) and real-time concurrent processes (Timed-CSP). PAT implements model-checking techniques to verify deadlock-freedom, refinement-checking, divergence-freedom akin to FDR. Moreover processes are model-checked against LTL properties [117] in PAT.

CSP Prover [73] is a theorem-prover for CSP processes utilizing the theorem-prover Isabelle [97] for reasoning about infinite state-space processes.

2.2 Design of Self-Adaptive Systems

Self-adaptation is a system's capability to autonomouslydetect when its operational environment changes and deploy counter-measures to guarantee the continued satisfaction of requirements in spite of changes, e.g., opening all the doors to and from rooms in the art gallery in the case of an emergency. The importance of exploded in recent times due to the wide range of knowledge and devices' versatility that systems need to manage. Problems that are widely researched regarding SA systems that are relevant to this thesis include: 1. the verification of SA systems 2. engineering processes to model SA systems [42, 127].

An CPS is a system where the behaviour is determined by both the digital processes and the physical aspects of the environment in which it operates. They rely on self-adaptation to satisfy requirements.

The definition of a self-adaptive systems is not precise. Cheng et al. define a self-adaptive system as a system that is able to adjust its behaviour in response to their perception of the environment and the system itself [32]. Brun et al. [21] argue that the *self* prefix indicates that the system decides autonomously (i.e., without or with minimal interference) how to adapt or organise to accommodate changes in its context and environment; whereas Esfahani et al. emphasise the "uncertainty in the environment or domain in which the software is deployed as a prevalent aspect of self-adaptive systems" [51].

A widely accepted view of SA systems is that such systems comprises at least two modules: the *base system* and the *adaptation procedure*. The *base system* represents the systems core functionality that is adapted autonomously at runtime, whereas the *adaptation procedure* localizes all adaptation functionality. The adaptation procedure is sometimes referred to as *adaptation manager* or *control loop*. In this dissertation, we use the term adaptation procedure to refer to the module in the system that implements the adaptation functionality. A prominent well-established architecture to model an adaptation procedure is known as the MAPE-K feedback loop. [75, 36] The MAPE-K feedback loop is a closed feedback loop comprising the following four steps: 1. *Monitor* the base system and context for changes 2. *Analyze* whether a change requires an adaptation (of the base system behaviour) 3. *Plan* the adaptation 4. *Execute* the adaptation, under a knowledge base K containing the contextual information. The *environment* or *context* contained in K refers to all the information we know about the system augmenting the systems state. This includes logging information, system topology and resources or assets status. One can view the environment as the part of the system that cannot be directly altered by the adaptation procedure but can only be monitored. A change in the environment may still require an adaptation of system's behaviour, e.g., a breakdown in a resource would need a fail-over procedure, which is implemented through self-adaptation.

In fig. 2.2, we present a taxonomy of the characteristics of adaptation procedures presented in [82].

Adaptation Triggers As summarized in [82], adaptation can be either reactive or proactive. In reactive adaptations, the adaptation procedure reacts to events or changes after they happen, e.g., [10], whereas in a proactive approach, the SA system aims to prevent the occurrence of threats or change e.g., [93, 94, 120]. The advantage of proactive adaptations is the absence of interruption for users because adaptation is performed before it is actually needed. This also gives the system more adaptation options. On the other hand, due to its predictive nature, proactive adaptation is more complex to implement in comparison to reactive adaptation and may result in suboptimal system behaviour as a side-effect of minimizing threats. An SA system may opt for a combination of proactive and reactive adaptation. The difference between the adaptation triggers is localized in the Analyse step within the MAPE feedback loop.

Change This distinguishes between different types of adaptation. Adaptations fall within three camps: behavioural, structural or contextual change. A behavioural change implies a change in the system behaviour. Here, an adaptation procedure communicates new behaviour to the base-system. Structural adaptation implies the addition and/or removal of components at runtime; and in contextual adaptation, the change is executed on the context itself. Even though the monitoring of the context is a fundamental step in the MAPE-K feedback loop, changing the context as an

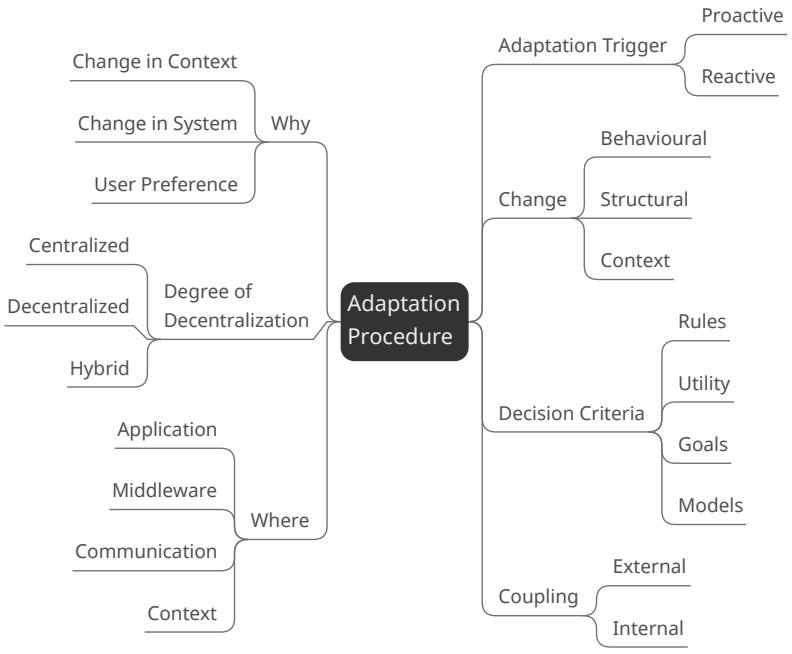


Figure 2.2: Design decision for engineering a self-adaptive systems from [82]

adaptation option has not been widely investigated. In this document, we focus on behavioural changes, where the adaptation procedure communicates algorithms or system behaviour to different parts of the base-system. Even though our process language can potentially encode structural changes, our verification approach cannot handle cases where components are added or removed dynamically. The modelling and verification of structural and contextual changes is left as future work.

Decision Criteria These criteria help a system designer understand adaptation—why and how a system adapt. A system designer decides the adaptation outcome, either through a utility function e.g., [95, 28, 29, 71, 55, 94], a set of rules or goal model e.g., [33]. In this document, we do not address this criteria. It is left up to a system designer to implement the adaptation decision process.

Where Here, we identify the level at which we implement the self-adaptive capabilities e.g., adaptation can be implemented at the network level or application level. These criteria does not apply in our thesis, because we do not implement the SA system.

Why Adaptation is an approach to ensure the continued satisfaction of requirements in the face of a changing operational environment. Identifying the source of possible changes is crucial for the monitoring and planning phases of the MAPE-K feedback loop. A change can be a change in the environment (events out of the system's control), a change in available resources (e.g., a server goes down) or a change in user preferences. For the monitoring phase, foreseeing potential changes guides the system designer to identify what needs to be monitored; whereas for the planning phase the type of change guides the adaptation decision. Thus, an SA system needs to adjust its behaviour to satisfy the requirement when the change occurs.

Coupling Two approaches are identified for implementing adaptation procedures. One approach is to have the adaptation functionality intertwined with the system functionality and another approach is to modularize the adaptation functionality. The former is referred to as an internal adaptation procedure whereas the latter is known as an external adaptation procedure. One major disadvantage of the internal approach is that adaptation becomes a cross-cutting concern that may impede the scalability of the system. The lack of separation of concerns makes the potential system also hard to understand and maintain. In our approach, we encode external adaptation procedures.

Degree of decentralization Another aspect of adaptation procedures is the degree of decentralization. In a centralized SA system, there is one adaptation procedure that implements all the adaptation functionality. Such adaptation procedure tends to have a complete view of the system to determine the most optimal adaptation. This suffices for small-systems, however for large-systems, a centralized adaptation procedure becomes a bottleneck and a single point of failure. Therefore a decentralized approach can improve scalability and reliability, where the adaptation procedure is decomposed and deployed at the level of each component. For CPSs, adaptation may need to be implemented over a group of components, making a hybrid of the two approaches more desirable. Our encoding is flexible enough to cater for all three cases. We allow the adaptation procedure to be placed around different grouping of components.

The website [2] contains a collection of examples of SA systems and model problems that have been used extensively within the community. Here, we overview a non-exhaustive list of SA systems found in the literature. Garlan et al. in [57] introduces znn.com which simulates a self-adaptive web server of a news site. Camara et al. in [29] and Moreno et al. in [94] evaluate their frameworks using a web-server as a case-study. Another case-study widely used in the literature is the traffic routing problem introduced in [129]. Authors in [85, 88, 50, 123, 24] evaluate their frameworks through a variant of the traffic routing problem. Related to transport, SA systems implementing unmanned autonomous vehicles (UAV) introduced in [77] have been used in the evaluation of [23, 16, 112, 78, 99, 128]. Smart spaces like warehouse, healthcare centres and buildings have been studied extensively as case-studies for SA systems in [48, 121, 120, 52, 128]. Assisted living and the applicability of self-adaptive capabilities in healthcare software systems have been studied in [27, 95, 104, 116].

2.2.1 Decentralizing the Adaptation Procedure

Today systems are expected to work in an open environment. For SA systems, this means that adaptation procedures do not have a complete view of the context to plan the most optimal global adaptation. Adaptation decisions now target a small part of the system and checks are put in place to make sure that adaptations satisfy both local and global requirements. This class of SA systems motivates the drive for systematic engineering techniques for building decentralized SA systems. Even though models for SA systems supporting compositionality are the norm [85, 22, 6, 84, 78, 48]. Such literature neglects to propose techniques to attain a compositional model or address explicitly the interference caused between different adaptation procedures in a decentralized SA systems [127, 126, 42].

Weyns et al., in [126], summarise the key challenges for designing decentralized SA systems. The challenges are motivated through two case-studies—a traffic monitoring application and a quality of service optimization system. The challenges are grouped under five main umbrella terms:

Uncertainty In a cyber-physical environment, the behaviour of physical components is unpredictable. As a result the system behaviour may become inconsistent with the environment intermittently. The volatility of SA systems is increased due to the lack of a centralized adaptation procedure that has a complete view of the environment.

Overhead Decentralized adaptation procedures need to co-ordinate with each other to ensure that global requirements remain satisfied. The co-ordination introduces additional overheads in terms of computational efficiency that an SA system may not be able to tolerate.

Conflicting Goals The outcome of adaptation procedures, trying to optimize the behaviour of local components of a CPS, may create interference at a global level. For instance, an adaptation procedure may hog resources from other adaptation procedures or the sub-system behaviour may conflict with other sub-systems behaviours. Approaches to reason about global goals and adaptation procedures cooperation in a decentralized systems are still in demand.

Systematic engineering Due to the inherent complexity, design patterns for modelling coordination between adaptation procedures are needed.

Guarantees The challenge for verifying decentralized SA system is increased merely as a by-product of the above challenges.

In our work, we tackle the last three challenges. We present a modelling process for decentralized SA CPSs that aims to minimize the coupling between adaptation procedures. We also present a verification approach to guarantee that adaptation procedures ensures the satisfaction of local requirements without interfering with the satisfaction of global requirements. We showcase that our verification approach can be applied to realistic SA CPSs through two case-studies.

Weyns et al., in [127], consolidate structural design patterns for decentralized SA systems. The authors also present guidelines for choosing a design pattern based on system priorities. For instance, for a self-optimizing system, a centralized approach may be the most appropriate as adaptation can be determined with a complete view of the system; for scalable SA systems the choice of design pattern depends on the volume of information that needs to be shared between adaptation procedures. The system designer must strive to find a balance between decentralizing the adaptation procedure and minimizing the communication costs between the adaptation procedures. In a robustness-oriented SA system, a decentralized adaptation procedure removes the bottleneck created by having a single centralized control loop and checks are put in place to guarantee the continued satisfaction of requirements when one of the adaptation procedures cease to function. Efficient response tends to favour decentralized adaptation procedures, however this may result in sub-optimal responses due to the partial view adaptation procedures have of the context. Abuseta et al., in [5], present a design pattern for each step in a MAPE-K feedback loop for decentralized SA systems. The design patterns target existing systems adding self-adaptation functionality. The approach is motivated using UML – a semi-formal modelling approach that may not be suited to verify behavioural properties.

Hachicha et al. [65, 66], propose design patterns for designing decentralized adaptation procedures in Event-B. The work defines two composition operators for overlapping and non-overlapping adaptation procedures. The authors also overview how different patterns for SA systems (e.g., slave-masters, hierarchical presented in [127]) can be composed to preserve the compositionality. They focus on structural adaptation whereas in this thesis we focus on behavioural adaptation.

Moreover, the composition guides a system designer to building SA systems bottom-up with scalability and compositionality not being the main concerns. Our model aims to achieve a tractable verification using existing techniques and thus compositionality within the model is of the utmost importance.

Calinescu et al. present the DECIDE framework—a runtime quantification verification-driven approach for decentralized SA systems [26]. The authors evaluate how runtime quantitative verification can support the dynamic decentralization of adaptation functionality and provide a practical verification approach for SA systems. In our approach, we focus on design-time verification and our decentralization approach is static determined manually by the system designer.

Numerous works to decentralize the adaptation procedure have been proposed from the distributed systems community e.g., based on multi-agent approaches [46, 56, 118] or service-based systems [96]. These works aim to alleviate the task of implementing the planning phase of a decentralized adaptation procedure. This goal is different from ours, as our aim is to verify the overall correctness of SA systems with decentralized adaptation procedures.

2.2.2 Topology-Aware Self-Adaptive Systems

Compositionality is a necessity when modelling and verifying SA CPSs. An approach for decomposing CPSs is system *topology*. By the topology of CPSs, we mean the structures and characteristics of both physical and digital components [102, 120]. Topology is the study of structures and spaces and how different spaces are connected [67]. A space can either be a physical entity like an agent, container or a digital entity like a connection point, the layout ofaccess domains to valuable information. The connections we consider in this document include 1. connectivity which describes how two spaces are connected e.g., rooms may be connected by a stairs or a door. This allows us to define other properties like 2. proximity, how quickly can one transition from a space to another space, and 3. reachability, is there a connection between the two spaces. 4. We also consider containment; for example rooms are contained in floors, and access domains may be contained in other access domains.

Example 2.2.1. Consider the art gallery from section 1.2. Here the topology consists both of physical and digital entities. The physical entities are *rooms A,B,D* in the exhibition area, connecting rooms like the *corridor 1* and the *stairs* and the upper floor consisting of the *restoration area* and *computer room*. The doors between the rooms represent a connectivity relation between the rooms. In the art gallery example, we can also define containment relations e.g., *rooms A,B,D* are contained in the *exhibition area* and in turn the *exhibition area* and *corridor 1* are contained in the *lower floor*. Containment aids us to decompose the system, first into two floor, then the floors into different sections and sections into rooms. From the topology we infer that a visitor may go from *room D* to *room B* in at least three steps: *room D* to *corridor 1*, then to *room A* and finally *room B*. : the closer an unauthorized user is to an asset, the higher the probability of a violation.]

The art gallery also has a digital *access point* that the *HVAC component* connects to. We define a digital connectivity between the two components. We can also define containment relation where the *HVAC* is contained in the *restoration area* and the *access point* is contained in the *computer room*. Note how CPS blurs the boundary between the digital and physical worlds and how through topology we can define the same relations for both physical and digital components and naturally connect digital elements with physical components. ◊

Topological knowledge has been applied by software engineering communities to distributed systems [37], network structure [124] and engineering SA systems [120, 102, 121, 103]. CPSs rely

on the interplay between physical and digital components to achieve its goal. Because the same topological relations apply to both physical and digital entities, Pasquale et al. overview how topology provides a natural composition for CPSs [102]. Topological changes, like the movement of agents or assets, may introduce or eliminate a threat to the satisfaction of a requirement and an adaptation may be needed to react to the new threat. Topology can guide a system designer in identifying changes that require adaptation [83].

2.3 Verification of Self-Adaptive Systems

Verification techniques for SA systems may be classified as: design-time or run-time verification. The main challenge in the former is trying to predict the whole state space for all the changing components, while runtime verification, even though the verification process focuses on the state the system is in at that point in time, the verification process has a tight deadline to return results. In this thesis, we understand runtime verification to mean heuristics based on runtime verification techniques to infer an adaptation that ensures the satisfaction of a specification, rather than the standard definition of runtime verification where we check that the execution of our application satisfies a specificationf. Thus, both approaches aim to reduce the search space of verification to to be effective. Here, we overview the main literature regarding both design-time and run-time verification of SA systems and how the complexity of verifying SA systems is addressed.

2.3.1 Design-time verification

The objective of design-time verification is to provide guarantees before a system is deployed and as such the verification process needs to predict and verify all systems states. For an SA system, such a state space may be huge and as stated earlier, this may lead to verification being computationally infeasible. Techniques have addressed this problem through compositional frameworks.

CSP-based approaches [62, 10, 63] discuss how CSP can compositionally model SA systems. Göthel et al. [62] overview how to model different design patterns for SA systems using CSP. Bartels et al. [10] study how SA systems can be modelled using CSP. The framework explicitly separates adaptive and non-adaptive behaviour. The authors discuss the limitation of using CSP because CSP does not support dynamic processes. The authors simulate dynamic process creation by identifying all dynamic processes and putting a guard around such processes. The activation of the process is encoded by setting the appropriate guard to true at runtime. This work is extended by Göthel et al. [63] to include the notion of time and temporal dependencies. Our process language supports dynamic processes, whereas for our verification approach we identify a subset of the processes where the dynamic behaviour can be modelled in FDR. We guard dynamic processes by distinguished CSP events instead of boolean expressions. Our work focuses on attaining compositional verification, which the above works do not address. Moreover, our framework enables the effective exploration of adaptation procedures, which can be a valuable tool for designing and verifying large-scale SA CPSs.

Session types and assume-guarantee reasoning have been used to model adaptation in SA systems [15, 115, 39]. Bono et al. [15] discuss how global session types can implement adaptation within a system. This provides a level of modularity for adaptation patterns, similar to our work. However, adaptation is modelled as a fine-grained communication filter rather than behaviour modification. It is unclear what class of properties can be enforced with this technique. Our approach

is different as it is based on a more natural encoding of self-adaptation with higher-order communications, giving us more flexibility to encode systems. We also use scoped locations, instead of session types to achieve compositional reasoning, which enables us to leverage existing verification tools such as CSP. Li et al. [98] extend Schroeder et al. [115] to model SA systems compositionally using a probabilistic assume-guarantee framework. This requires special-purpose verification tool which is still to be developed [40, 72]. Our framework leverages existing verification tool – FDR. However, it would be interesting to explore possibilities of combining the two frameworks.

Automata-based Bruni et al. [23] use Maude to model each step in the MAPE-K feedback loop as an abstract state machine (ASM), which can be model-checked using PVesta – a statistical model-checker. Iftikhar et al. [85] use timed automata and timed computational tree logic (TCTL) to model decentralized SA systems and specify temporal safety and liveness properties. These properties are model-checked using Uppaal model-checker. Klarl et al. [79] present hierarchical LTS (H-LTS) which can be model checked with the SPIN model-checker and translated automatically into Java code. Zhao et al. [132] model SA systems using mode-automata and define mode extended LTL (mLTL); an extension of LTL with context-dependent formulas for the specification of systems. The semantics of mLTL is derived from that of LTL. Zhao et al. used the NuSMV model checker as a verification tool. Jalili et al. [3] explore a method to model and verify at runtime decentralized SA systems based on the HPobSAM framework, taking advantage of decentralisation to achieve compositionality. In these works, the ability to decompose systems into independently verifiable components is limited or non-existent. Our work provides a structured method to do this, when requirements allow it, even in systems with components that are intricately linked with cyber and physical relationships. In the automata-based models mentioned above adaptation events are hard-coded in the system automaton. A change in the adaptation pattern requires potentially significant change in the automaton. In contrast, our modelling methodology allow us to model the system automaton by specifying *may*-adapt events at system states where adaptation is possible. Through composition with a separate adaptation automaton the precise adaptation pattern is selected. This allows us to model sophisticated patterns such as time-triggered adaptation patterns. More importantly, by localising the adaptation pattern, we can experiment with different patterns without having to change the main system. Moreover, we present a modelling verification methodology that through topology alleviates the task of verifying SA CPS. These work do not tackle the complexity to systematically derive a compositional model and potential interference cause by overlapping adaptation procedures working on the same part of the CPS.

Higher-order process languages with passivation Passivation has been introduced in process calculi to enable the encoding of complex distributed systems with components that can be stopped and resumed [113]. Bavetti et al. [19] propose the $\mathcal{E}$-calculus, a higher-order calculus inspired by the Calculus of Communicating Systems (CCS) without restriction and renaming. A process P which is adaptable and located at a, is denoted as $a\,[P]$. The environment can at any time communicate a context to a, installing the context at the a-location. This context may have holes which are then filled by copies of P. This results in a highly dynamic language; verification in this language would require new special-purpose techniques and tools to be developed. Moreover, the absence of location restriction (scoping) severely limits compositional reasoning in this language. Our framework is based on a more tamed higher-order language, which enables a translation to existing verification tools based on first-order languages, as we have shown with our translation to FDR. We make use of location restriction which allows us to consider parts of the system where all locations are locally scoped. Such parts can be verified independently from the rest of

the system, thus enabling useful compositional verification. We also utilise CSP-style multi-party synchronisation rather than CCS binary communication, to improve the separation of adaptation procedures, which must monitor the events in the system, from the system itself. Bavetti et al. studies the (un)decidability of verifying safety and liveness properties with sub-variants of the language. In contrast to these works, our framework is designed to encode adaptation patterns with *must*-adaptation events. Moreover, we show that we can translate our encoding of CPSs to FDR (and we envisage to other established tools), taking advantage of existing and future verification tools.

Other process languages Debois et al. [45] define the DCR process language, a Turing-complete declarative process language to model and verify runtime adaptation in a modular fashion. They define a non-invasive adaptation in a decidable fragment of the language. An adaptation is non-invasive if a new process is spawned during the adaptation. This sub-language is able to represent systems where a new resource or new condition is introduced during adaptations. Lochau et al. [84] define DeltaCCS, an extension of CCS to explicitly model behavioural variability in processes. The authors also implemented a DeltaCCS model checker to verify the processes. In our work, the process language is based on CSP-style synchronisation to achieve a better separation of concerns as base system components are oblivious to the adaptation procedures that may affect them. Moreover, our process language, based on higher-order communication can naturally encode a larger class of systems.

PetriNets Zhang et al. [130] model SA systems as a collection of petri nets. Each petri net has a single initial and final state. An SA system transition between petri nets through these states. This work has been extended in [30] to incorporate temporal constraints by considering the time-based petri nets [14]. In a similar fashion, Context petri Nets (CPN), introduced by Cardozo et al. [31], allow dynamic reconfiguration of petri nets to model adaptation; CPN can be then translated automatically into petri nets. Ding et al. [47] explain how neural networks can be utilised to implement the adaptation function wiring petri nets together. petri nets are not easily decomposable, and thus compositional verification is hard to achieve. Moreover, they do not provide a fertile ground to explore alternative adaptation procedures at different granularities of the system, as precise adaptation procedures are hard to encode and modify independently of the base system model.

2.3.2 Runtime verification

Runtime verification techniques has been proposed as a heuristic to automatically derive adaptation outcomes on the fly. The goal is to move the implementation of SA systems to runtime as much as possible. A (possibly partial) system model is generated automatically at runtime to guide the adaptation process . Here we overview the main techniques investigated for runtime verification of self-adaptive systems.

Model-checking exhaustively checks that a model conforms to a temporal logic specification. Runtime model-checking techniques proposed in [38, 88, 85, 61] investigate how model-checking can be utilized at run-time to verify adaptation results. Tsigkanos et al. [120] use bigraphical reactive systems [91] to represent topological relationships of cyber-physical systems. They apply explicit state model checking to reason about security threats brought by changes in the topological relationships that may occur at runtime. They also provide an automated planning technique to identify security controls to prevent or mitigate discovered threats.

MDP incorporate knowledge about the probability of event occurrences. Probabilistic model-checking allows us to design a trade-off between accuracy and performance of the verification process [95, 28, 29, 71, 55, 94]. Filieri and Tamburrelli [54] describe a number of strategies (i.e. state elimination [41] and algebraic approaches [53]) for using probabilistic model checking at runtime when the system behaviour is expressed as a Discrete Time Markov Chain (DTMC). These strategies can only be applied when changes in the system representation can be expressed as different assignments to its parameters (e.g., different probabilities for the transitions of the DTMC) and cannot cope with changes in the structure or behaviour of the system and its operating environment. Sensitivity analysis guiding the adaptation process to determine which adaptation leads to a more resilient and robust solution has been investigated in [29, 55, 85] . The logic Timed Computational Tree Logic (TCTL) has been proposed to specify requirements to enable the reasoning about timing constraints in [9, 132, 85]. Balasubramaniyan et al. in [9] also discuss how probabilistic model-checking can handle multiple objectives. Filieri et al. in [54] investigate the performance of different probabilistic model-checkers to be used as a runtime tool to verify SA systems. Our approach, unlike all these approaches is intended for design-time verification and we focus on decomposition techniques for SA systems to optimize the verification process. Calinescu et al., in [25], present the ENTRUST framework—a methodology for the systematic engineering of self-adaptive systems with assurances. Their framework leverages existing design-time and runtime probabilistic verification to ensure the satisfaction of requirements in a changing operational environment. They outline a set of generic properties that adaptation procedures should possess (e.g., deadlock freedom) and propose a pattern for modelling SA systems that supports the synthesis of adaptation procedures at runtime. The framework leverages the existing verification tools UPPAAL and PRISM to verify the correctness of synthesized adaptation procedures. Johnson et al. in [74] present the INVEST framework for the compositional reverification of component-based systems. The framework augments existing assume-guarantee verification approaches with the ability to reverify results when the structure of components changes with minimal efforts. The framework focuses on the verification of probabilistic safety properties using PRISM. Akin to our framework, the INVEST framework outlines a modelling methodology that enables the compositional verification of properties with existing verification tools. However, we motivate our methodology using a design-time verification tool instead of a probabilistic run-time model checking tool and we exploit the topology of CPSs to attain a compositional model.

UML has been utilized to model structural adaptation and has been studied in [60, 131, 103, 68, 7]. Goldsby et al. [60] propose AVIDA-MDE, a model driven engineering tool that generates UML of the desired target system. The tool can be utilised to implement advanced adaptation functions. Zhao et al. [131] outline how UML can be model-checked at runtime against RT-OCL specifications to implement self-optimisation. Pasquale et al. [103] present SecuriTAS – a tool for deploying adaptive security measures based on goal models that are updated at runtime. Unlike this work, we focus on design-time verification. Moreover, we model adaptation patterns independently from the main system, which allows easier exploration and verification of alternative patterns. Hebig et al. [68], explain how advanced feedback loops can be modelled with UML to expose problematic architectural structures. Almorsy et al. [7] model system components and features using UML and utilise refinement and aspect-oriented programming techniques to infer the security control that must be enforced at runtime. These models focus on structural adaptation.

2.4 Other Related Work

In this section, we compare our work with the closest literatures [10, 19, 120, 74].

Bartels et al., in [10], present a CSP framework for the verification and implementation SA CPSs. The authors present a refinement-based approach for specifying, verifying and automatically implementing SA systems. An SA system is encoded as the parallel composition of adaptable components. An adaptable component AC_i comprises a set of boolean guards $G_i = \{g_{i1}, \ldots, g_{im}\}$, adaptation events $AE_i = \{ac_{i1}, \ldots, ac_{im}\}$, such that the event ac_{ij} implies that the component AC_i requests an adaptation on AC_j. The CSP processes C_{ik} represent the system behaviour for component i that is enabled when guard g_{ik} becomes true. The adaptation is determined by communicating with an event ac_{ij} values that would set one of the boolean guards to true. An adaptable components is encoded as

$$AC_i(s) = \Box_{j \in \{1 \ldots n\}} ac_{ij}?x \rightarrow \left(\Box_{k \in \{1 \ldots m\}} \Big\{ g_{ik}(s,x) \,\&\, (C_{ik} \,;\, AC_i(s)) \right)$$

The verification is split into two parts. First, a system designer verifies that the encoding conforms to a family of adaptation specifications (AS), which check that the system does not unintentionally diverge or deadlock by utilizing the failure-divergence semantics,

$$AS \sqsubseteq_{FD} (AC_1 \parallel \ldots \parallel AC_n) \setminus Events$$

Then the system designer verifies that the encoding refines the functional requirements of what the system should do.

This framework has a number of differences from ours. Firstly, adaptation is realized by setting the appropriate parameters—parameter adaptation whereas we extend CSP with higher-order communication to express adaptation as the communication of behaviour—behaviour adaptation. In our framework, we also distinguish between a base-system and an adaptation procedure. In our modelling methodology, we aim to localize adaptation procedures to a small part of the system to improve compositionality and thus the scalability of our verification approach. In Bartels et al. framework, adaptation is a cross-cutting concern intertwined within the base-system. In fact, to verify the adaptation specification, the framework has to hide system events and vice versa to verify functional specifications the framework hides adaptation events. Moreover, adaptation is reactive and is only triggered if a component becomes idle. In our framework, adaptation can occur intermittently, a process is pre-empted and replaced by the communicated process from the adaptation procedure. This allows us to model more sophisticated adaptation triggers that are more responsive to the unpredictable behaviour of CPSs.

Tsigkanos et al., in [120], present a topology-driven framework to synthesize adaptation at runtime. Similar to our approach, their model is built based on the topology and topological relations, where a topological change, e.g., agent movements between rooms is a transition. However, their modelling framework utilizes bi-graphs and graph transformation semantics, where transitions have attached costs that are decided by domain experts. Adaptation is triggered on every transition. The main focus of the work is on the planning step in the MAPE-K feedback loop. The adaptation decision processes entails a novel threat analysis to synthesize adaptation. The threat analysis starts with the most permissive model of the system that allows all behaviour and performs an exhaustive (bounded) search to prohibit actions that directly transition to a violating state. A state is violated if it does not satisfy a CTL formula specifying the system behaviour. The system reuses the threat analysis between adaptations to improve the efficiency of the adaptation process.

Our work took a lot of inspiration from this work. In particular, both models are guided by the topology and topological relations. However, our goals are to exploit the topology of CPSs to localize adaptation procedures to small parts of the CPS to attain a tractable verification approach for SA CPSs. We motivate our approach through a design-time verification technique. Moreover, the adaptation decision process in our framework is left up to the system designer. Our encoding of adaptation procedures enables exploration of execution points where an SA CPS *must*-adapt. This technique enables a system to skip the threat analysis where adaptation is clearly not needed.

Johnson et al. in [74] present the INVEST framework that extends existing probabilistic assume-guarantee verification approaches with the ability to identify the minimal set of components that needs to be included in the reverification step after a system change and execute the verification at runtime. The framework presents a set of operations on components that triggers adaptation. The operations include component addition, removal and modification. These events guide the tool to identify the components that need to be included in the reverification steps to reduce the state space of the verification. The applicability of the framework is evaluated though a cloud-deployed software service case-study. Akin to our methodology, the authors leverage existing compositional verification techniques to effectively verify SA CPSs, with techniques that exploit compositionality proposed. In comparison, we motivate our work with a design-time verification technique where we preclude probabilistic knowledge. Moreover, we exploit the topology of CPSs and different adaptation procedures scope overlap types to reduce the search state in verification tasks. Both work present a verification process. In our verification process, we first verify that an adaptation procedure in isolation satisfies a requirement and then verify that the composition of adaptation procedures preserves the requirement. In [74], an incremental verification process is presented to verify each adaptation at run-time.

Bravetti et al., in [19], define a novel process language $\mathcal{E}$-calculus to address the limitations of standard process calculi (e.g., [89, 70]) for modelling self-adaptive behaviour. The process language extends CCS with higher-order communication to express adaptation capabilities. Akin to our process language, the construct $l\langle P\rangle$ represents an adaptable process, where other processes can communicate adaptation over the name l. An adaptation is communicated through the event $\tilde{l}\{U\}$ where l is the location and U is a context that has zero or more holes $\bullet$. The synchronization of the adaptation prefix $\tilde{l}\{U\}$ with the named process $l\langle P\rangle$ evolves to $U\{\{P\}\}$, where the process P replaces the holes in the context. This is more expressive than our process language as the process communicates a context rather than a process. Our adaptation can be encoded in $\mathcal{E}$-calculus when U does not contain any holes and therefore U is a closed process. The process language $\mathcal{E}$-calculus, however, does not support restriction of names needed to capture compositionality.

The authors investigate the decidability of two verification problems for different subsets of the language. In particular, they investigate the decidability of *bounded adaptation*, where they check that there is not an infinite amount of consecutive adaptations; and *eventual adaptation*, that an adaptation always returns. They investigate the decidability of the two verification problems for six subsets of the language: $\mathcal{E}_d, \mathcal{E}_s$ where d stands for dynamic topology where names can be created and destroyed dynamically and s stands for static topology where the creation and destruction of names is not permitted. $\mathcal{E}_d^2, \mathcal{E}_s^2$ does not allow holes in the context for both dynamic and static topologies and $\mathcal{E}_d^3, \mathcal{E}_s^3$ where exactly one hole is allowed in the context, and thus adaptation can only extend the functionality of the named process. Our process language is closest to $\mathcal{E}_s^2$. The authors show that for $\mathcal{E}_d^2, \mathcal{E}_s^2$ bounded adaptation is decidable but eventual adaptation is not.

The goal of the work is different from ours as we identify a subset of the process language for which higher-order communication can be encoded in first-order process languages. We also propose a *verification process* to tractably verify SA CPSs.

Chapter 3

Abstract View of Self-Adaptive Systems

A system designer of an SA system must decide at which system execution points should adaptation be invoked in order for the system to operate efficiently and satisfy its requirements. These points can be as simple as timing events (e.g., every X seconds), can be triggered by specific system events, or even have a complex logic taking account the execution history and state of the system.

Existing modelling approaches for SA systems often distinguish the adaptation decision process from the base system, but encode adaptation event points directly in one, or both, of these components (e.g., [10, 64, 79, 47, 132, 115, 77]). The drawback of this approach is that when searching for a correct and optimal model, modifying the pattern of adaptation events requires significant changes to the description of the base system and/or the adaptation decision process. This is because the pattern becomes a cross-cutting concern which is scattered within the model of the system.

The modularization of the adaptation patterns is not easy [58, 4]. Although some systems may be able to support such events at every state, others must disallow them during critical sections of their execution. Moreover, adaptation decision processes may be able to operate only at a subset of the system states. For example, a collision avoidance system of autonomous vehicles, realised as an adaptation decision process, may assume that it is invoked at states where vehicles are not too close to each other, where reasonable direction changes can avoid collisions. These aspects must be taken into account when adaptation patterns are modelled or implemented, and the maximum degree of freedom for choosing these patterns should be allowed in each use case.

We propose an abstract model for self-adaptive systems, which we call Self-Adaptive Automata (SAA). With this automata-based model we abstract away as many implementation details as possible of self-adaptive systems and focus only on the composition of three main components: base system, adaptation decision process, and adaptation pattern. This allows us to explore the design space of these components, keeping the others constant. In particular we can use this model to identify efficient adaptation patterns in self-adaptive systems. Our model is also useful in proving system correctness (and thus the correctness of adaptation patterns) against requirements. We provide an adequate translation from SAA to FDR [59], where we can verify system properties. This high-level model can be used as an abstract description of self-adaptive systems, where key system aspects such as adaptation patterns can be explored, and key requirements verified. From that, a correct and efficient implementation could be derived through refinement, leveraging FDR's refinement infrastructure.

In SAA, adaptation events are exposed in execution traces by a novel, special-purpose $\star$-transition. The effect of a $\star$-transition is the modification of the automaton's entire transition function (a modification of its behaviour), according to the adaptation decision process, encoded as a partial function within the model. This can capture *may-* and *must-adapt* events. We use may-adapt events in the modelling of base systems, exposing the states where the system can invoke the adaptation decision process. These events are optional because other transitions from the same states allow the system to skip adaptation. Must-adapt events are used in the definition of adaptation patterns, which are distinct automata—potentially implementing complex logic—with states whose only outgoing transition is an adaptation ($\star$-transition). Composition of a system model with an adaptation pattern model synchronises adaptation events, in effect enforcing the adaptation pattern on the system. The benefit of this approach is that we can change the adaptation pattern without changing the model of the whole system.

Our model extends standard automata with a self-modifying feature. Related extensions have been shown to significantly enhance the base model of computation [106]. Here we prove that this is not the case with SAA, by showing a correspondence between SAA and the standard model of execution monitors [114]. This also shows that adaptation in our framework can enforce all *safety properties*.

We illustrate the use of SAA through the use case of a self-adaptive system of autonomous search-and-rescue vehicles, inspired by related work [88, 3, 77, 38]. Using our model, we are able to encode the base system and the adaptation decision process independently, separating them from adaptation patterns. We are thus able to explore radically different adaptation patterns, and prove their correctness through our translation to FDR.

This is the first step towards developing a *compositional* framework for SA CPSs. In this chapter, we propose a modular structure for an adaptation procedure and investigate generic properties adaptation procedures should possess. Through the art gallery example, we discuss how topology and requirements guide a system designer to localize adaptation procedures over a small part of the CPS, e.g., the adaptation procedure satisfying Req. 4: *at most 10 visitors in room D*, can be localized over *room D* only. Such adaptation procedure is too simple to motivate this work, yet having a single adaptation procedure that aims to ensure the satisfaction of all requirements in the art gallery is too complex to motivate this work. Thus, to better understand SA systems and the structure of adaptation procedures, we motivate the framework with an auxiliary example, that of unmanned autonomous vehicles (UAV). Later, in section 5.1.1, we discuss how we can incorporate the concepts from SAA in our compositional encoding of SA CPSs.

3.1 Self-Adaptive Autonomous Vehicles

Here we consider a simple self-adaptive system for search-and-rescue operations by unmanned vehicles, inspired by previous work [88, 3, 77, 38]. The system consists of a number of vehicles moving autonomously in a search area, and a central coordinator responsible for avoiding collisions between vehicles and vehicles escaping the search area. This is achieved by the vehicles reporting their position to the coordinator at specific points in their execution, and the coordinator running a centralised adaptation decision process, changing the behaviour of vehicles through remote commands when a collision is imminent. If two vehicles are closer together than a minimum critical distance, the coordinator adjusts their behaviour so that they move further apart before continuing their normal movement. The following basic requirements must be satisfied by the system:

Req. 1: Vehicles must not collide with each other.

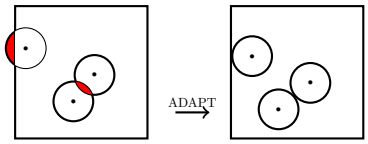


Figure 3.1: Self-Adaptive Autonomous Vehicles System.

Req. 2: All vehicles must remain within the search area.

Req. 3: Every position in the search area must be eventually explored by the vehicles.

The base system here involves the actual vehicle movement, which can be random, or towards specific coordinates when instructed by the adaptation. The adaptation decision process can take as input the current coordinates of the vehicles, their speed, and direction, and issue adaptation commands to the vehicles, nudging them apart, when a possible collision is detected. Both of these components do not depend on the exact points where adaptation is invoked. It would thus be desirable to model our base system in a liberal way, allowing adaptation at virtually every state.[1]

Such a model would allow us to explore different adaptation patterns that ensure system correctness, and evaluate them in terms of efficiency. Here we consider two such patterns.

Time-Triggered Implementation: A possible adaptation pattern could invoke the adaptation procedure at predefined time intervals. This means that time must be explicit in the model which, in an automata-based model, can be done with *tock* self-loops in every state of the base system.

The adaptation decision process, assuming a minimum frequency of adaptation events, can compute new positions for vehicles that are in danger of collision or escaping the search area (see fig. 3.1) It should then be possible to verify that if adaptation events occur with at least that frequency, then Req. 1 and Req. 2 are satisfied. The third requirement, Req. 3, should also be satisfied, if vehicle movement is indeed random and the adaptation procedure does not always send vehicles to the same coordinates.

Event-Triggered Implementation: An alternative implementation approach of the vehicles' coordinator is to invoke an adaptation procedure when vehicles are closer than a minimum distance and a collision is possible. With this adaptation pattern, the position of the vehicles must be monitored and once the minimum distance is reached for a pair of vehicles, an adaptation event occurs.

In large search areas this event-triggered adaptation pattern can be more efficient, in that it can achieve the system requirements with fewer invocations of the adaptation decision process.

3.2 Overview of the Modelling Framework

Two main components of a self-adaptive system are the base system and the adaptation decision process. As we discussed in the introduction, a third, equally important component is the adaptation pattern. In our modelling framework, we use the abstract formalism of automata to express these components.

The base system is essentially a standard automaton, with states and a labelled-transition function. This expresses the regular behaviour of the system. To encode the adaptation decision process, we extend such automata with a special-purpose transition, annotated with a $\star$. During

[1] If the adaptation decision manager makes assumptions about the system states it is invoked (e.g., a minimum distance between vehicles) then this should be taken into account by the allowed adaptation points.

this transition, an adaptation function Π, which is part of the definition of our automata, inputs the current system state and outputs *a new transition function* and a new state for the system. The new transition function replaces the existing transition function of the base system, changing the automaton's behaviour and encoding adaptation. The new state enables the encoding of information sent from the adaptation decision process to the base system. Note that Π is partial, meaning that $\star$-transitions may not be defined for some system states. This allows us to encode *may-adapt* transitions; i.e., the possibility of adaptation at these states.

When a system designer models a self-adaptive system in SAA, the base behaviour of the system is first identified and encoded as a standard automaton. Then, inspecting the requirements of the system, a decision procedure is encoded as a function from states to transition functions and states. The states for which this function is defined get a $\star$-transition (can-adapt).

The last ingredient of the model is the adaptation pattern: when *must* the adaptation function be invoked during the execution of the automaton. In SAA this is encoded as a separate automaton which we call *adaptation automaton*, satisfying specific properties (see Def. 3.3.3). After composition with the base system, the adaptation automaton determines the states where adaptation must happen, essentially pruning the outgoing transitions of can-adapt states of the base system. The encoding of adaptation patterns as adaptation automata allows the system designer to encode multiple—simple and complex—adaptation patterns, and evaluate them in terms of efficiency. Once a suitable adaptation automaton is established, the system can be verified for correctness through a translation to FDR.

3.3 Self-Adaptive Automata

We now present Self-Adaptive Automata (SAA), a formalism for modelling self-adaptive systems, and use them to model the autonomous vehicles example. Key aspects of SAA are the encoding of adaptation by dynamic modification of the transition function, and the inclusion of adaptation events in execution traces through a special-purpose $\star$-action. The latter is important for adaptation actions to synchronise during SAA composition. The formal definition of SAA is the following.

Definition 3.3.1 (Self-Adaptive Automata)**.** A Self-Adaptive Automaton (SAA) $\mathcal{M}$ is a tuple $\langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$ where:

- Q is a set of states;
- Σ is a set of symbols; we let a range over Σ;
- $\Delta \in \mathcal{P}(Q \times \Sigma \rightharpoonup Q)$ is a set of transition functions, partially mapping states and symbols to states;
- $q_0 \in Q$ is the initial state;
- $\delta_0 \in \Delta$ is the initial transition function;
- $\Pi \in Q \rightharpoonup Q \times \Delta$ is the adaptation function, partially mapping states to states and transition functions.

$\Diamond$

In SAA, behaviour modification (i.e., adaptation) is an atomic action. It occurs as a single $\star$-event. This is a useful simplification when considering high-level models of SA systems, although it

may not capture the behaviour of systems where adaptation is propagated non-atomically, perhaps for performance reasons. Moreover, the definition considers deterministic SAA in order to compare with Execution Monitors (EMs) (in section 3.5) which are also deterministic. The translation to CSP, presented in section 3.4, can be adapted to non-deterministic SAAs.

The operational semantics of an SAA is defined as a labelled transition system (LTS) over configurations containing the current state and active transition function of the automaton. We use the symbol $\star$ to label adaptation transitions; we let $\Sigma_\star = \Sigma \cup \{\star\}$ and $a_\star$ range over $\Sigma_\star$.

$$\langle q, \delta\rangle \xrightarrow{a} \langle q', \delta\rangle \qquad \text{if } \delta(q, a) = q'$$
$$\langle q, \delta\rangle \xrightarrow{\star} \langle q', \delta'\rangle \qquad \text{if } \Pi(q) = \langle q', \delta'\rangle$$

This semantics shows that SAA configurations transition according to the function δ in the configuration, which can change by $\star$-transitions, modelling adaptation. These adaptation transitions make SAA an expressive framework for adaptive systems, while still remaining a standard computation model (section 3.5). Note traditional deterministic automata are a special case of SAA, with no $\star$-transitions.

Example 3.3.2. We now turn our attention to the example in section 3.1. We can model this example with an SAA

$$\mathcal{M}_1 = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$$

composed of the following elements:

$$\Sigma = \{goto.p_1.p_2 \mid p_1, p_2 \in Loc\}$$
$$Q = \{\langle p_1, p_2\rangle \mid p_1, p_2 \in Loc\}$$

Here we assume a set of locations *Loc* representing the possible positions of a vehicle, and a *distance* function $d(p_1, p_2)$ which gives the cartesian distance of any two points. We assume that there are positions every one unit on the horizontal and vertical axes. For simplicity we restrict our attention to a system with two vehicles, and consequently the transitions $goto.p_1.p_2$ change the positions of these vehicles over time. We use one state for each pair of positions of the two vehicles.

The initial transition function δ_0 allows the vehicles to move to any position which is at most one distance unit away from the current position; in effect vehicles can move one position to the left, right, up or down, or stay at the same position.

$$\delta_0(\langle p_1, p_2\rangle, goto.m_1.m_2) = \begin{cases} \langle m_1, m_2\rangle & \text{if } d(p_1, m_1) \leq 1 \\ & \text{and } d(p_2, m_2) \leq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Recall from section 3.1 that if the coordinator deems that the vehicles are too close, it will issue a command for the vehicles to move further apart. This is captured here with a transition function where the two vehicles can only move to certain positions, before making any more moves. This transition function depends on the current positions of the vehicles, p_1 and p_2, and the positions that they must move to, l_1 and l_2, respectively. Thus we have a family of transition functions

which depend on these parameters:

$$\delta_{\langle s_1,s_2\rangle\to\langle l_1,l_2\rangle}(\langle p_1,p_2\rangle, goto.m_1.m_2) =$$
$$\begin{cases} \langle m_1,m_2\rangle & \text{if } m_1=l_1, m_2=l_2, s_1=p_1 \text{ and } s_2=p_2 \\ \langle m_1,m_2\rangle & (s_1\neq p_1 \text{ or } s_2\neq p_2) \text{ and } d(p_1,m_1)\leq 1, d(p_2,m_2)\leq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

This function allows only a single move from $\langle s_1,s_2\rangle$; that to $\langle l_1,l_2\rangle$. From any other $\langle p_1,p_2\rangle$ all valid moves are allowed.

The model has an adaptation transition from every state.

$$\Pi(\langle p_1,p_2\rangle) =$$
$$\begin{cases} (\langle p_1,p_2\rangle,\ \delta_{\langle p_1,p_2\rangle\to\langle p_1',p_2'\rangle}) & \text{if } \mathit{danger}(\langle p_1,p_2\rangle) \\ & \text{and } \mathit{safe}(\langle p_1,p_2\rangle)=\langle p_1',p_2'\rangle \\ (\langle p_1,p_2\rangle,\ \delta_0) & \text{otherwise} \end{cases}$$

Adaptation relies on the auxiliary predicate *danger* which identifies the states where vehicles are too close to each other, and the function *safe* which, when given a dangerous position, returns a safe position that the vehicles can move to, as depicted in fig. 3.1.

Although adaptation is possible at every state, the system does not need to adapt after every move, provided that there is enough distance between vehicles, and between vehicles and the border. For example if $\mathit{danger}(\langle p_1,p_2\rangle)$ is true when p_1 and p_2 are at distance less than six units between them and less than three units from the border, and assuming *safe* moves vehicles to positions where the *danger* predicate is false, then the system can safely adapt every *two* transitions, guaranteeing no collisions.

As we will see in the next subsection, different adaptation automata can be defined independently from the base system and adaptation decision process, and be combined with them through automata composition. ◊

3.3.1 Adaptation Automata

Here we model adaptation patterns as a special class of SAA, which we call adaptation automata. We also define composition of SAA, used for enforcing an adaptation patterns on system models.

An adaptation automaton pinpoints which adaptation moves *must* be performed. Consequently, it can be modelled by an SAA whose adaptation transitions are mandatory. This means that states with outgoing adaptation transitions have no other outgoing transition.

Moreover, we assume that a single adaptation move is powerful enough to give the system the desired behaviour. Therefore, adaptation patterns, by definition, have no two consecutive $\star$-moves.

Definition 3.3.3 (Adaptation Automaton)**.** We say an SAA

$$\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$$

is an adaptation automaton when for all $q\in Q$ and $\delta\in\Delta$ the following conditions hold:

1. if $\langle q,\delta\rangle \not\xrightarrow{\star}$ then $\langle q,\delta\rangle \xrightarrow{a}$, for all $a\in\Sigma$ and;
2. if $\langle q,\delta\rangle \xrightarrow{\star} \langle q',\delta'\rangle$ then $\langle q,\delta\rangle \not\xrightarrow{a}$, for all $a\in\Sigma$;

3. if $\langle q, \delta \rangle \xrightarrow{\star} \langle q', \delta' \rangle$ then $\langle q', \delta' \rangle \not\xrightarrow{\star}$

$\Diamond$

We will call a state with an outgoing $\star$-transition, an *adaptation state*, and any other state a *regular state*. The aim of the adaptation automaton is to pinpoint the adaptation points during a system's execution. The adaptation automaton should not influence the execution in any other way or include alternative execution paths which allow a system to bypass an adaptation. These two properties are respectively enforced by Conditions 1 and 2 in the above definition. Condition 1 ensures that if an adaptation is not to occur from the current state then the adaptation automaton must enable all Σ-transitions so as not to influence the system's execution. Condition 2, together with the fact that SAA are deterministic by definition, ensures that when adaptation automaton state that an adaptation is to happen, both the adaptation automaton and the base system have no alternative transition except the adaptation.

One of the benefits of SAAs is that they can be composed by automata intersection. In particular we will use this to compose models of systems with adaptation automata.

Definition 3.3.4 (SAA Composition)**.** Let

$$\mathcal{M}_1 = \langle Q_1, \Sigma, \Delta_1, q_0', \delta_0', \Pi_1 \rangle$$

and

$$\mathcal{M}_2 = \langle Q_2, \Sigma, \Delta_2, q_0'', \delta_0'', \Pi_2 \rangle$$

be SAA. We define $\mathcal{M}_1 \cap \mathcal{M}_2$ to be the SAA:

$$\langle Q_1 \times Q_2, \Sigma, \Delta, (q_0', q_0''), \delta_0' \cap \delta_0'', \Pi \rangle$$

where $Q_1 \times Q_2$ is the cartesian product of the state sets and

$$\Delta = \{\delta_1 \cap \delta_2 | \delta_1 \in \Delta_1 \text{ and } \delta_2 \in \Delta_2\}$$

$$(\delta_1 \cap \delta_2)(\langle q_1, q_2 \rangle, a) = \begin{cases} \langle q_1', q_2' \rangle & \text{if } \delta_1(q_1, a) = q_1',\ \delta_2(q_2, a) = q_2' \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Pi(\langle q_1, q_2 \rangle) = \begin{cases} \langle (q_1', q_2'), \delta_1' \cap \delta_2' \rangle & \text{if } \Pi_1(q_1) = \langle q_1', \delta_1' \rangle,\ \Pi_2(q_2) = \langle q_2', \delta_2' \rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\Diamond$

Theorem 3.3.5. *For any two SAAs $\mathcal{M}_1$ and $\mathcal{M}_2$:*

1. *if M_1 or M_2 is an adaptation automaton then $M_1 \cap M_2$ is an adaptation automaton;*

2. *if M_1 and M_2 are finite then $M_1 \cap M_2$ is finite;*

Here we show the proof for 1. The proof for the other clause follow Lemma A.0.2 to A.0.5.

Proof. An automaton is an adaptation automaton when for all $q, q' \in Q_1 \times Q_2$ and transition functions δ then $\langle q, \delta \rangle \xrightarrow{\star} \langle q', \delta' \rangle$ implies

- there not exists $a \in \Sigma$ such that $\langle q, \delta \rangle \xrightarrow{a} \langle q', \delta \rangle$ Proven by contradiction. Assume that

there exists $q_1, q_2 \in Q_1 \times Q_2$, $a \in \Sigma$ and transition function δ such that

$$\delta((q_1 q_2), a) = (q_1', q_2') \tag{3.1}$$
$$\Pi((q_1, q_2)) = \langle (q_1', q_2'), \delta' \rangle \tag{3.2}$$

From the definition of the intersection, it must be the case that

$$\delta = \delta_1'' \cap \delta_2'' \tag{3.3}$$
$$\delta' = \delta_1' \cap \delta_2' \tag{3.4}$$
$$\delta_1''(q_1, a) = q_1' \text{ and } \delta_2''(q_2, a) = q_2' \tag{3.5}$$
$$\Pi_1(q_1) = \langle q_1', \delta_1' \rangle \text{ and } \Pi_2(q_2) = \langle q_2', \delta_2' \rangle \tag{3.6}$$

Without loss of generality assume that M_1 is a adaptation automaton. A contradiction arise as it cannot be $\delta_1''(q_1, a) = q_1'$ and $\Pi_1(q_1) = \langle q_1', \delta_1' \rangle$.

- $\langle q', \delta' \rangle \not\xrightarrow{\star}$ Proven by contradiction, Assume that $\mathcal{M}_1$ is a adaptation automaton. The transition $\langle q, \delta \rangle \xrightarrow{\star} \langle q', \delta' \rangle$ could happen because there exists $q_1 \in Q_1$ and $q_2 \in Q_2$ such that $q = (q_1, q_2)$

$$\Pi((q_1, q_2)) = \langle (q_1', q_2'), \delta_1' \cap \delta_2' \rangle \tag{3.7}$$
$$\Pi_1(q_1) = \langle q_1', \delta_1' \rangle \text{ and } \Pi_2(q_2) = \langle q_2', \delta_2' \rangle \tag{3.8}$$
$$q' = (q_1', q_2') \text{ and } \delta' = (\delta_1' \cap \delta_2') \tag{3.9}$$

Similarly, for the second transition $\langle q', \delta' \rangle \xrightarrow{\star}$

$$\Pi((q_1', q_2')) = \langle (q_1'', q_2''), \delta_1'' \cap \delta_2'' \rangle \tag{3.10}$$
$$\Pi_1(q_1') = \langle q_1'', \delta_1'' \rangle \text{ and } \Pi_2(q_2') = \langle q_2'', \delta_2'' \rangle \tag{3.11}$$

A contradiction arise as $\langle q_1'', \delta_1' \rangle \xrightarrow{\star}$ contradicts the initial assumption that $\mathcal{M}_1$ is a adaptation automaton.

□

Returning to the example of section 3.1 we note that a self-adaptive system may satisfy a requirement only under certain adaptation patterns. For example, the vehicle system modelled in Ex. 3.3.2 avoids collisions only if adaptation runs at least once every two system transitions.

Example 3.3.6. In section 3.1 we discussed two adaptation patterns for autonomous vehicles. Here we encode them as adaptation automata as shown in fig. 3.2. The time-triggered adaptation automaton requires adaptation every two transitions. Let

$$\mathcal{A}_1 = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$$

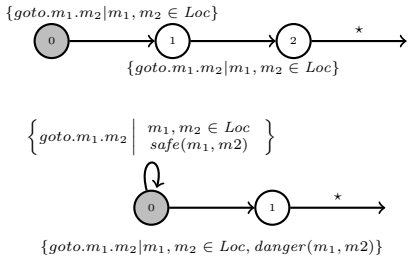


Figure 3.2: The transition function for $\mathcal{A}_1$ (top) and $\mathcal{A}_2$ (bottom)

where

$$Q = \{q_0, q_1, q_2\} \qquad \Sigma = \{goto.p_1.p_2 \mid p_1, p_2 \in Loc\} \qquad \Delta = \{\delta_0\}$$

$$\delta_0(q_i, goto.p_1.p_2) = \begin{cases} q_{i+1} & \text{if } i \in \{0, 1\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Pi(q) = \begin{cases} \langle q_0, \delta_0 \rangle & \text{if } q = q_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Dually, the event-triggered adaptation automaton discussed in section 3.1 requires an adaptation when the vehicles becoming dangerously close to one another. Let

$$\mathcal{A}_2 = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$$

where

$$Q = \{q_0, q_1\} \qquad \Sigma = \{goto.p_1.p_2 \mid p_1, p_2 \in Loc\} \qquad \Delta = \{\delta_0\}$$

$$\delta_0(q, goto.p_1.p_2) = \begin{cases} q_0 & \text{if } q = q_0 \text{ and } safe(p_1, p_2) \\ q_1 & \text{if } q = q_0 \text{ and } danger(p_1, p_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Pi(q_1) = \langle q_0, \delta_0 \rangle$$

$\Diamond$

Note that the conditions of adaptation automata allow us to abstract over the implementation details of both the system and the adaptation function but pinpoint exactly where the adaptation must happen. For instance, in $\mathcal{A}_1$, only the $\star$-transition is enabled in q_2, in contrast with q_0 and q_1 which enable all non-adaptive transitions. The correctness of these solutions is presented in Ex. 3.4.4

3.4 Refinement-based Verification

Here we give an encoding of SAA into CSP and use the FDR verifier to prove such policies (safety requirements) in our examples using FDR's trace model. We also use the encoding to reason about functional requirements using the CSP failure model, as well as the failure-divergence model. As we show in this section, our example systems do indeed satisfy the safety requirements, and the functional requirements under the failure model. However they do not satisfy the functional requirements with respect to the failure-divergence model. This is because our systems have

inherent internal divergences. In the vehicle system, for example, the vehicles may move over the same positions without exploring the entire search space.

3.4.1 Translation to CSP

Any SAA

$$\mathcal{M}{=}\langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$$

with finite Δ can be translated into CSP_M by following the following steps:

1. We define a data-type $\widehat{Q}$ representing the set of states Q. We write $\widehat{q}$ for the translated state q.

2. We define events to represent Σ . We also define low-level adaptation events $\{\star\langle\widehat{q}\rangle \mid q \in Q\}$; where we attach the state to the adaptation event. For instance $\star\langle\widehat{q_0}\rangle$ represents $\Pi(q_0)$.

3. We translate a transition function $\delta \in \Delta$ as a parametric process $\widehat{\delta}$ taking the state and performing the next transition from that state as defined by δ, i.e.,

$$\widehat{\delta}(\widehat{q}) = \Box\{e \to \widehat{\delta}(\widehat{q'}) \;\mid\; e \in \Sigma \text{ and } \delta(q,e) = q'\}$$

   The SAA semantics allows us to perform a $\star$-transition if q is an adaptable state ($\Pi(q)$ is defined). For each adaptable state, we add the $\star$-transition to the choice of events that can be performed,

$$\widehat{\delta}(\widehat{q}) = \star\langle\widehat{q}\rangle \to \mathit{STOP} \quad \text{if } \Pi(q) = \langle\delta, q'\rangle$$

$$\Box\{e \to \widehat{\delta}(\widehat{q'}) \;\mid\; e \in \Sigma \text{ and } \delta(q,e) = q'\}$$

4. Adaptation is the replacement of one transition function by another. In the translation this is encoded as the interruption of a process to initiate another process. In our model, adaptation is defined with respect to the state at the time of the adaptation. We utilise the interrupt construct in CSP to implement the adaptation functionality. Recall we communicate the state with an $\star$-transition. Thus, $\widehat{\Pi}$ needs to evolve to the correct process according to that state. Intuitively,

$$\widehat{\Pi} = \Box \begin{cases} \star\langle\widehat{q_1}\rangle \to \widehat{\delta_1'}(\widehat{q_1'}) & \text{if } \Pi(q_1) = \langle\delta_1', q_1'\rangle \\ \star\langle\widehat{q_2}\rangle \to \widehat{\delta_2'}(\widehat{q_2'}) & \text{if } \Pi(q_2) = \langle\delta_2', q_2'\rangle \\ \quad\vdots \end{cases}$$

   Consider the CSP process $P = (e \to P) \,\Box\, (\star\langle q_1\rangle \to \mathit{STOP})$ nested in the process $P \,\triangle_{\star\langle\_\rangle}\, \widehat{\Pi}$. Here P can evolve through the event e, but as soon as the external choice transition to $\star\langle q_1\rangle$, this event synchronises with one of the events in $\widehat{\Pi}$ and the process $\widehat{\delta_1'}(\widehat{q_1'})$ is initiated instead of P.

   Note that the new processes P_1 may need to adapt as well so we recursively include the interrupt construct in $\widehat{\Pi}$,

$$\widehat{\Pi} = \Box \left\{ \star\langle\widehat{q}\rangle \to (\widehat{\delta}(\widehat{q'}) \,\triangle_{\star\langle\_\rangle}\, \widehat{\Pi}) \;\middle|\; \begin{array}{c} \Pi(q) = \langle\delta, q'\rangle \\ \text{and } q \in Q \end{array} \right\}$$

5. The final step is to strip state information attached to adaptation events by renaming all adaptation events to a single distinguished CSP event $\star$. We initialise the SAA $\mathcal{M}$ to the initial configuration $\delta_0(q_0)$. Recall that this process can be interrupted by an adaptation as explained in 4. We define $\mathcal{M}$ as

$$\widehat{\mathcal{M}} = \left(\widehat{\delta}_0(\widehat{q_0}) \triangle_{\star\langle\_\rangle} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!]$$

In our model an SA system is defined as the composition of the system and an adaptation automaton. An adaptation automaton is an SAA and hence can be translated to a CSP process using the technique explained above. The intersection of the two SAA, $\mathcal{A} \cap \mathcal{M}$, can be implemented using the CSP parallel composition construct.

$$\widehat{\mathcal{A}} \underset{Events}{\parallel} \widehat{\mathcal{M}}$$

The next two lemmas we show that

Lemma 3.4.1. *Let $\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$, then for all $q \in Q$ and $\delta \in \Delta$, $\alpha \in \Sigma_\star$*
$\langle q, \delta\rangle \xrightarrow{\alpha} \langle q', \delta'\rangle$ implies
$$\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \xrightarrow{\alpha} \left(\widehat{\delta'}(\widehat{q'}) \triangle_{\{\star\}} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!]$$

Proof. By case-analysis on α

- $\alpha = a$. From the reduction $\langle q, \delta\rangle \xrightarrow{\alpha} \langle q', \delta'\rangle$, we know

$$\delta(q, a) = q' \tag{3.12}$$
$$\langle q, \delta\rangle \xrightarrow{a} \langle q', \delta\rangle \tag{3.13}$$

From the other side, we know that

$$\widehat{\delta}(\widehat{q}) = \left(\Box_{(e,q')\in\delta(q)} e \to \widehat{\delta}(\widehat{q'})\right) \Box \star(q) \to STOP \tag{3.14}$$

We know though that $a \in \delta(q)$. This allows us to construct the reduction,

$$\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right) \xrightarrow{a} \left(\widehat{\delta'}(\widehat{q'}) \triangle_{\{\star\}} \widehat{\Pi}\right) \tag{3.15}$$
$$\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \xrightarrow{a} \left(\widehat{\delta'}(\widehat{q'}) \triangle_{\{\star\}} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \tag{3.16}$$

- $\alpha = \star$. From the reduction $\langle q, \delta\rangle \xrightarrow{\star} \langle q', \delta'\rangle$, we know

$$\Pi(q) = \langle q', \delta'\rangle \tag{3.17}$$

From the other side, we know that,

$$\widehat{\delta}(\widehat{q}) = \left(\Box_{(e,q')\in\delta(q)} e \to \widehat{\delta}(\widehat{q'})\right) \Box \star(\widehat{q}) \to STOP \tag{3.18}$$

$$\widehat{\Pi} = \Box_{q\in\mathrm{dom}(\Pi)} \star(\widehat{q}) \to \left(P_q \triangle_{\{\star\}} \Pi\right) \tag{3.19}$$

where $P_q = \widehat{\delta}_q(\widehat{q'})$ and $\Pi(q) = \langle q', \delta_q \rangle$. From $\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right)$, we can construct the reduction

$$\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right) \xrightarrow{\star(\widehat{q})} \widehat{\delta}_q(\widehat{q'}) \triangle_{\{\star\}} \Pi \tag{3.20}$$

$$\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \xrightarrow{\star} \left(\widehat{\delta}_q(\widehat{q'}) \triangle_{\{\star\}} \Pi\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \tag{3.21}$$

□

Lemma 3.4.2. *Let $\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$, then for all $q \in Q$ and $\delta \in \Delta$, $\alpha \in \Sigma_\star$, $\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \xrightarrow{\alpha} P$ implies*

- $P = \left(\widehat{\delta'}(\widehat{q'}) \triangle_{\{\star\}} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!]$
- $\langle q, \delta \rangle \xrightarrow{\alpha} \langle q', \delta' \rangle$

Proof. By case-analysis on α

- $\alpha = a$. We know that,

$$\widehat{\Pi} = \square_{q \in \mathrm{dom}(\Pi)} \left\{ \star\langle\widehat{q}\rangle \to (\widehat{\delta}(\widehat{q'}) \triangle_{\star\langle\_\rangle} \widehat{\Pi}) \mid \Pi(q) = \langle \delta, q' \rangle \right\} \tag{3.22}$$

$$\widehat{\delta}(\widehat{q}) = \star\langle\widehat{q}\rangle \to STOP \quad \text{if } \Pi(q) \text{ is defined} \tag{3.23}$$

$$\square\{e \to \widehat{\delta}(\widehat{q'}) \mid e \in \Sigma \text{ and } \delta(q, e) = q'\}$$

This implies that $\widehat{\Pi}$ cannot reduce through an a event. From $\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \xrightarrow{a} P$, we know this is only possible through a reduction $\widehat{\delta}(\widehat{q}) \xrightarrow{a}$, we know this implies

$$P = \left(\widehat{\delta}(\widehat{q'}) \triangle_{\{\star\}} \widehat{\Pi}\right) [\![^\star/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \tag{3.24}$$

$$\delta(q, a) = q' \tag{3.25}$$

We can, thus, construct the reduction

$$\langle q, \delta \rangle \xrightarrow{a} \langle q', \delta \rangle \tag{3.26}$$

as required.

- $\alpha = \star$. We know that,

$$\widehat{\Pi} = \square_{q \in \mathrm{dom}(\Pi)} \left\{ \star\langle\widehat{q}\rangle \to (\widehat{\delta}(\widehat{q'}) \triangle_{\star\langle\_\rangle} \widehat{\Pi}) \mid \Pi(q) = \langle \delta, q' \rangle \right\} \tag{3.27}$$

$$\widehat{\delta}(\widehat{q}) = \star\langle\widehat{q}\rangle \to STOP \quad \text{if } \Pi(q) \text{ is defined} \tag{3.28}$$

$$\square\{e \to \widehat{\delta}(\widehat{q'}) \mid e \in \Sigma \text{ and } \delta(q, e) = q'\}$$

This means that the only reduction possible is

$$\left(\widehat{\delta}(\widehat{q}) \triangle_{\star\langle\_\rangle} \widehat{\Pi}\right) [\![^{\star}/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \xrightarrow{\star} STOP \triangle_{\star\langle\_\rangle} \left(\widehat{\delta}'(\widehat{q}') \triangle_{\star\langle\_\rangle} \widehat{\Pi}\right) [\![^{\star}/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \tag{3.29}$$

$$\left(\widehat{\delta}(\widehat{q}) \triangle_{\star\langle\_\rangle} \widehat{\Pi}\right) [\![^{\star}/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \xrightarrow{\star} \left(\widehat{\delta}'(\widehat{q}') \triangle_{\star\langle\_\rangle} \widehat{\Pi}\right) [\![^{\star}/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!] \quad \text{by structural equivalence} \tag{3.30}$$

where $\Pi(q) = \langle q', \delta' \rangle$. This allows us to construct the reduction,

$$\langle q, \delta \rangle \xrightarrow{\star} \langle q', \delta' \rangle \tag{3.31}$$

□

Theorem 3.4.3. *For an SAA* $\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$. *We show that for all* $q \in Q$, $\delta \in \Delta$, $\alpha \in \Sigma_{\star}$

1. $\langle q, \delta \rangle \xrightarrow{\alpha} \langle q', \delta' \rangle$ *implies* $\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right) \sigma \xrightarrow{\alpha} \left(\widehat{\delta}'(\widehat{q}') \triangle_{\{\star\}} \widehat{\Pi}\right) \sigma$
2. $\left(\widehat{\delta}(\widehat{q}) \triangle_{\{\star\}} \widehat{\Pi}\right) \sigma \xrightarrow{\alpha} P$ *implies* $P = \left(\widehat{\delta}'(\widehat{q}') \triangle_{\{\star\}} \widehat{\Pi}\right) \sigma$ *and* $\langle q, \delta \rangle \xrightarrow{\alpha} \langle q', \delta' \rangle$

where $\sigma = [\![^{\star}/_{\star\langle\widehat{q}\rangle} \mid q \in Q]\!]$,

Proof. Follows from Lemma 3.4.1 and 3.4.2 □

This theorem shows that the translation is a strong bisimulation. Therefore the trace, failure, and failure-divergence models of FDR apply to SAA through the translation.

Example 3.4.4. Recall the system in section 3.1, modelled as an SAA in Ex. 3.3.2. We show the CSP representation of the example with two vehicles.

1. First we define the set of states $\widehat{Q} = \{\langle p_1, p_2 \rangle \mid p_1, p_2 \in Loc\}$
2. We define the set of events as Σ and the low-level adaptation events which incorporate the system state:

$$\widehat{\Sigma} = \{goto.m_1.m_2 \mid m_1, m_2 \in Loc\} \cup \{\star\langle p_1, p_2 \rangle \mid p_1, p_2 \in Loc\}$$

3. We have two types of transition functions: the initial transition function δ allowing all movements and $\delta_{(m,n)}$ which initially allows one transition (coordinator's command) and then any movement

$$\widehat{\delta}_0(\widehat{q}) = \Box \begin{cases} goto.m_1.m_2 \to \widehat{\delta}_0(\widehat{y}) & \delta_0(q, goto.m_1.m_2) = y \\ \star\langle\widehat{q}\rangle \to STOP & \Pi(q) = \text{ is defined} \end{cases}$$

$$\widehat{\delta}_{m\to n}(\widehat{q}) =$$

$$\Box \begin{cases} goto.m_1.m_2 \to \widehat{\delta}_{m\to n}(\widehat{y}) & \delta_{m\to n}(q, goto.m_1.m_2) = y \\ \star\langle\widehat{q}\rangle \to STOP & \Pi(q) \text{ is defined} \end{cases}$$

4. The adaptation function Π is implemented as

$$\widehat{\Pi} = \Box \left\{ \star\langle p_1, p_2 \rangle \to (P \triangle_{\star\langle\_\rangle} \widehat{\Pi}) \;\middle|\; \begin{array}{l} \Pi(\langle p_1, p_2 \rangle) = \langle \delta_{m\to n}, \langle p_1, p_2 \rangle \rangle \\ \text{and } P = \widehat{\delta}_{m\to n}(\langle p_1, p_2 \rangle) \end{array} \right\}$$

5. With the above, we implement $\widehat{\mathcal{M}}$ in CSP as

$$\widehat{\delta}_0(\langle l_1, l_2\rangle) \triangle_{\star\langle\_\rangle} \widehat{\Pi}$$

We translate also the two adaptation automata presented in Ex. 3.3.6. The adaptation automaton $\mathcal{A}_1$ requires adaptation every two steps. We model $\mathcal{A}_1$ in CSP as shown below,

$$\begin{aligned}
\widehat{Q}_{\mathcal{A}_1} &= \{0..2\} \\
\widehat{\Sigma} &= \{goto.m_1.m_2 \mid m_1, m_2 \in Loc\} \cup \{\star\langle x\rangle \mid x \in 0..2\} \\
\widehat{\delta}_{\mathcal{A}_1}(x) &= \text{if } x < 2 \\
&\qquad \text{then } \Box\{goto.m_1.m_2 \to \widehat{\delta}_{\mathcal{A}_1}(x+1) \mid m_1, m_2 \in Loc\} \\
&\qquad \text{else } \star\langle 2\rangle \to STOP \\
\widehat{\Pi} &= \star\langle 2\rangle \to (\widehat{\delta}_{\mathcal{A}_1}(0) \triangle_{\star\langle 2\rangle} \widehat{\Pi})
\end{aligned}$$

Similarly, the trigger-based adaptation automaton $\mathcal{A}_2$ requires adaptation when the vehicles' locations satisfy the predicate *danger*. We model $\mathcal{A}_2$ as

$$\begin{aligned}
\widehat{Q}_{\mathcal{A}_2} &= \{0,1\} \quad \widehat{\Sigma} = \{goto.m_1.m_2 \mid m_1, m_2 \in Loc\} \cup \{\star\langle 0\rangle, \star\langle 1\rangle\} \\
\widehat{\delta}_{\mathcal{A}_2}(0) &= \Box \begin{cases} goto.m_1.m_2 \to \widehat{\delta}_{\mathcal{A}_2}(0) & \text{if } m_1, m_2 \in Loc, safe(m_1, m_2) \\ goto.m_1.m_2 \to \widehat{\delta}_{\mathcal{A}_2}(1) & \text{if } m_1, m_2 \in Loc, danger(m_1, m_2) \end{cases} \\
\widehat{\delta}_{\mathcal{A}_2}(1) &= \star\langle 1\rangle \to STOP \\
\widehat{\Pi} &= \star\langle 1\rangle \to (\widehat{\delta}_{\mathcal{A}_2}(0) \triangle_{\star\langle 1\rangle} \widehat{\Pi})
\end{aligned}$$

Finally, $\widehat{\mathcal{A}_1 \cap \mathcal{M}}$ is

$$\texttt{Impl} = \widehat{\delta}_{\mathcal{A}_1}(0)[\star/\star\langle\widehat{q}\rangle \mid q \in Q_{\mathcal{A}_1}] \underset{\widehat{\Sigma}\cup\{\star\}}{\|} \widehat{\mathcal{M}}[\star/\star\langle\widehat{q}\rangle \mid q \in Q_{\mathcal{M}}]$$

Note that $\widehat{\mathcal{A}_2 \cap \mathcal{M}}$ is defined analogously by replacing the adaptation automaton only,

$$\texttt{Impl2} = \widehat{\delta}_{\mathcal{A}_2}(0)[\star/\star\langle\widehat{q}\rangle \mid q \in Q_{\mathcal{A}_2}] \underset{\widehat{\Sigma}\cup\{\star\}}{\|} \widehat{\mathcal{M}}[\star/\star\langle\widehat{q}\rangle \mid q \in Q_{\mathcal{M}}]$$

Properties Proven using FDR Using FDR, we verify that the system with both adaptation policies satisfy the requirements Req. 1, Req. 2 and Req. 3 from section 3.1 by proving that the implementations `Impl` and `Impl2` refine the following three specification processes. The safety requirement Req. 1 assert that vehicles never collide or stated differently no two vehicles go to the same location. This can be specified as

```
Req1 = goto?x:  Int?y:  {y | y<-Int, d(x,y) > 0} -> Req1
```

Similarly, we verify that all vehicles remain within the search space (Req. 2) by showing the implementations refines a specification process `Req2` which recursively accepts any $goto.m_1.m_2$ for

m_1 and m_2 within the search space. We assume `Loc` to be the set of all locations in the search space.

```
Req2 = goto?_ :  Loc ?_ :  Loc -> Req2
```

We use the trace-semantic model to verify safety properties by checking that any derivable trace in the implementation is also derivable in the Specification. In particular, through FDR we can verify the following assertion

```
assert Req1 [T= Impl assert Req2 [T= Impl
```

Note that we can verify the second implementation by either doing similar assertions or alternatively show that the implementations are trace equivalent up-to the adaptation automaton. In the assertions below, we verify that if we hide the $\star$ transitions, the implementation are trace equivalent. The process `P |\ A` is equivalent to `P \ diff(Events,A)` where `P` performs only events not in `A`,

```
assert Impl2 |\{*} [T= Impl |\{*}
assert Impl |\{*} [T= Impl2 |\{*}
```

For the last requirement Req. 3, we want to verify that all positions in the search area can eventually be visited. Since this is a liveness property, trace inclusion is not sufficient. We verify Req. 3 by running the implementation in parallel with a test and check that all possible paths can be extended to pass the test. For each location $l \in$ `Locs`, we define a test that broadcasts a `success` event, signalling that the test passed, when the location l has been visited by a vehicle:

```
T(l) = goto?x?y -> if d(x,p) == 0 or d(y,p) == 0
                   then success -> RUN({|goto|})
                    else T(l)
```

We run all the tests in parallel with the implementation and hide all the events except the `success` events,

```
Tests =
(([|{|goto|}|] l:Locs @ T(l)) [|{|goto|}|] Impl) |\{success}
```

The requirement is satisfied if all the tests pass or stated alternatively the number of success events is the same as the number of locations.

```
Count(n) = n > 0 & success -> Count(n-1)
           n == 0 & ok -> STOP
```

In the assertion, we check that the `ok` event is never refused. The verification ensures that from any path we can eventually broadcast the `ok` event,

```
assert ok->STOP [F= (Count(n) [{success}] Tests) |\{ok}
```

Note that because the vehicles randomly choose the next goto position, the implementation may diverge and running the assertion using the failure-divergence semantic model would rightfully fail. For instance, if the vehicles transition indefinitely between the same pair of locations, such trace can be extended to pass the tests but the test never terminates.

$\Diamond$

3.5 Expressiveness of Self-Adaptive Automata

Our last result is a study on the expressiveness of SAA by translating SAAs to standard automata. We show that SAA, although dynamically change the transition function, do not alter the model of computation of traditional automata. We do this by translating SAAs to Execution Monitors (EM) [114]. Execution Monitors are automata accepting a prefix-closed set of traces (the monitored traces) which is co-recursively enumerable. This result shows that SAA are unlike other dynamic automata, such as self-modifying finite automata, which are more powerful than standard finite automata [106]. Moreover, since EM-enforceable properties correspond to *safety properties* [114], it follows that adaptation in our framework can enforce all such properties.

Theorem 3.5.1. *SAA precisely enforce the safety properties.*

Proof. Details of the proof has been moved to appendix A. □

3.6 Summary

We presented SAA, an automata-based approach for abstract modelling of self-adaptive systems. In our model we decompose a self-adaptive system into three main components: the base system, the adaptation decision process and the adaptation pattern. The adaptation decision process identifies the execution points where the system *may*-adapt, whereas the adaptation pattern selects the execution points where the system *must*-adapt. The adaptation pattern allows a system designer to explore multiple adaptation patterns without changing the base system or the adaptation decision process.

Applying SAA in a compositional setting is not straightforward. Firstly, we have a single distinguished adaptation event to model adaptation. We cannot distinguish between adaptations from different SAAs. A solution for this is to introduce scoping and having adaptation scoped within an SAA. Moreover, the composition formalized in Def. 3.3.4 synchronizes over all events. However, two SAAs may need to synchronize on a subset of the events—their interface. A parallel composition synchronizing on a subset of the events and event scoping are customary in process languages. We therefore move away from SAAs as our modelling framework and opt for process-based approach that naturally supports compositional modelling. The SAA model helped us understand the main questions a system designer needs to answer to model an SA system and how we can modularise the model to enable the experimentation with different implementations of adaptation procedures. Thus, in later chapters, we incorporate the key points of the SAA framework in our compositional encoding for SA CPSs.

Chapter 4

A Methodology for Modelling and Verifying Self-Adaptive CPSs

4.1 The Methodology

In this chapter we introduce a modelling and verification methodology for SA CPSs, which aims to provide assurances for such systems, especially when employed in critical domains. Providing assurances is particularly challenging due to the highly dynamic nature of SA CPSs, which increases the complexity of these systems. Our main tool for tackling this complexity is by employing *compositionality* in both the modelling and verification approach. Compositionality allows a system designer to potentially localize the verification task to a small number of components, while ignoring irrelevant parts of the system. This reduces the state space exploration of the verification task, significantly reducing the effect of the state explosion problem. As we show in this and following chapters, to attain a compositional model and verification, our methodology takes advantage of topological relationships naturally present in CPSs, and a requirement-driven approach to encode self-adaptation. A key ingredient of our methodology is a novel process language, Adaptive CSP, presented in the second part of the chapter.

Our methodology is organised in the following six main steps.

Step 1 (Modelling the CPS)**.** A system designer first identifies the main cyber and physical components of a CPS (e.g., rooms and assets). Each component may have containment and connectivity topological relationships with other components. We represent the components of the art gallery example and their relationships in fig. 4.1. For example, *corridor 2* can contain agents (e.g., visitors, employees and guards) and it is physically connected to the *Stairs* and the *restoration area*. The *access point* have connectivity relationships with the *HVAC* and with other employees' and visitors' devices located in *Floor 2* that are connected to the wireless network. The *Computer Room* has a containment relationship with the *access point* and the agents that are located in it. Similarly, the *restoration area* has containment relationship with the *HVAC* and the agents in it.

Topological relationships can enable execution of system actions. For example, if an agent is contained in a physical area, she can access other connected physical areas and perform actions in them. In the art gallery example, if an agent moves from the *Stairs* to *corridor 2*, she can connect a device she is carrying to the *access point*. Moreover, agents contained in a physical areas can access and/or control co-located assets, if available. For example, agents in the *Computer Room* can switch on/off the *access point*.

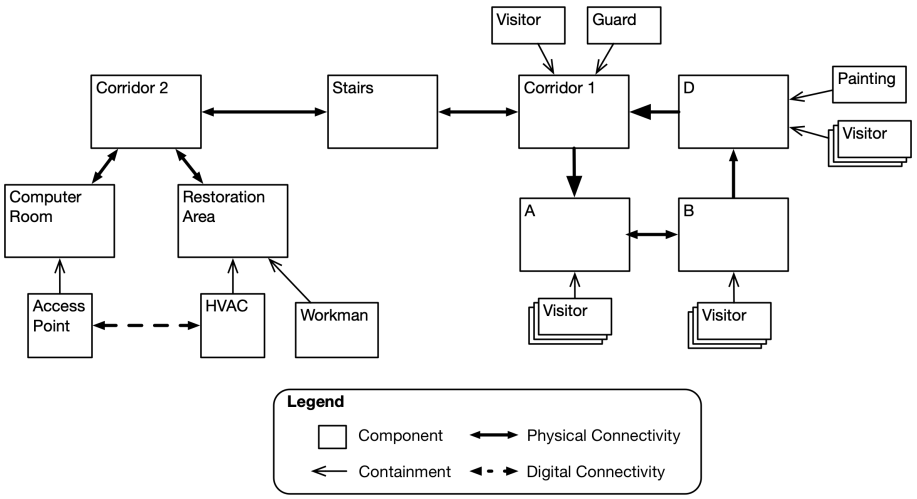


Figure 4.1: Components of the Art Gallery (partial model).

◊

Step 2 (Exploring adaptation procedures)**.** For each requirement, we define an adaptation procedure that aims to ensure its satisfaction by adapting a subset of the components. In this step, the system designer examines each requirement, determining the components that can affect its satisfaction. A component may pose a direct threat to the satisfaction and thus an adaptation procedure needs to adapt system components behaviour to minimize the threat. Alternatively, adaptations may depend on the state of the component and an adaptation procedure needs to monitor its behaviour, without changing, to plan adaptations. The set of adapted and monitored components make up the *scope of an adaptation procedure*. Here we consider the task of identifying the scope of an adaptation procedure to be a manual task, but it can be guided by the topology of the CPS. Certainly, relevant components include the one the requirement refers to. They may also include additional components identified considering its containment and connectivity relationships. For example, the components affecting satisfaction of Req. 1 can include the *Restoration Area* and those related to it (*corridor 2*, *Stairs* and *corridor 1*). Different adaptation procedures can be explored starting from the most specific component relevant to the requirement, and including more components affecting the requirement satisfaction as necessary. For example, to satisfy Req. 1: *Visitors should not interfere with the restoration process*, as we discussed in Ad. Proc. 1.1, an adaptation procedure may consider the *restoration area* and *corridor 2*. More precisely, it may keep track of the presence of a guard in *corridor 2*, and only allow access/exit to/from the *restoration area* if a guard is in *corridor 2*. However, this may not always be desirable because when a guard is not in *corridor 2*, workers may not be allowed to freely access or leave the *restoration area*. Alternatively, the adaptation procedure could monitor visitor access to the *Stairs* from *corridor 1* and notify the guard stationed in *corridor 1* to escort visitors in *corridor 2*, as discussed in Ad. Proc. 1.2. ◊

Step 3 (Encoding with Adaptive CSP)**.** In this step, the system designer manually encodes the part of the CPS relevant to a requirement in our ACSP language. This includes the adaptation procedure and the components in its scope. Each component can be encoded as a parallel process or an internal state. In our example, we encode all components except agents as processes. Components that are adapted must be modelled as processes. We tag such components by unique identifiers called *locations*. Components running in named locations can be adapted through higher-order communication over the location. On a higher-order communication, the

process running in the named location is replaced by the process communicated. In our encoding, each component representing a physical location has an internal state representing the number of visitors, guards and workers that are contained in it. System actions, such as agent movements or *access point* connections, are implemented as first-order communication events (i.e. transmission of data). These first-order communication events encode the connectivity relation that connects two neighbouring components in fig. 4.1. We refer to the family of events that connects a component c to the rest of the system as the interface of c – $\mathcal{I}(c)$. The last element that we encode is the adaptation procedure. We encode adaptation procedures as an ACSP process comprises two parallel processes: an *adaptation pattern* that listens to the first-order events of components in its scope and trigger adaptation when needed and an *adaptation function* that plan and implement adaptations by issuing higher-order communication events to specific components based on the system state. This encoding follows the principles described in Chapter 3.

◊

Step 4 (Verifying system requirements)**.** The system designer verifies each requirement. This is done by translating *parts* of the self-adaptive CPS, as encoded in our language, to existing verification tools—here we choose FDR. A compositionality theorem guarantees that successful verification of a requirement over such a set of components implies the satisfaction of the requirement for the entire system. The smallest set of components that need to be translated is easy to find. For each requirement, only the following elements are translated to FDR:

1. The adaptation procedure that aims to ensure the satisfaction of the requirement

2. The components in the scope of the adaptation procedure that affect the satisfaction of the given requirement. This include components adapted by the adaptation procedure and components whose behaviour is monitored by the adaptation procedure.

3. The first-order communication events of the components included in the translation.

In this dissertation, this ACSP process is referred to as the *cluster* in Def. 5.1.1. In this manner we translate the smallest set of components that include those relevant to the requirement, and whose higher-order adaptation events can be entirely *internalised* (no adaptation event can cross the boundaries of this set) and encoded as internal transitions in FDR. The requirement itself is *specified* using the capabilities of the verification tool; in FDR we use refinement. Specifications encoding describe the correct behaviour of components in the scope of the adaptation procedure according to the requirement. This is described in more detail in Chapter 6. If the verification fails then the requirement and/or adaptation procedure must be re-examined, potentially going back to *Step 2* to consider a different granularity for the adaptation procedure. ◊

Step 5 (Composing adaptation procedure)**.** The system designer, needs to ensure the satisfaction of requirements is preserved when adaptation procedures are composed together. When adaptation procedures have disjoint scopes then the verification in Step 4 is sufficient to ensure requirements satisfaction in the entire system. However, when adaptation procedure scopes overlap, there is the potential of *interference* between them. In this case, we need to re-verify that each requirement is satisfied when all adaptation procedures, and all system components, are composed back together, if there is a potential of interference. We recommend an iterative approach for verifying the satisfaction of requirements so violations are identifying early. If a violation is because a proposed adaptation procedure does not ensure the satisfaction of a requirement, we want to identify the violations before introducing other adaptation procedures, to alleviate debugging. Similarly, if

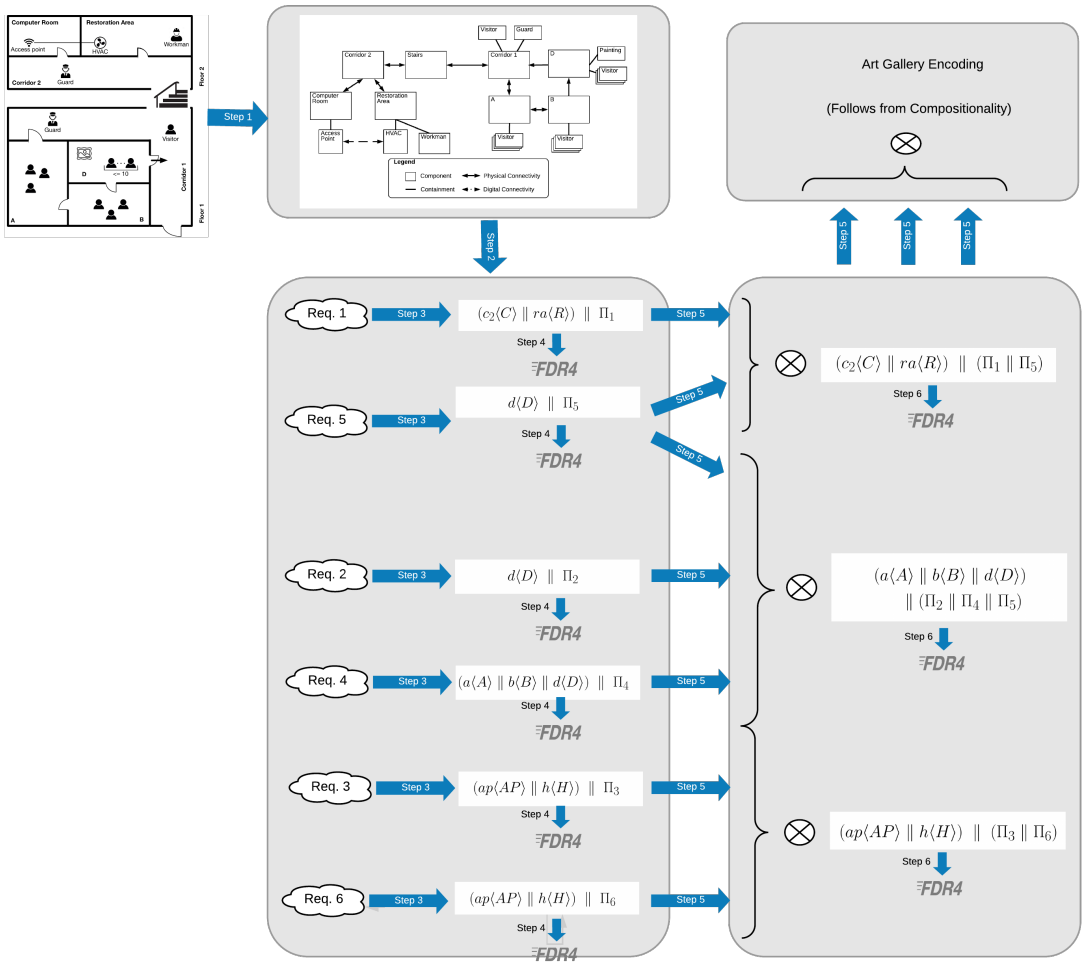


Figure 4.2: The application of the Methodology to the art gallery example. At the top a component model is constructed which is then used to explore the granularity for adaptation procedures. For each requirement, the adaptation procedures and the relevant components are encoded using ACSP. In step 4, we verify that requirements are satisfied locally. After this, we iteratively compose the parts of the encoding and re-verify the satisfaction of requirements by the composition if necessary until all parts of the encoding are composed together.

a violation arises from the composition of two adaptation procedures, iterative verification of compositions localises the source of the violation to the smallest set of adaptation procedures.

◊

Step 6 (Verifying the composition of adaptation procedures)**.** Finally, the system designer verifies the satisfaction of requirements by the composition of adaptation procedures. This is done by translating *parts* of the self-adaptive CPS, as encoded in our language, to FDR – similar to Step 4. If the verification fails then the last specification and adaptation procedure composed must be re-examined, possibly going back to *Step 2* and considering a different granularity for the adaptation procedure. ◊

In fig. 4.2, we present at a high-level the application of the methodology to the art gallery example. We first construct the component model (*step 1*). We utilize the apparent topological layout of the CPS and topological relations to derive this model. For each requirement, we explore levels of granularity and select a subset of the components over which we define an adaptation

Symbol	Meaning
h	higher-order communication event
e	first-order event
α	first-order and higher-order event
l, m, n	locations
P, Q, M, N	ACSP processes
S, T	CSP Processes
Γ	set of locations
$\Gamma \vdash P$	well-formed check for an ACSP process P with respect to an environment Γ
$P \rhd S$	translation of an ACSP process P to CSP process S
$S_{\tilde{R}A}^{M}$	cluster that aims to satisfy requirements $\tilde{R}$ where A and M are the set of locations adapted and monitored respectively (parameters are omitted if they are insignificant to the example)
$S_1 \otimes S_2$	merging of clusters S_1 and S_2
$\Pi_{\tilde{R}}$	adaptation procedure that aims to satisfy requirements $\tilde{R}$
C_c	the internal state of a component c
P_R	the adaptation pattern in the adaptation procedure Π_R
F_R	the adaptation function in the adaptation procedure Π_R
$Spec_R$	the specification for requirement R

Table 4.1: Notations and symbols used throughout the rest of the dissertation

procedure that aims to ensure the satisfaction of a given requirement (*step 2*). We encode the adaptation procedure and the selected components in ACSP (*step 3*). Next, we verify the satisfaction of each requirement in isolation using FDR (*step 4*). Through our theory of compositionality, it suffices to translate to CSP, the input language of FDR, only the selected components and their first-order events and the adaptation procedure defined to satisfy the requirement's satisfaction. From the compositionality theory, we show that the verification results hold for the whole art gallery. In *step 5*, we verify the satisfaction of requirements is preserved when adaptation procedures are composed together. We develop on our compositionality theorem to infer when this verification can be skipped. In particular, overlapping adaptation procedures do not change any of the common components, interference is ruled out and we show that satisfaction follows from our theory of compositionality. In other cases, we need to verify in FDR that requirements satisfaction is preserved when adaptation procedures are composed. The details of this figure become clear in later chapters. In table 4.1 we summarise the notation and symbols used in the remaining chapters of the dissertation. Similarly, the details of the table become clear in later chapters.

We now present the process language—Adaptive CSP, that is the foundation of our modelling framework.

4.2 The Process Language ACSP

Our technique is based on a novel process language which we call Adaptive CSP (ACSP). This language extends Communicating Sequential Processes (CSP) [70] with locations and higher-order communication. ACSP is defined over a set of *first-order events*, ranged over by e, and a set of *location names*, ranged over by l. We reserve the special first-order event τ for internal communication,

Event Transitions:

$$\textsc{EvCh}\ \frac{j \in \mathcal{I}}{\Box_{i\in\mathcal{I}}\, e_i \to P_i \xrightarrow{e_j} P_j} \qquad \textsc{EvParL}\ \frac{M \xrightarrow{e} M' \quad e \notin E}{M \underset{E}{\parallel} N \xrightarrow{e} M' \underset{E}{\parallel} N} \qquad \textsc{EvSync}\ \frac{M \xrightarrow{e} M' \quad N \xrightarrow{e} N' \quad e \in E \quad e \neq \tau}{M \underset{E}{\parallel} N \xrightarrow{e} M' \underset{E}{\parallel} N'} \qquad \textsc{Rec}\ \frac{\sigma = [\vec{e}, (\mathrm{rec}X(\vec{y} := \vec{e}).P)/\vec{y}, X]}{\mathrm{rec}X(\vec{y} := \vec{e}).P \xrightarrow{\tau} P\sigma}$$

$$\textsc{EvHide}\ \frac{P \xrightarrow{e} P'}{(\nu e)P \xrightarrow{\tau} (\nu e)P'} \qquad \textsc{EvEsc}\ \frac{P \xrightarrow{e} P' \quad c \neq e}{(\nu c)P \xrightarrow{e} (\nu c)P'} \qquad \textsc{IfTrue}\ \frac{e_1 \leq e_2}{\mathrm{if}\, e_1 \leq e_2 \,\mathrm{then}\, P \,\mathrm{else}\, Q \xrightarrow{\tau} P} \qquad \textsc{EvLoc}\ \frac{P \xrightarrow{e} P'}{l\langle P\rangle \xrightarrow{e} l\langle P'\rangle}$$

Adaptation Transitions:

$$\textsc{AdSnd}\ \frac{}{l!P.Q \xrightarrow{l!P} Q} \qquad \textsc{AdRcv}\ \frac{}{l\langle Q\rangle \xrightarrow{l?P} l\langle P\rangle} \qquad \textsc{AdSyncL}\ \frac{M \xrightarrow{l!R} M' \quad N \xrightarrow{l?R} N'}{M \underset{E}{\parallel} N \xrightarrow{\tau} M' \underset{E}{\parallel} N'} \qquad \textsc{AdParL}\ \frac{M \xrightarrow{h} M'}{M \underset{E}{\parallel} N \xrightarrow{h} M' \underset{E}{\parallel} N} \qquad \textsc{AdEsc}\ \frac{P \xrightarrow{h} P' \quad c \notin h}{(\nu c)P \xrightarrow{h} (\nu c)P'} \qquad \textsc{AdLoc}\ \frac{P \xrightarrow{h} P'}{l\langle P\rangle \xrightarrow{h} l\langle P'\rangle}$$

Figure 4.3: Transition Semantics of ACSP (omitting symmetric rules).

and use c, d to range over events and location names.

$$c, d ::= e \mid l$$

We also use X for process variables and y for event variables; we let $\hat{e}$ range over events and event variables. We use standard notation for sequences ($\vec{\cdot}$) and the usual syntactic sugar from process languages.

The abstract syntax of the language is:

$$P, Q, R, M, N, \Pi ::= \mathit{SKIP} \mid \Box_{i\in\mathcal{I}}\, \hat{e}_i \to P_i \mid \mathrm{if}\, \hat{e}_1 \leq \hat{e}_2 \,\mathrm{then}\, P \,\mathrm{else}\, Q$$

$$\mid P \underset{\Sigma}{\parallel} Q \mid (\nu c)P \mid X(\vec{\hat{e}}) \mid \mathrm{rec}X(\vec{y} := \vec{\hat{e}}).P \mid l!P.Q \mid l\langle P\rangle$$

Processes, ranged over by P, Q, R, M, N, Π include the standard CSP processes: the inactive process (SKIP); *external choice* where the environment can choose between a set of events $\{e_i \mid i \in \mathcal{I}\}$, written as $\Box_{i\in\mathcal{I}}\, e_i \to P_i$, with $\mathcal{I}$ being an indexing set; a conditional process ($\mathrm{if}\, e \leq e' \,\mathrm{then}\, P \,\mathrm{else}\, Q$); parallel composition of processes ($P \underset{\Sigma}{\parallel} Q$) which synchronise on a set of non-τ events Σ; scope restriction of events and location names ($(\nu c)P$); and recursion ($\mathrm{rec}X(\vec{y} := \vec{e}).P$) through process variables (X) and event parameter variables ($\vec{y}$). ACSP processes also include the new constructs of a higher-order prefix $l!P.Q$ that sends a process P to location l and evolves to process Q, and named locations ($l\langle P\rangle$). Intuitively, a named location $l\langle P\rangle$ marks process P with name l, which can be adapted to $l\langle Q\rangle$ through a higher-order communication with a prefix $l!Q.R$, residing outside of the location.

The operational semantics of ACSP is defined by labelled transitions annotated by: τ, when the transition is internal; $e \neq \tau$, when it is a first-order synchronisation event; $l?P$, when location l is being adapted and becomes P; $l!P$, when an external process initiates such an adaptation. Higher-order transition annotations are ranged over by h; all transition annotations are ranged over by α.

$$h ::= l!P \mid l?P \qquad\qquad \alpha ::= e \mid h$$

The rules of the transition semantics of ACSP are shown in fig. 4.3. Adaptation transitions are the main novelty of ACSP; event transitions are similar to CSP. External choice (EvCh) can transition to one of its residual processes P_j, annotating the transition with the corresponding action e_j. The parallel rule (EvParL) and its (omitted) symmetric rule propagate an event transition e of P over a parallel composition $P \underset{E}{\parallel} Q$, provided that e is not mentioned in the set of events E on which P and Q must synchronise. Rule EvSync synchronises such an event.

Rule Rec unfolds a recursion by an internal transition during which the formal parameters of the recursive process $\vec{y}$ are replaced by the actual parameters $\vec{e}$, and the recursion variable X is replaced with the recursive process itself. Note that substitution of X in processes of the form $X(\vec{e'})$ preserves $\vec{e'}$ as the formal parameters. That is, $X(\vec{e'})[^{\mathrm{rec}X(\vec{y}:=\vec{e}).P}/_X]$ becomes $\mathrm{rec}X(\vec{y} = \vec{e'}).P$, and we can have the following example transitions:

$$\begin{aligned}
&\mathrm{rec}X(y := 1).(e.y \to X(y+1)) \xrightarrow{\tau} \\
&e.1 \to \mathrm{rec}X(y := 1+1).(e.y \to X(y+1)) \xrightarrow{e.1} \xrightarrow{\tau} \\
&e.2 \to \mathrm{rec}X(y := 2+1).(e.y \to X(y+1)) \xrightarrow{e.2} \xrightarrow{\tau} \dots
\end{aligned}$$

Rule EvHide hides an event from the surrounding processes, converting it into τ; EvEsc propagates an event over a scope restriction, and IfTrue and (omitted) IfFalse evaluate a conditional. Our language also includes rule EvLoc which propagates event transitions over locations.

We have chosen CSP-style synchronisation for first-order events in our language because they simplify the encoding of *monitor processes*, which can be used to keep track of state and encode adaptation procedures.

Example 4.2.1. Consider *corridor 2* (c_2) from fig. 1.1, which is connected with a door to the *stairs* (s). We can encode the movement of visitors, employees and guards from one space to the other by a family of first-order events: $vis_{(s,c_2)}$ encodes the movement of a visitor from the *stairs* to *corridor 2*, and $vis_{(c_2,s)}$ the reverse; similarly, events $grd_{(s,c_2)}$ and $grd_{(c_2,s)}$ encode the movement of guards between the two spaces. A process that models visitor movement is the following:

$$C2_0 = \Box \begin{cases} vis_{(s,c_2)} \to C2_0 \\ vis_{(c_2,s)} \to C2_0 \end{cases}$$

Here we assume that a guard is already in, and not allowed to leave c_2. In order to keep track of the number of visitors in *floor 2*, entering from the stairs, we can use a monitor process for the events $vis_{(s,c_2)}$ and $vis_{(c_2,s)}$: [1]

$$C(v) = \Box \begin{cases} vis_{(s,c_2)} \to C(v+1) \\ v > 0 \,\&\, vis_{(c_2,s)} \to C(v-1) \\ cnt.v \to C(v) \end{cases}$$

Note that here we write $C(v) = P$ instead of $C = \mathrm{rec}X(y := v).P$, and assume a standard encoding of natural numbers. Process $C(v)$ keeps the number of visitors in v, which it can report through the (parametrised event) $cntv$. We can now compose the two processes

$$C2_0 \underset{vis}{\parallel} C(v)$$

[1] we use $b\&P$ as a shorthand for if b then P else $STOP$

such that whenever a visitor enters or leaves *corridor 2* from the *stairs*, the state of $C(v)$ increments or decrements, accordingly. ◊ ◊

Adaptation is a higher-order transition which has a single sender and a single receiver. The sender is a process $l!P.Q$ which performs a transition annotated with $l!P$, according to rule AdSnd of fig. 4.3. The receiver is a location with name l which performs transition $l?P$, according to rule AdRcv. These transitions synchronise with rule AdSyncL (and its omitted symmetric rule), and are propagated over parallel, scope restriction, and locations according to rules AdParL (and its omitted symmetric), AdEsc, and AdLoc, respectively. Because of the sender-receiver communication pattern of adaptation, and since we intend locations to have unique names, we choose binary communication for these higher-order transitions.

Example 4.2.2. Continuing from Ex. 4.2.1, an adaptation procedure can query the counter after every visitor move, and change the behaviour of *corridor 2*, when for example there are no more visitors in *floor 2*. To achieve this we consider that process $C2_0$, encoding visitor movement in and out of c_2, is inside a location l_{C2}, and can thus be adapted. The following process Π installs process $C2_1$ in this location when all visitors have left c_2:

$$\Pi = vis_{(\_,\_)} \to cnt.v \to \text{if } v = 0 \text{ then } l_{C2}!C2_1.\Pi \text{ else } \Pi$$

Here $vis_{(\_,\_)}$ represents any visitor events. Process $C2_1$ allows the guard to leave c_2 and prevents further movement into c_2, encoding the closing of the space.

$$C2_1 = grd_{(c_2,s)} \to SKIP$$

The model of *corridor 2* is the composition of the above processes.

$$(\nu\ l_{C2})(\ l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(1) \underset{vis,cnt}{\parallel} \Pi\ )$$

The following execution shows how location l_{C2} is adapted when the last visitor leaves:

$$\begin{aligned}
&(\nu\ l_{C2})(\ l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(1) \underset{vis,cnt}{\parallel} \Pi\ ) \xrightarrow{vis_{(c_2,s)}} \\
&(\nu\ l_{C2})(\ l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(0) \underset{vis,cnt}{\parallel} \Pi_1\ ) \xrightarrow{cnt.0} \\
&(\nu\ l_{C2})(\ l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(0) \underset{vis,cnt}{\parallel} \Pi_2(0)\ ) \xrightarrow{\tau} \\
&(\nu\ l_{C2})(\ l_{C2}\langle C2_0\rangle \underset{vis}{\parallel} C(0) \underset{vis,cnt}{\parallel} l_{C2}!C2_1.\Pi\ ) \xrightarrow{\tau} \\
&(\nu\ l_{C2})(\ l_{C2}\langle C2_1\rangle \underset{vis}{\parallel} C(0) \underset{vis,cnt}{\parallel} \Pi\ )
\end{aligned}$$

Here we let $\Pi_1 = cnt.v \to \Pi_2(v)$ and $\Pi_2(v) = (\text{if } v = 0 \text{ then } l_{C2}!C2_1.\Pi \text{ else } \Pi)$. The last transition is due to the synchronisation of the transitions

$$l_{C2}\langle C2_0\rangle \xrightarrow{l?C2_1} l_{C2}\langle C2_1\rangle \qquad\qquad l_{C2}!C2_1.\Pi \xrightarrow{l!C2_1} \Pi$$

Note that in this example we restricted the scope of the l_{C2} to illustrate that the adaptation of a location can be *localised*—no process outside the restriction can adapt l_{C2}. ◊

We now outline a series of definitions that we utilize in later chapters. In Def. 4.2.3, we define the functions $out(P)$ and $in(P)$ that return the free locations in a process P used in send-prefixes

and locations, respectively.

Definition 4.2.3. $out(P)$ and $in(P)$ are defined by the rules

$$\begin{aligned} out((\nu c).P) &= out(P) - \{c\} \\ out(l!P.Q) &= \{l\} \cup out(P) \cup out(Q) \\ in(l\langle P\rangle) &= \{l\} \\ in((\nu c).P) &= in(P) - \{c\} \end{aligned}$$

All other constructs of the language are derived by the union of the recursive calls results to these functions. ◊

We define three denotational semantic models for ACSP, inspired from CSP [105]. Prior, we outline important definitions needed for defining the semantic models.

Definition 4.2.4. A trace t is a sequence of first-order and higher-order events, denoted as $\langle \alpha_1, \alpha_2, \ldots \rangle$. We say an ACSP process P has a trace t, written as $P \xrightarrow{t}$ iff $t = \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$ and $P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} P_n$ ◊

We define closed processes. An ACSP process P is closed when all the traces contain only first-order events. Any adaptation is localized within P.

Definition 4.2.5 (Closed Process)**.** A process P is closed iff $traces(P) \subseteq \Sigma^*$ ◊

Definition 4.2.6. The set of traces, failures and divergence of a process P are:

$$\begin{aligned} traces(P) &= \{t \mid P \xrightarrow{t} \} \\ refusal(P) &= \{X \subseteq \Sigma \mid P \not\xrightarrow{a} Q \text{ and } a \in X\} \\ failures(P) &= \{(t, X) \mid P \xrightarrow{t} Q \text{ and } X \in refusal(Q)\} \\ Q_0 \Uparrow &= \text{ for all } n \in N. \exists Q_{n+1}. \; ; Q_n \xrightarrow{\tau} Q_{n+1} \\ div(P) &= \{t \mid P \xrightarrow{t} Q \text{ and } Q \Uparrow\} \end{aligned}$$

◊

Using these definitions, we define three denotational semantic models—*trace*, *failure* and *failure-divergence* semantic models.

Definition 4.2.7 (Trace Semantic Model)**.** Any trace in process $Impl$ can be matched with a trace in *Spec*. This define what a process *can* do. Trace refinement is synonyms with the verification of safety properties, where we check that nothing bad ever happens,

$$Spec \sqsubseteq_{\mathtt{T}(ACSP)} Impl \quad \text{iff} \quad traces(Impl) \subseteq traces(Spec)$$

◊

Definition 4.2.8 (Failure Semantic Model)**.** This refinement relation allows us to describe safety and liveness properties. Akin to the trace semantic model, we check that all traces in process $Impl$ are contained in process *Spec* and we assert that the process $Impl$ must eventually accept the trace in *Spec* or diverges. A violation is the absence of a suffix to a desired state.

$$\begin{aligned} Spec \sqsubseteq_{\mathtt{F}(ACSP)} Impl \quad \text{iff} \quad & failures(Impl) \subseteq failures(Spec) \\ & \text{and } traces(Impl) \subseteq traces(Spec) \end{aligned}$$

◊

Definition 4.2.9 (Failure-Divergence Semantic Model)**.** This semantic model allows us to distinguish between deadlocks and livelocks. It adds to the other semantic models the notion of divergence, where a process does not perform any visible events. Additional to the other semantic models, the failure-divergence model checks that all divergence in $Impl$ are found in $Spec$, i.e., he process $Impl$ does not diverge from any state where the process $Spec$ does not

$$Spec \sqsubseteq_{\mathtt{FD}(ACSP)} Impl \quad \text{iff} \quad failures(Impl) \subseteq failures(Spec) \text{ and } div(Impl) \subseteq div(Spec)$$

◊

We now define a notion of equivalence for ACSP processes. We identify equivalence between two process if an external observer cannot distinguish between the two processes. This is the notion of strong bisimulation.

Definition 4.2.10 (Bisimulation relation)**.** The relation $\mathcal{R} \subseteq P_1 \times P_2$ is said to be a (strong) bisimulation iff $P_1 \mathcal{R} P_2$ implies the following

1. If $P_1 \xrightarrow{a} P_1'$ implies there is a P_2' such that $P_2 \xrightarrow{a} P_2'$ and $P_1' \mathcal{R} P_2'$
2. If $P_2 \xrightarrow{a} P_2'$ implies there is a P_1' such that $P_1 \xrightarrow{a} P_1'$ and $P_1' \mathcal{R} P_2'$

◊

Two processes are said to be bisimilar $P \sim Q$, iff there is a bisimulation relation $\mathcal{R}$ that relates them $(P, Q) \in \mathcal{R}$. The relation $\sim$ is the largest bisimulation.

These definitions are referenced in later chapters to draw down properties about our ACSP language from properties in CSP.

4.3 Next Chapters

In the first part of the chapter, we outlined a methodology to compositionally model and verify the satisfaction of security requirements in SA CPSs. We exploit the topological layout of CPSs to manually deduce all the components and their connectivity. For each requirement, we systematically explore different levels of granularities for encoding an adaptation procedure that aims to ensure its satisfaction possibly with the aid of a domain expert. An adaptation procedure is the composition of two sub-processes: one deciding when to adapt and the other what the adaptation should be. Lastly, we compose the adaptation procedures to ensure the satisfaction of requirements is preserved in the overall system.

We then presented ACSP a process language inspired from CSP, that we utilize to model SA CPSs. For the process language, we overviewed the main syntax, transition rules and semantic models.

In the next chapters, we expand upon each step of our methodology starting with Step 2. In particular, in Chapter 5, we investigate how the topology guides us to localize the satisfaction of requirements to small parts of the CPS and how to encode clusters in ACSP language. In Chapter 6, we present our verification approach that leverages existing verification techniques and supports compositional verification. Finally, in Chapter 7, we overview a technique to systematically verify the satisfaction of requirements by adaptation procedures composition. There we highlight cases where re-verification results follows from the compositionality theory.

Chapter 5

Steps 2 & 3: Exploring Adaptation Procedures and ACSP Encoding

After the system designer maps the architecture of a CPS, she identifies a subset of these components that affect the satisfaction of each given requirement. An adaptation procedure over this family of components (the adaptation procedure scope) is defined to preserve a given requirement. Although manual, the task of identifying components relevant to a requirement can be guided by topological relationships [102] (e.g., containment and connectivity between system components). For example, in a smart building a valuable physical asset may be contained in a room while a digital asset, such as data, could be contained in a server. Moreover, physical areas can be connected through doors or digital devices can be connected through an IP network. If a requirement aims to preserve integrity of digital information stored in a server, the components that can affect its satisfaction are, for example, the rooms that are connected to the one containing the server as well as the devices digitally connected to the server.

We model components as parallel processes in ACSP and system actions (e.g., access/exit to/from a room, connection to the wireless network) as first-order communication (i.e., transmission of data). The process language ACSP allows representing adaptation procedures which use higher-order communication (i.e., transmission of processes) to implement adaptation.

5.1 The Cluster

In the verification of a requirement R, the system designer identifies a subset of the components that affect the satisfaction of R and encodes an adaptation procedure Π_R that monitors and adapts some of the selected components with the aim to ensure the satisfaction of R. The ACSP process composing the adaptation procedure and the components in its scope in parallel is referred to as an *unary cluster*.

Definition 5.1.1 (Cluster)**.** An ACSP process $S_{\vec{R}}$ is a *cluster* satisfying requirements $\vec{R}$ if it is of the form

$$S_{\vec{R}} = (\nu\, l_{m\ldots n}) \left(\left(\Big\| _{i\in\{m\ldots n\}} [E_i]\ l_i\langle P_i\rangle \right) \|_{E_{m\ldots n}} \left(\Big\| _{r\in\vec{R}} \Pi_r \right) \right)$$

where for every requirement $r \in \vec{R}$, the adaptation procedure Π_r is designed to ensure the satisfaction of r, and $out(\Pi_r) \subseteq l_{m\ldots n}$. A cluster comprises also a subset of the *named* components in our

system, selected by the system designer, which affect the satisfaction of the requirements $\vec{R}$. These components are encoded in the process $\left(\Vert_{i \in m \ldots n} [E_i]\ l_i\langle P_i \rangle \right)$ which is the replicated parallel construct. The replicated parallel construct compose the processes $l_i\langle P_i \rangle$ in parallel where $i \in m \ldots n$ and each process $l_i\langle P_i \rangle$ synchronizes with all the other processes in the replicated parallel construct over the events $E_i \cap \bigcup_{x \in \{m \ldots n\} \setminus i} E_x$. ◊

Definition 5.1.2 (Unary cluster)**.** An ACSP process is a unary cluster if and only if it is a cluster associated with a single requirement. ◊

To lighten notation, we drop the $\vec{R}$ subscript from a cluster when it is clear from the context.

In later sections, we discuss how the topology guides a system designer to identify components that affect the satisfaction of a requirement. A component may affect the satisfaction of a requirement in two ways. Firstly, a component may pose a threat to the satisfaction of a requirement and thus an adaptation procedure needs to adapt its behaviour to minimize the threat. Alternatively, adaptation may depend on the state of the component and an adaptation procedure needs to monitor its behaviour, without changing, to plan adaptations. We distinguish between the adapted and monitored components from the cluster encoding.

Definition 5.1.3 (Adapted Component)**.** A component at location l is *adapted* in the cluster $S_{\vec{R}}$ iff $l \in out(\Pi_{\vec{R}})$. We range over $A_{\vec{R}}$ for the adapted components for a cluster $S_{\vec{R}}$. ◊

Definition 5.1.4 (Monitored Component)**.** A component at location l is *monitored* in the cluster $S_{\vec{R}}$ iff $l \notin out(\Pi_{\vec{R}})$ but $l \in in(\Pi_{\vec{R}})$. We range over $M_{\vec{R}}$ for the monitored components for a cluster $S_{\vec{R}}$. ◊

When the distinction between monitored and adapted components is important, we include this information in the cluster definition. We write $S_{\vec{R}_A^M}$ to be the cluster $S_{\vec{R}}$ in which the set of locations A and M are adapted and monitored in the cluster respectively; otherwise we drop the parameters A and M.

5.1.1 Adaptation Procedures Encoding

In Chapter 3. we present an abstract model for SA systems, which we called SAA. The novel aspect of the framework is the distinction of execution points where the system *may*-adapt from execution points where the system *must*-adapt. We achieve this by introducing a third component to the traditional modules of an SA system: the *adaptation pattern*, that selects the execution points where an SA system *must*-adapt. The pattern synchronizes the execution points with the traditional elements of an SA systems—an *adaptation function* which determines the adaptation and when the system *may*-adapt, and a *base-system*. The modularization of the adaptation procedure provides the system designer a technique to feasibly experiment with different adaptation procedures to pick the most optimal and correct encoding.

This encoding also sits closer to the implementation of adaptation procedures for SA CPSs. In a CPS, like the art gallery, components are in different physical locations, e.g., rooms. The adaptation procedures, monitoring a set of components, need to deploy and embed monitoring functionality close to or on the components being monitored. These monitors communicate intermittently the state of the components to a (potentially centralized) adaptation function that communicates back the adaptation. By having the implementation reflect more faithfully the model, it is easier for the system designer to make changes to the model post-deployment and reduce the gap between the model and the implementation. Consider for instance in the art gallery example, tail-gating to

room D has been noticed by the security guard and thus the system needs to be adjusted to allow for a small degree of tail-gating. The system designer identifies that the adaptation procedure ensuring the satisfaction of Req. 2 needs to be updated. She decides to add more check-points to detect tail-gating more accurately and thus updates the adaptation pattern. She verifies the new behaviour and then maps the change to the appropriate monitor inside *room D*. Alternatively, she may change the adaptation function to allow fewer visitors in *room D* and thus updates the adaptation function. Once the changes are verified, she maps the change to the appropriate process.

Applying the SAA framework in a compositional setting is not straightforward. Consider the art gallery example, where we define an adaptation procedure for Req. 2, which constraints the number of visitors in *room D* to 10. The *room D* component and the adaptation procedure make up a single SAA, but this SAA is only one part in the art gallery encoding and we need to compose it with the other SAAs having their own $\star$-transitions. The composition needs to be able to distinguish between different $\star$-transitions and also synchronize with other SAAs on a subset of the events—the interface between the components.

These issues are handled naturally by process languages. We thus propose an encoding of adaptation procedures in ACSP inspired from the SAA framework, where an adaptation procedure comprises an *adaptation pattern*—a process that monitors a subset of the components and determines when the system *must*-adapt, and an adaptation function that determines when the system *may*-adapt and the outcome of the adaptation. We now discuss the encoding of the adaptation pattern and adaptation function.

The adaptation pattern: We let P_R mean the encoding in ACSP for the adaptation pattern in the adaptation procedure Π_R defined to ensure the satisfaction of a requirement R. In Def. 3.3.3, we outline generic properties adaptation patterns should possess. An adaptation pattern should track the first-order events of the components in the scope. At every execution point, either only the adaptation $\star$-event or all the first-order events from the selected components are accepted. This guarantees that the adaptation pattern only interfere with the behaviour of the system's components through adaptation. In our encoding, an adaptation pattern tracks and communicates the state of the components to an adaptation function, when triggering adaptation. We define a family of distinguished events to model the $\star$-transition,[1]

$$\{|\star|\} = \{\star\langle s\rangle \mid s \in State\}$$

Because an adaptation pattern comprises solely first-order events, we can, using FDR, check that the patten only comprises the first-order events of the selected components and adaptation events and if adaptation events are hidden, the pattern does not refuse any first-order events from the components.

The adaptation function: We let a process F_R represent the adaptation function of the adaptation procedure that designed to ensure the satisfaction of requirement R. The general structure of adaptation functions accepts an external choice of $\{|\star|\}$ events. Depending on the state attached with the $\star$-event, the adaptation function communicates higher-order outputs to the appropriate locations.

$$F_R = \Box_{s \in State} \Big\{ \star\langle s\rangle \rightarrow l!Q(s).F_R$$

[1]Here we make use of enumerate sets: $\{|goto|\}$ means the set of all *goto* events

In the SAA framework, the adaptation function was a stateless, atomic injective map. In the encoding above, adaptation is non-atomic. One interleaving between the adaptation pattern and the adaptation function is to synchronize on an $\star\langle s\rangle$ event but then the adaptation pattern proceeds to track first-order events delaying the higher-order output. For example, consider the adaptation procedure Π_2, guaranteeing at most 10 visitors in room D, the process consists of an (parametrised) adaptation pattern P_2 that triggers adaptation when the number of visitors is 10 (represented by the number in the parenthesis) and an adaptation function F_2 encoding a simple condition statement. Below, we illustrate one possible trace of execution for an adaptation,

$$
\begin{aligned}
& (\nu\, \{\!|\star|\!\}) \left(P_2(10) \underset{\{\!|\star|\!\}}{\parallel} F_2 \right) \\
\xrightarrow{\star\langle 10\rangle} \quad & (\nu\, \{\!|\star|\!\}) \left(P_2(10) \underset{\{\!|\star|\!\}}{\parallel} \text{if } v \geq 10 \text{ then } l_d!R''_D.F_2 \text{ else } l_d!R'_D.F_2 \right) \\
\xrightarrow{vis_{(ED,EB)}} \quad & (\nu\, \{\!|\star|\!\}) \left(P_2(9) \underset{\{\!|\star|\!\}}{\parallel} \text{if } v \geq 10 \text{ then } l_d!R''_D.F_2 \text{ else } l_d!R'_D.F_2 \right)
\end{aligned}
$$

Here, the adaptation pattern triggers adaptation by broadcasting $\star\langle 10\rangle$ to F_2. However, the process above continues monitoring first-order events from *room D*, bypassing the adaptation entirely. We can encode an atomic adaptation function by adding an acknowledge event *ack* that F_R broadcasts back to P_R signalling the completion of adaptation.

$$
\begin{aligned}
P_R &= \star\langle s'\rangle \to ack \to P_R \qquad \text{for some } s' \in State \\
F_R &= \underset{s\in State}{\Box} \left\{ \star\langle s\rangle \to l!Q(s).ack \to F_R \right.
\end{aligned}
$$

The composition: We range over Π for adaptation procedures. We subscript the requirement that the adaptation procedure is designed to enforce e.g., Π_R ensures the satisfaction of R. An adaptation procedure composes in parallel the processes P_R and F_R that synchronizes on the *ack* event and $\{\!|\star|\!\}$ events. These events are also scoped (and hidden) in the adaptation procedure.

$$
\Pi_R = (\nu\ \{\!|\star|\!\}, ack) \left(P_R \underset{\{\!|\star|\!\}, ack}{\parallel} F_R \right)
$$

Proposition 5.1.5. *In our encodings of adaptation procedures, we know the following*

1. *An adaptation pattern P_R is a well-formed (closed) ACSP process such that $loc(P_R) = \emptyset$.*
2. *For an adaptation procedure Π, $in(\Pi) = \emptyset$*

5.2 Modelling the Art Gallery Example

Here we discuss how the requirements from the art gallery example can be encoded in ACSP. We show the encoding for Requirements 1 and 3, the encoding for the remaining requirements can be found in appendix C.

Following Steps 2 and 3 described in section 4.1, we define adaptation procedures, over a subset of the cyber and physical components of the system, that aims to ensure the continued satisfaction of requirements through self-adaptation. Identifying the cyber and physical components of the

systems is guided by the topology and their topological relationships. A partial component model for the art gallery is shown in fig. 4.1.

Requirement 1: ***Visitors should not interfere with the restoration process*** To satisfy this requirement we require that when visitors are in the *restoration area*, the guard stays present.

Here we encode a simple access policy, described in Ad. Proc. 1.1: *the door between corridor 2 and restoration area precludes entrance of visitors to the latter, unless a guard is in the corridor and the door from corridor 2 to the stairs precludes the guard from leaving floor 2 if there are visitors in the restoration area.*

The components relevant to this policy are the *restoration area* and *corridor 2*. These components are connected in the model of fig. 4.1 through the physical connectivity of a door. We design a family of first-order events to encode this connectivity: $E_{ra} = \{t_{(c_2,\mathrm{ra})}, t_{(\mathrm{ra},c_2)}\}$ respectively encode the movement of agents from *corridor 2* (c_2) to the *restoration area* (ra) and *vice-versa*. Here t can take the form vis or grd to represent visitor or guard movement, respectively.

The *restoration area* (including its door) can have one of two functionalities:

1. Visitors are allowed to enter/leave the *restoration area*, encoded by the process:

$$R_0 = \Box_{t\in\{grd,vis\}} \begin{cases} t_{(\mathrm{ra},c_2)} \to R_0 \\ t_{(c_2,\mathrm{ra})} \to R_0 \end{cases}$$

   We assume that this process models an unlocked door with free movement of agents.

2. Visitors are not allowed to enter the restoration area, encoded by the process:

$$R_1 = grd_{(\mathrm{ra},c_2)} \to R_1 \,\Box\, grd_{(c_2,\mathrm{ra})} \to R_1 \,\Box\, vis_{(\mathrm{ra},c_2)} \to R_1$$

   In process R_1, the event $vis_{(c_2,\mathrm{ra})}$ representing visitor moving from *corridor 2* (c_2) to the *restoration area* (ra) is precluded.

The internal state of the *restoration area* keeps track of visitors and the guard movements to make sure that the number of people leaving is smaller than or equal to the number of people entering the area. We define two processes $C_{(ra,v)}$ which monitor the visitors' movements and $C_{(ra,g)}$ monitoring the guard movements.

$$\begin{aligned} C_{ra} =& C_{(ra,v)}(0) \,|||\, C_{(ra,g)}(0) \\ C_{(ra,g)}(n) =& \Box \begin{cases} n > 0 \,\& & grd_{(\mathrm{ra},c_2)} \to C_{(ra,g)}(n-1) \\ & grd_{(c_2,\mathrm{ra})} \to C_{(ra,g)}(n+1) \end{cases} \\ C_{(ra,v)}(n) =& \Box \begin{cases} n > 0 \,\& & vis_{(\mathrm{ra},c_2)} \to C_{(ra,v)}(n-1) \\ & vis_{(c_2,\mathrm{ra})} \to C_{(ra,v)}(n+1) \end{cases} \end{aligned}$$

In order for adaptation to take effect, the process responsible for access to the *restoration area* should be in a location with name ra. The encoding of the *restoration area*, initially with no guard or visitor present in the *floor 2*,

$$ResArea = \left(C_{ra} \underset{E_{ra}}{\|} ra\langle R_1 \rangle \right)$$

The *corridor 2* component allows movement to/from the *restoration area* and the *stairs*. We encode the movement between *corridor 2* (c_2) and the *stairs* (s) by the family of events $t_{(c_2,s)}$ and $t_{(s,c_2)}$, where $t \in \{grd, vis\}$. We assume the set of event $E_{c_2} = \{t_{(c_2,\text{ra})}, t_{(\text{ra},c_2)}, t_{(c_2,s)}, t_{(s,c_2)}\}$ to represent the access/exit movement to and from *corridor 2*, where t can take the form vis or grd to represent visitor or guard movement, respectively.

To satisfy Req. 1, the guard should be allowed to leave the upper floor (through the stairs) *only* when there are no visitors in the *restoration area*. Thus *corridor 2* can have one of two functionalities:

1. The guard is allowed to leave *corridor 2* as there are no visitors in the *restoration area*, encoded as the process $C2_0$ where visitors and the guard are allowed to move between *corridor 2* and adjacent rooms,

$$C2_0 = \Box_{t\in\{vis,grd\}} \begin{cases} t_{(s,c_2)} \to C2_0 \\ t_{(c_2,s)} \to C2_0 \\ t_{(c_2,\text{ra})} \to C2_0 \\ t_{(\text{ra},c_2)} \to C2_0 \end{cases}$$

2. The guard is not allowed to leave, encoded by the process $C2_1$ where the event $grd_{(c_2,s)}$ is precluded,

$$C2_1 = \Box \begin{cases} vis_{(s,c_2)} \to C2_1 \\ vis_{(c_2,s)} \to C2_1 \\ vis_{(c_2,\text{ra})} \to C2_1 \\ vis_{(\text{ra},c_2)} \to C2_1 \\ grd_{(s,c_2)} \to C2_1 \end{cases}$$

The internal state of *corridor 2* keeps track of the number of visitors and guard in the space, which can be expressed as a monitor of the movement events.

$$\begin{aligned} C_{c_2} =& C_{(c_2,v)}(0) \,|||\, C_{(c_2,g)}(0) \\ C_{(c_2,g)}(n) =& \Box_{rm\in\text{ra},s} \begin{cases} & grd_{(rm,c_2)} \to C_{(c_2,g)}(n+1) \\ n>0\,\& & grd_{(c_2,rm)} \to C_{(c_2,g)}(n-1) \end{cases} \\ C_{(c_2,v)}(n) =& \Box_{rm\in\text{ra},s} \begin{cases} & vis_{(rm,c_2)} \to C_{(c_2,v)}(n+1) \\ n>0\,\& & vis_{(c_2,rm)} \to C_{(c_2,v)}(n-1) \end{cases} \end{aligned}$$

The encoding of *corridor 2* uses location $c2$ to adapt the functionality of the door connecting it to the *stairs*.

$$Corr2 = \left(C_{c_2} \underset{E_{c_2}}{\|} c2\langle C2_0\rangle \right)$$

We now define the adaptation procedure Π_1 that monitors and adapts the behaviour of the *restoration area* and *corridor 2* components to guarantee the satisfaction of Req. 1,. The adaptation procedure Π_1 ensures the following two criteria:

1. The Guard cannot leave upstairs by moving from *corridor 2* to the *stairs* if visitors are in the *restoration area*

2. Visitors cannot enter the *restoration area* through the door connecting the area to *corridor 2* if the guard is not in *floor 2*, i.e., the *restoration area* or *corridor 2*

As outlined in section 5.1.1, an adaptation procedure comprises an adaptation pattern that tracks the state of the components and identifies the execution points where a system *must*-adapt and an *adaptation function* that determines the adaptation outcome and when a system *may*-adapt. We define an adaptation pattern P_1 that triggers adaptation when a visitor enters an empty *restoration area*, when the last visitor exits the *restoration area* and when a guard moves to or from *floor 2*.

$$
P_1(v,g) = \Box \begin{cases}
vis_{(c_2,\mathrm{ra})} \rightarrow & \text{if } v = 0 \text{ then } \star\langle v+1, g\rangle \rightarrow ack \rightarrow P_1(v+1,g) \\
& \text{else } P_1(v+1,g) \\
vis_{(\mathrm{ra},c_2)} \rightarrow & \text{if } v = 1 \text{ then } \star\langle v-1, g\rangle \rightarrow ack \rightarrow P_1(v-1,g) \\
& \text{else if } v > 0 \text{ then } P_1(v-1,g) \\
& \text{else } P_1(v-1,g) \\
grd_{(c_2,s)} \rightarrow & \text{if } g > 0 \text{ then } \star\langle v, g-1\rangle \rightarrow ack \rightarrow P_1(v,g-1) \\
& \text{else } P_1(v,g-1) \\
grd_{(s,c_2)} \rightarrow & \star\langle v, g+1\rangle \rightarrow ack \rightarrow P_1(v,g+1) \\
e \rightarrow & P_1(v,g) \text{ where } e \in E_{ra}, E_{c_2} \backslash \{vis_{(c_2,\mathrm{ra})}, vis_{(\mathrm{ra},c_2)}, grd_{(c_2,s)}, grd_{(s,c_2)}\}
\end{cases}
$$

At every execution point, either all first-order events from the *restoration area* and *corridor 2* are accepted or the $\{\!|\star|\!\}$-event are accepted.

We now define a process F_1 that encodes the adaptation function. The process, through higher-order outputs, precludes the guard from leaving *floor 2* if visitors are in the *restoration area* and the visitors from entering the *restoration area* without the presence of the guard.

$$
\begin{aligned}
F_1 = & \text{let } F_1'' = \star\langle v, g\rangle \rightarrow \text{if } g > 0 \text{ then } ra!R_0.ack \rightarrow F_1 \text{ else } ra!R_1.ack \rightarrow F_1 \\
& \text{within } \star\langle v, g\rangle \rightarrow \text{if } v > 0 \text{ then } c2!C2_1.ack \rightarrow F_1'' \text{ else } c2!C2_0.ack \rightarrow F_1''
\end{aligned}
$$

The adaptation procedure is encoded as

$$
\Pi_1 = (\nu \{\!|\star|\!\}, ack) \left(P_1(0,0) \underset{\{\!|\star|\!\}, ack}{\parallel} F_1 \right)
$$

The *unary cluster* that we verify to ensure the satisfaction of Req. 1 comprises $Corr2$ and $ResArea$ and the adaptation procedure Π_1,

$$
S_1 = (\nu ra, c2) \left(\left(Corr2 \underset{E_{(c_2,ra)}}{\parallel} ResArea \right) \underset{E_{c_2}, E_{ra}}{\parallel} \Pi_1 \right)
$$

where the event set $E_{(c_2,ra)} = E_{c_2} \cap E_{ra}$ represents the connectivity between *corridor 2* and *restoration area*.

As we explain in the following chapter, we can verify the correctness of S_1 above *independently* of the rest of the system, through our translation to FDR. We later show that the model above

trace refines the specification $Spec_1$ below. We specify the requirement as an abstract process describing the correct behaviour described informally in the requirement. We define the processes *NoGrd* and *GrdPres* to describe the accepted behaviour when the guard is not upstairs and when the guard eventually moves upstairs respectively,

$$Spec_1 = \text{let}$$

$$NoGrd = \Box \begin{cases} grd_{(s,c_2)} \rightarrow GrdPres(0) \\ vis_{(s,c_2)} \rightarrow NoGrd \\ vis_{(c_2,s)} \rightarrow NoGrd \end{cases} \qquad GrdPres(n) = \Box \begin{cases} & vis_{(c_2,\text{ra})} \rightarrow GrdPres(n+1) \\ n > 0 \,\& & vis_{(\text{ra},c_2)} \rightarrow GrdPres(n-1) \\ n = 0 \,\& & grd_{(c_2,s)} \rightarrow NoGrd \\ & grd_{(\text{ra},c_2)} \rightarrow GrdPres(n) \\ & grd_{(c_2,\text{ra})} \rightarrow GrdPres(n) \\ & grd_{(s,c_2)} \rightarrow GrdPres(n) \end{cases}$$

$$\text{within } NoGrd$$

We can also encode an alternative policy, described in Ad. Proc. 1.2 in which the door to the restoration area is rarely locked, improving the movement of restoration workers. To do this, we extend the adaptation procedure of the *restoration area*, so that it monitors the entry of visitors to the *stairs* and calls the guard from *corridor 1* when necessary.

$$\begin{aligned} \Pi_r(b) = {} & grd_{(s,c_2)} \rightarrow RA!R_0.\,\Pi_r(\texttt{tt}) \\ & \Box\ grd_{(c_2,s)} \rightarrow RA!R_1.\,\Pi_r(\texttt{ff}) \\ & \Box\ vis_{(c,s)} \rightarrow \text{if } \neg b \text{ then } call\_guard \rightarrow \Pi_r(b) \text{ else } \Pi_r(b) \\ & \Box\ vis_{(\text{ra},c_2)} \rightarrow \Pi_r(b) \Box\ vis_{(c_2,\text{ra})} \rightarrow \Pi_r(b) \end{aligned}$$

Note that the rest of the encoding of the system needs no change.

Requirement 3: ***the HVAC should not be controlled remotely by unauthorised users***
An user can get unauthorized access to the *HVAC* through the *access point*. This is represented through a connectivity relation in the component model in fig. 4.1. The *HVAC* does not need to be connected to the internet all the time. We therefore satisfy the requirement by disconnecting the *HVAC* while visitors are connected to the internet through the same *access point*, as explained in Ad. Proc. 3.1. Employees may also connect to internet to conduct sensitive work (through the *access point*). As a security measure, visitors are disconnected if there are any employees connected to the network. Thus, in the presence of an employee connection all visitors are disconnected from the *access point* and because no visitor is connected, it is safe to reconnect the *HVAC*.

The components that affect the satisfaction of this requirement are the *HVAC* and the *access point*. The HVAC component perform connect and disconnect events that synchronize with the access point, encoding the connectivity relation between the two components. We define a family of first-order events $E_H = \{conn_{hvac}, disconn_{hvac}\}$ where $conn_{hvac}$ represent a connect commands from the HVAC component and $disconn_{hvac}$ represents a disconnect event. The behaviour of the component is a cycle of *connect* and *disconnect* events

$$H = conn_{hvac} \rightarrow disconn_{hvac} \rightarrow H$$

The *access point* listens for *connect* and *disconnect* events from the *HVAC*, employees and visitors. We define a family of first-order events to encode this communication: $conn_t$ and $disconn_t$

where $t \in \{vis, emp, hvac\}$. We also encode the visitors disconnect command as *disconnect* which states that all visitors are disconnected. We assume E_{ap} to represent the set of events for the *access point*.

The HVAC component listens for connect and disconnect commands from the *access point*. We thus focus the adaptation procedure on the *access point*. The process for the *access point* component can be in one of the following functionalities

1. The *HVAC* is connected and the *access point* is listening for connections from either employees or visitors. This is the initial state of the *access point*, where only the *HVAC* is connected

$$A0 = conn_{hvac} \rightarrow \Box_{t \in \{vis, emp\}} \begin{cases} conn_t \rightarrow A0 \\ disconn_t \rightarrow A0 \end{cases}$$

2. The HVAC is disconnected because a visitor has connected to the *access point*. Other visitors and employees can connect to the *access point* thereafter

$$A1 = disconn_{hvac} \rightarrow \Box_{t \in \{vis, emp\}} \begin{cases} conn_t \rightarrow A1 \\ disconn_t \rightarrow A1 \end{cases}$$

3. All visitors are disconnected through the *disconnect* event because an employee connected to the access point. Since no visitor is connected, the *HVAC* is re-connected.

$$A2 = disconnect \rightarrow conn_{hvac} \rightarrow \Box \begin{cases} conn_{emp} \rightarrow A2 \\ disconn_{emp} \rightarrow A2 \end{cases}$$

4. After the last employee disconnects from the *access point*, visitors are allowed to re-connect

$$A3 = \Box_{t \in \{vis, emp\}} \begin{cases} conn_t \rightarrow A3 \\ disconn_t \rightarrow A3 \end{cases}$$

We define a process C_{ap} to track the internal state of the *access point* to make sure that the number of connections always exceeds or is equal to the number of disconnections. The process C_{ap} is the parallel interleaving of three sub-processes, tracking the number of visitors, employees and *HVAC* connections respectively.

$$C_{ap} = C_{(ap,v)}(0) \;|||\; C_{(ap,e)}(0) \;|||\; C_{(ap,h)}(False)$$

$$C_{(ap,v)}(n) = \Box \begin{cases} & conn_{vis} \rightarrow C_{(ap,v)}(n+1) \\ n > 0 \,\& & disconn_{vis} \rightarrow C_{(ap,v)}(n-1) \\ & disconnect \rightarrow C_{(ap,v)}(0) \end{cases}$$

$$C_{(ap,e)}(n) = \Box \begin{cases} & conn_{emp} \rightarrow C_{(ap,e)}(n+1) \\ n > 0 \,\& & disconn_{emp} \rightarrow C_{(ap,e)}(n-1) \end{cases}$$

$$C_{(ap,h)}(b) = \Box \begin{cases} \neg b \,\& \, conn_{hvac} \rightarrow C_{(ap,h)}(T) \\ b \,\& \, disconn_{hvac} \rightarrow C_{(ap,h)}(F) \end{cases}$$

We encapsulate the initial process $A0$ of the *access point* component in the location ap—$ap\langle A0\rangle$ to make it adaptable. We define an adaptation procedure Π_3 that aims to ensure the satisfaction of Req. 3. The adaptation procedure monitors the $HVAC$ component and adapts the *access point* component.

We define an adaptation pattern that monitors their behaviour as the parallel interleaving of P_3' and P_3''

The process P_3' tracks the number of visitors connected to the *access point*. Adaptation is triggered when the first visitor connects or the last visitor disconnects from the *access point*. The number of visitors connected to the *access point* is communicated to an adaptation function F_3,

$$P_3'(n) = \square \begin{cases} conn_{vis} \rightarrow \text{if } n = 0 \text{ then } \star_v\langle n+1\rangle \rightarrow ack \rightarrow P_3'(n+1) \text{ else } P_3'(n+1) \\ disconn_{vis} \rightarrow \text{if } n = 1 \text{ then } \star_v\langle n-1\rangle \rightarrow ack \rightarrow P_3'(n-1) \text{ else if } n > 0 \text{ then } P_3'(n-1) \text{ else } P_3'(0) \\ disconnect \rightarrow P_3'(0) \\ update \rightarrow P_3'(n) \\ conn_t \rightarrow P_3'(n) \quad \text{where } t \in \{HVAC, emp\} \end{cases}$$

We also define the process P_3'' that tracks the number of employees connected to the *access point*. Adaptation is triggered when the first employee connects or the last employee disconnects from the *access point*. The process P_3'' broadcasts the number of employees connected to the *access point* to the same adaptation function F_3 over a $\star_e$-event. Note, we use a different $\star$-event from P_3' to differentiate the type of connections

$$P_3''(n) = \square \begin{cases} conn_{emp} \rightarrow \text{if } n = 0 \text{ then } \star_e\langle n+1\rangle \rightarrow ack \rightarrow P_3''(n+1) \text{ else } P_3''(n+1) \\ disconn_{emp} \rightarrow \text{if } n = 1 \text{ then } \star_e\langle n-1\rangle \rightarrow ack \rightarrow P_3''(n-1) \text{ else if } n > 0 \text{ then } P_3''(n-1) \text{ else } P_3''(0) \\ disconnect \rightarrow P_3''(n) \\ conn_t \rightarrow P_3''(n) \quad \text{where } t \in \{HVAC, Open\} \\ disconn_t \rightarrow P_3''(n) \quad \text{where } t \in \{HVAC, Open\} \end{cases}$$

Both processes P_3' and P_3'' monitor all the events from the *access point* and $HVAC$ components E_{ap}, E_H. At every execution point, each process either accepts all first-order events or triggers adaptation.

The adaptation function listens for the $\star_v$ and $\star_e$- events and communicates adaptation commands to location ap as needed to ensure the continued satisfaction of Req. 3,

$$F_3 = \square \begin{cases} \star_v\langle n\rangle \rightarrow \text{if } n = 1 \text{ then } ap!A1.ack \rightarrow F_3 \text{ else if } n = 0 \text{ then } ap!A0.ack \rightarrow F_3 \text{ else } ack \rightarrow F_3 \\ \star_e\langle n\rangle \rightarrow \text{if } n = 1 \text{ then } ap!A0.ack \rightarrow F_3 \text{ else if } n = 0 \text{ then } ap!A2.ack \rightarrow F_3 \text{ else } ack \rightarrow F_3 \end{cases}$$

Even when F_3 decides that no adaptation is needed, the process broadcasts the ack-event signalling the end of the adaptation.

The adaptation procedure is defined as the composition of the adaptation patterns P_3', P_3'' and adaptation function F_3, synchronizing over the $\star_v, \star_e$ events and acknowledge ack events,

$$\Pi_3 = (\nu \{\!|\star_v, \star_e|\!\}, ack) \left(\left(P_3'(0) \underset{E_{ap}, E_H}{\|} P_3''(0) \right) \underset{ack, \{\!|\star_v, \star_e|\!\}}{\|} F_3 \right)$$

The process S_3 models CPS, required to verify Req. 3. The process is referred to as the unary

cluster for Req. 3,

$$S_3 = (\nu\, ap, h)\left(\left(\left(ap\langle A0\rangle \underset{E_{ap}}{\parallel} C_{ap}\right) \underset{E_{(ap,h)}}{\parallel} h\langle H\rangle\right) \underset{E_{ap},E_H}{\parallel} \Pi_3\right)$$

where the event set $E_{(ap,h)} = E_{ap} \cap E_H$ represents the connectivity relation—connect and disconnect events between the *access point* and the $HVAC$ components. We later verify that the process S_3 refines the following specification

$$Spec_3 = \text{let}\, T(n) = \Box \begin{cases} n = 0\,\& & conn_{vis} \to disconn_{hvac} \to T(n+1) \\ n > 0\,\& & conn_{vis} \to T(n+1) \\ n > 1\,\& & disconn_{vis} \to T(n-1) \\ n = 1\,\& & disconn_{vis} \to conn_{hvac} \to T(n-1) \\ & & disconnect \to T(0) \\ & & conn_{emp} \to T(n) \\ & & disconn_{emp} \to T(n) \\ n = 0\,\& & conn_{hvac} \to T(n) \end{cases}$$

$$\text{within}\, conn_{hvac} \to T(0)$$

5.3 Summary

In this chapter, we present a technique to utilize topological relationships to systematically explore different grouping of components that can affect—independently from the rest of the system—satisfaction of a given requirement in CPSs. A (self-)adaptation procedure aiming to satisfy a given requirement can be localized to the components that affect its satisfaction. As the state space of these components is typically smaller than that of the whole system , we can use formal verification tools, such as FDR, to check that the components behaviours and the adaptation procedure identified by the system designer can satisfy a given set of requirements. This process is referred to as the *cluster* of a requirement We also present a technique for encoding the adaptation procedure in our process language ACSP, inspired from the SAA framework that tackles the complexity of encoding adaptation procedures by modularize the problem into two processes: a process that decides when the system *must*-adapt and another process that decides the adaptation.

Chapter 6

Step 4: Verification of Requirements in Isolation

Next, the system designer verifies that the adaptation procedures indeed ensure the satisfaction of the requirement. The model of the adaptation procedure and the components in its scope, making up the *unary cluster* can be verified independently from the entire CPS. Because the state space of such sets of components can be significantly smaller than that of the entire system, it is feasible to use formal verification tools. In our verification approach, we use FDR [59], a refinement-checking tool for CSP; however, our technique is general enough to allow the use of other verification tools for process calculi, such as (bi-)simulation, testing preorders, and modal logic techniques (e.g., [89, 108, 69, 43, 35, 11]). If the verification fails, the system designer must explore alternative adaptation procedures, which can be implemented at a different granularity, i.e. across a fewer or more CPS components.

6.1 Theory of the Verification Technique

We present an adequate translation for a well-defined subset of our process language ACSP to CSP. We call this subset the class of *well-formed* processes. The encodings in the previous chapter are all contained in this subset and thus can be translated to CSP and verified using FDR.

6.1.1 Well-formed Processes

The language ACSP is powerful enough to support nested locations, adaptation procedures within locations (which can themselves be adapted) and location redundancy. However for the purposes of this thesis and to simplify the translation of ACSP processes to existing verification tools, we restrict the syntax of the language to *well-formed processes*.

Definition 6.1.1 (Well-Formed Processes)**.** An ACSP process P is well-formed when:

Unique Names: Every location name in P is unique; i.e., every sub-term of P of the form $Q_1 \underset{E}{\parallel} Q_2$ has the property that $in(Q_1) \cap in(Q_2) = \emptyset$.

Flat Structure: Locations are not nested; i.e., every sub-term of P of the form $\square_{i \in \mathcal{I}} Q_i$, (if $e \leq e'$ then Q else Q'), $(e \rightarrow Q)$, $(l!Q.Q')$, $(\mathrm{rec}X(\vec{y} := \vec{e}).Q)$, $l\langle Q \rangle$ does not contain locations in Q, Q', Q_i.

Static Adaptation: Adaptation processes cannot send out processes containing higher-order events; i.e., every sub-term of P of the form $l\langle Q\rangle$, $l!Q.R$ does not contain location outputs (adaptations actions) in Q.

Reasonable Adaptation For every sub-term of P of the form $(\nu L)Q_1 \underset{E}{\parallel} Q_2$, the processes are encapsulated by $(\nu\, L)$ such that all locations in named processes are in one sub-term and the higher-order prefixes are in the other sub-term are scoped in L, $out(Q_1) \cap in(Q_2) \subseteq L$ and $in(Q_1) \cap out(Q_2) \subseteq L$. ◊

Remark 6.1.2. The class of well-formed processes presented in a previous iteration of this work [17] represents a strictly smaller set, where no more than one process could have a higher-order output to a location. Here, we allow such outputs to appear in multiple processes. This is needed to handle overlapping adaptation procedures discussed in the next chapter, which was not handled in [17]. This is done by replacing the condition *single adaptation procedure* in [17] with the condition *reasonable adaptation*. ◊

We say a process is well-formed if we can derive P by the rules in fig. 6.1, starting with an empty Γ, i.e., there are no free locations in the process.

Definition 6.1.3 (Well-Formed Processes)**.** An ACSP process P is well-formed iff $\emptyset \vdash P$ ◊

The rule wPar follows inductively from its sub-term and a number of side-conditions to ensure the satisfaction of the unique name and reasonable adaptation constraints. The rule wPar checks that every location name is unique (*unique name*): $in(Q_1) \cap in(Q_2) = \emptyset$; and that for all locations $l \in L$, the higher-order prefixes of l can be localized in one sub-term and the named process in the other (*reasonable adaptation*): $out(Q_1) \cap in(Q_2) \subseteq L$, $in(Q_1) \cap out(Q_2) \subseteq L$ and $out(Q_1) \cap out(Q_2) \subseteq \Gamma$ (i.e., not in L). Later on in the translation to CSP, we synchronize the higher-order communications on $l \in L$ over the top-level parallel rule $Q_1 \underset{E}{\parallel} Q_2$. In the process with the higher-order outputs Q_2, we allow the interleaving (without synchronization) of multiple higher-order outputs on L. In wLoc, the rule ensures that the process P does not contain nested locations (*flat structure*) or perform higher-order prefixes (*static adaptation*) by checking $\emptyset \vdash P$. The rule wSnd is derived from the well-formedness of the communicated process with respect to an empty environment (*static adaptation* and *flat structure* constraint), and for the continuation we check that the *flat structure* constraint is satisfied. The rules wRec, wChx and wIf all have the side-condition $in(P) = \emptyset$ to guard the constraint *flat structure*. Name scoping is restricted to wPar, whereas we have the rule wScp for first-order events scopes.

The clusters presented in the previous chapter for the art gallery satisfy all the criteria for well-formed processes. We later show that their encoding can also be translated to FDR for verifying the satisfaction of requirements.

Example 6.1.4. Recall the encoding S_1, which models Req. 1,

$$S_1 = (\nu ra, c2)\left(\left(Corr2 \underset{E_{(c_2,ra)}}{\parallel} ResArea\right) \underset{E_{c_2},E_{ra}}{\parallel} \Pi_1\right)$$

The process above satisfies the *unique names* property: the pairwise intersection of $in(Corr2), in(ResArea)$ and $in(\Pi_1)$ is empty. We know that this holds because $in(Corr2) = \{c2\}, in(ResArea) = \{ra\}$ and $in(\Pi_1) = \emptyset$. Locations are at the top level, which satisfies the *flat structure* property. Moreover, all processes communicated through a higher-order output do not contain higher-order outputs themselves (*static adaptation*). Only the process Π_1 performs higher-order outputs in S_1. The

$$\textsc{wChx}\ \frac{i \in \mathcal{I} \text{ implies } \Gamma \vdash P_i \text{ and } in(P_i) = \emptyset}{\Gamma \vdash \Box_{i\in\mathcal{I}}\, e_i \to P_i} \qquad \textsc{wScp}\ \frac{\Gamma \vdash M}{\Gamma \vdash (\nu e)M} \qquad \textsc{wSkp}\ \Gamma \vdash SKIP \qquad \textsc{wRec}\ \frac{\Gamma \vdash P \quad in(P) = \emptyset}{\Gamma \vdash \mathrm{rec}X(\vec{y} := \vec{e}).P}$$

$$\textsc{wIf}\ \frac{\Gamma \vdash P \quad \Gamma \vdash Q \quad in(P) = \emptyset \quad in(Q) = \emptyset}{\Gamma \vdash \text{if } e_1 \le e_2 \text{ then } P \text{ else } Q} \qquad \textsc{wLoc}\ \frac{l \in \Gamma \quad \emptyset \vdash P}{\Gamma \vdash l\langle P\rangle} \qquad \textsc{wApp}\ \Gamma \vdash X(\vec{e}) \qquad \textsc{wSnd}\ \frac{l \in \Gamma \quad \emptyset \vdash P \quad \Gamma \vdash Q \quad in(Q) = \emptyset}{\Gamma \vdash l!P.Q}$$

$$\textsc{wPar}\ \frac{\Gamma \uplus L \vdash Q_1 \quad \Gamma \uplus L \vdash Q_2 \qquad in(Q_2) \cap in(Q_1) = \emptyset \qquad out(Q_1) \cap out(Q_2) \subseteq \Gamma \quad in(Q_2) \cap out(Q_1) \subseteq L \quad in(Q_1) \cap out(Q_2) \subseteq L}{\Gamma \vdash (\nu\, L)Q_1 \underset{E}{\parallel} Q_2}$$

Figure 6.1: Well-Formed Processes Rules

process Π_1 communicates the processes $R_0, R_1, C2_0$ and $C2_1$, where none of them perform any higher-order outputs. Finally, the *reasonable adaptation* procedure is satisfied as the adaptation function within Π_1 only performs higher-order outputs. ◊

Example 6.1.5. Consider the simple example,

$$\emptyset \vdash (\nu\, l)\ \Big((l!P_1 \parallel l!P_2) \quad \Big\| \quad l\langle SKIP\rangle\Big)$$

We check $\{l\} \vdash l!P_1 \parallel l!P_2$ using wPar, wSnd, wSkp and similarly $\{l\} \vdash l\langle SKIP\rangle$ using wLoc, wSkp. Composing the two sub-processes together with wPar follows from four other constraints,

$$\begin{aligned} in(Q_1) \cap in(Q_2) = \emptyset &= in(l!P_1 \parallel l!P_2) \cap in(l\langle SKIP\rangle) = \emptyset \\ in(Q_2) \cap out(Q_1) \subseteq L &= \{l\} \cap \{l\} \subseteq \{l\} \\ in(Q_1) \cap out(Q_2) \subseteq L &= \emptyset \cap \emptyset \subseteq \{l\} \\ out(Q_1) \cap out(Q_2) \subseteq \Gamma &= out(l!P_1 \parallel l!P_2) \cap out(l\langle SKIP\rangle) \subseteq \emptyset \end{aligned}$$

When translating this process to CSP, we can synchronize all the communications over l at the top level parallel construct between the outputs and named process, but internally on the left process $((l!P_1 \parallel l!P_2))$ the outputs are interleaved. ◊

Example 6.1.6. Consider now the ACSP process

$$\emptyset \vdash (\nu\, l)\ \Big((l!P_1 \parallel l\langle SKIP\rangle) \quad \Big\| \quad l!P_2\Big)$$

This process is not well-formed. From the structure of the process, we know that only wPar can be applied. One of the side-condition in wPar is

$$out(Q_1) \cap out(Q_2) \subseteq \Gamma \quad = \quad out((l!P_1 \parallel l\langle SKIP\rangle)) \cap out(l!P_2) \subseteq \emptyset$$

which does not hold because both sides contain the location l. Later on in the translation of this process, we need to synchronize the communication between the named processes and the process performing the higher-order outputs. In this example, we do not have a single parallel construct where we can perform the synchronization between the named process and the interleaving of the higher-order outputs without changing the structure of the process. ◊

We prove *progress* for well-formed processes as a corollary of theorem 6.1.7,

Theorem 6.1.7 (Well-formed Progress)**.** *For a process P, such that $\Gamma \vdash P$, we have:*

- $P \xrightarrow{e} P'$ *implies* $\Gamma \vdash P'$
- $P \xrightarrow{l!R} P'$ *implies* $\Gamma \vdash P'$ *and* $\emptyset \vdash R$ *and* $l \in \Gamma$
- $P \xrightarrow{l?R} P'$ *and* $\emptyset \vdash R$ *implies* $\Gamma \vdash P'$ *and* $l \in \Gamma$

Proof. Follows directly from Lem. B.0.1. □

Well-formedness is preserved by the transition semantics, presented in fig. 4.3, therefore, starting from well-formed processes we only reach well-formed processes.

Corollary 6.1.8. *If P is well-formed and $P \xrightarrow{\alpha} P'$ then P' is well-formed.*

Proof. Follows from the reflexive, transitive closure of theorem 6.1.7. □

We show that any well-formed process with no free locations, i.e., an empty environment, is also closed—its set of traces contains only first-order events.

Theorem 6.1.9. *$\emptyset \vdash P$ implies P is closed.*

Proof. By theorem 6.1.7, we know that for all $t \in \Sigma^*$, $P \xrightarrow{t} P' \xrightarrow{h}$, then $h \in \Gamma$. By the contra-positive of this statement, we know that $h \notin \Gamma$ implies $\neg(P \xrightarrow{t} P' \xrightarrow{h})$. Since we assume an empty Γ, we deduce that for all h, $h \notin \emptyset$ and thus $P \not\xrightarrow{th}$ for all t. □

Proposition 6.1.10. *For well-formed process P and Q and first-order events E, $P \underset{E}{\parallel} Q$ is well-formed.*

Proof. $P \underset{E}{\parallel} Q$ is well-formed iff $\emptyset \vdash P \underset{E}{\parallel} Q$. From the structure of the process, we know that only wPar can be applied. From the lemma definition, we know that $\emptyset \vdash Q$ and $\emptyset \vdash P$. From the side-conditions $in(P) \cap in(Q) = \emptyset$, $in(P) \cap out(Q) = \emptyset$, $out(P) \cap in(Q) = \emptyset$ and $out(P) \cap out(Q) = \emptyset$ hold. From the contra-positive of Pro. B.0.9, we know that

$$in(P) = in(Q) = \emptyset \tag{6.1}$$

$$out(P) = out(Q) = \emptyset \tag{6.2}$$

□

6.1.2 Translation into CSP

In fig. 6.2, we depict the translation of well-formed ACSP processes into CSP processes. This is defined by structural induction on ACSP terms, and presented by judgments of the form $P \rhd S$ translating an ACSP process P to a CSP process S.

By a pre-processing step, we can collect all location names used, and all processes inside higher-order outputs. This is possible because we work with well-formed processes (Def. 6.1.1). We can thus assume an injective map m mapping from higher-order prefixes to distinguished CSP events. We trivially extend this mapping to first-order events, such that $m(e) = e$ for all events e. We also let p be the inverse mapping, taking events back to the process communicated i.e., $p(e) = P$ if there exists a location l where $m(l!P) = e$, or e otherwise. Furthermore, a function ch returns the set of events attached to each location l i.e., $ch(l) = \{e \mid \forall P \in Proc.\, m(l!P) = e\}$.

The rules TChx, tScp, tRec, tIf, tSkp, tApp give a direct mapping to CSP of many ACSP processes. The adaptation mechanism is encoded in the rules tSnd, tLoc and tPar. Rule tSnd

$$\textsc{tChx}\ \frac{i \in \mathcal{I} \text{ implies } P_i \rhd S_i}{\Box_{i\in\mathcal{I}}\, e_i \to P_i \rhd \Box_{i\in\mathcal{I}}\, e_i \to S_i} \qquad \textsc{tScp}\ \frac{M \rhd S}{(\nu e)M \rhd S \setminus \{e\}} \qquad \textsc{tRec}\ \frac{P \rhd S}{\mathrm{rec}X(\vec{y} := \vec{e}).P \rhd \mathrm{let}\, X(\vec{y}) = S \,\mathrm{within}\, X(\vec{e})}$$

$$\textsc{tIf}\ \frac{P \rhd S \quad Q \rhd T}{\mathrm{if}\, e_1 \le e_2 \,\mathrm{then}\, P \,\mathrm{else}\, Q \rhd \mathrm{if}\, e_1 \le e_2 \,\mathrm{then}\, S \,\mathrm{else}\, T} \qquad \textsc{tSkp}\ \frac{}{SKIP \rhd SKIP} \qquad \textsc{tApp}\ \frac{}{X(\vec{e}) \rhd X(\vec{e})} \qquad \textsc{tSnd}\ \frac{m(l!P) = e \quad Q \rhd S}{l!P.Q \rhd e \to S}$$

$$\textsc{tLoc}\ \frac{P \rhd S}{l\langle P\rangle \rhd S \triangle rec(l)} \qquad \textsc{tPar}\ \frac{M \rhd S \quad N \rhd T \quad A = \{m(l!R) \mid R \in Proc, l \in L\}}{(\nu\, L)M \underset{E}{\parallel} N \rhd (S \underset{E,A}{\parallel} T) \setminus A}$$

Figure 6.2: Translation into CSP

translates $l!P.Q$ by prefixing the translation of Q with the event defined in $m(l!P)$. Rule tLoc translates a location l, which is the receiving side of adaptation of l. We utilise the interrupt construct to implement the location: the translation of process P can be interrupted by any event in $ch(l)$. Here we use

$$rec(l) = \Box_{e\in ch(l)}\ e \to (T_e \triangle rec(l))$$

where any $e \in ch(l)$ translates to CSP process T_e by $p(e) \rhd T_e$. This CSP interrupt unfolds $rec(l)$ with every $ch(l)$ event, guaranteeing the execution of the right (translated) process that should run after each adaptation of l, and re-establishing the interrupt.

Example 6.1.11 (Adaptation Processes)**.** Assume the map m such that $m(l!a \to SKIP) = e_1$ and $m(l!b \to SKIP) = e_2$. Then we have the translation

$$l\langle a \to b \to SKIP\rangle \quad \rhd \quad (a \to b \to SKIP) \triangle R$$

where $R = e_1 \to (a \to SKIP \triangle R) \,\Box\, e_2 \to (b \to SKIP \triangle R)$

The process $l!a \to SKIP.SKIP$ initiating an adaptation translates to CSP according to: $l!a \to SKIP.SKIP \quad \rhd \quad e_1 \to SKIP$. ◊

Finally, rule tPar translates a parallel composition $M \underset{E}{\parallel} N$ into a CSP parallel composition. The set of events E is transferred to the CSP parallel, extended with synchronization of events encoding adaptations between M and N over the set of locations L. This prevents the interruption of (translated) locations, encoded in tLoc, if there are no corresponding prefix processes.

Example 6.1.12 (Adaptation)**.** The processes in Ex. 6.1.11 can be composed together using tPar to model a complete adaptable system. Note that the chosen L required by the rule needs to contain at least l as $l \in out\,(l!a \to SKIP.SKIP) \cap in\,(l\langle a \to b \to SKIP\rangle)$. It follows

$$(\nu l)\left(l\langle a \to b \to SKIP\rangle \underset{\emptyset}{\parallel} l!a \to SKIP.SKIP\right) \rhd \left((a \to b \to SKIP) \triangle R \underset{\{e_1,e_2\}}{\parallel} e_1 \to SKIP\right) \setminus \{e_1, e_2\}$$

◊

We prove that the translation is deterministic,

Proposition 6.1.13 (Translation Deterministic)**.** *For a well-formed ACSP process P, $P \rhd S_1$ and $P \rhd S_2$ implies $S_1 = S_2$.*

Proof. Follows from Lem. B.0.4. □

For all well-formed processes P, we show there is a CSP process S derived through the translation in theorem 6.1.14 and that the translation is a strong bisimulation in theorem 6.1.15.

Theorem 6.1.14. $\Gamma \vdash P$ *implies* $P \rhd S$

Proof. By rule induction on $\Gamma \vdash P$

wPar

$$\frac{\Gamma \uplus L \vdash Q_1 \quad \Gamma \uplus L \vdash Q_2 \qquad in(Q_2) \cap in(Q_1) = \emptyset \qquad out(Q_1) \cap out(Q_2) \subseteq \Gamma \quad in(Q_2) \cap out(Q_1) \subseteq L \quad in(Q_1) \cap out(Q_2) \subseteq L}{\Gamma \vdash (\nu\, L)Q_1 \underset{E}{\parallel} Q_2}$$

case *From the rule's premises, we know*

$$\Gamma \uplus L \vdash Q_1 \tag{6.3}$$
$$\Gamma \uplus L \vdash Q_2 \tag{6.4}$$
$$in(Q_2) \cap in(Q_1) = \emptyset \tag{6.5}$$
$$out(Q_1) \cap out(Q_2) \subseteq \Gamma \tag{6.6}$$
$$in(Q_2) \cap out(Q_1) \subseteq L \tag{6.7}$$
$$in(Q_1) \cap out(Q_2) \subseteq L \tag{6.8}$$

By IH with eqs. (6.3) *and* (6.4)*, we know*

$$Q_1 \rhd S_1 \tag{6.9}$$
$$Q_2 \rhd S_2 \tag{6.10}$$

We construct the set of first-order events A such that

$$A = \{m(l!R) \mid R \in Proc, l \in L\} \tag{6.11}$$

Using tPar *with eqs.* (6.9) *to* (6.11)

$$(\nu\, L)Q_1 \underset{E}{\parallel} Q_2 \rhd (S_1 \underset{E,A}{\parallel} S_2) \setminus A \tag{6.12}$$

wSnd

$$\frac{l \in \Gamma \quad \emptyset \vdash P \quad \Gamma \vdash Q \quad in(Q) = \emptyset}{\Gamma \vdash l!P.Q}$$

case *From the premises*

$$l \in \Gamma \tag{6.13}$$
$$\emptyset \vdash P \tag{6.14}$$
$$\Gamma \vdash Q \tag{6.15}$$
$$in(Q) = \emptyset \tag{6.16}$$

By IH eq. (6.15)*,*

$$Q \rhd S \tag{6.17}$$

From eq. (6.14)*, we know that* P *is well-formed and so* $m(l!P)$ *is defined. By* tSnd *with eq.* (6.17)

$$l!P.Q \rhd m(l!P) \to S \tag{6.18}$$

case $\dfrac{\textsc{wLoc}\quad l \in \Gamma \quad \emptyset \vdash P}{\Gamma \vdash l\langle P\rangle}, \dfrac{\textsc{wChx}\quad i \in \mathcal{I} \text{ implies } \Gamma \vdash P_i \text{ and } in(P_i) = \emptyset}{\Gamma \vdash \Box_{i\in\mathcal{I}}\, e_i \to P_i}, \dfrac{\textsc{wRec}\quad \Gamma \vdash P \quad in(P) = \emptyset}{\Gamma \vdash recX(\vec{y} := \vec{e}).P}$

$\dfrac{\textsc{wScp}\quad \Gamma \vdash M}{\Gamma \vdash (\nu e)M}, \dfrac{\textsc{wIf}\quad \Gamma \vdash P \quad \Gamma \vdash Q \quad in(P) = \emptyset \quad in(Q) = \emptyset}{\Gamma \vdash if e_1 \le e_2 \text{ then } P \text{ else } Q}$

Here, we show the proof for wLoc*, the others are similar. From the premises, we know*

$$l \in \Gamma \tag{6.19}$$

$$\emptyset \vdash P \tag{6.20}$$

By IH we know

$$P \rhd S \tag{6.21}$$

By tLoc

$$l\langle P\rangle \rhd S \triangle rec(l) \tag{6.22}$$

case $\Gamma \vdash X(\vec{e}), \Gamma \vdash SKIP$ *immediate.* (wApp, wSkp)

□

We prove that the translation, depicted in fig. 6.2, is a strong bisimulation, i.e., transitions of the ACSP term are in loc-step with the corresponding transitions of the CSP translation presented in fig. 2.1.

Theorem 6.1.15. *For a well-formed ACSP process* M *Let* $M \rhd S$*; then:*

1. *If* $M \xrightarrow{\alpha} M'$ *then there exists* S' *such that* $S \xrightarrow{m(\alpha)} S'$ *and* $M' \rhd S'$
2. *If* $S \xrightarrow{e} S'$ *then there exist* M' *and* α *such that* $m(\alpha) = e$, $M \xrightarrow{\alpha} M'$ *and* $M' \rhd S'$

Proof. Follows directly from Lemma B.0.2 and B.0.3 □

The theorems 6.1.7 and 6.1.15 allow us to conclude that every property of the ACSP is also a property of the translated CSP processes and vice-versa. Thus reasoning in FDR about the translated process leads to verification results about the original ACSP processes. Moreover, we leverage existing algebraic laws defined for CSP to define algebraic laws in our process language and prove that our verification approach is congruent.

6.1.3 Verification Results for the ACSP Process Language

For well-formed processes, we provide a direct correspondence with traces in CSP. This allows us to draw down a mapping between semantic models in the process languages.

Theorem 6.1.16. *We provide a direct correspondence between definitions of traces, failures and divergences for well-formed (closed) ACSP processes, in Def. 4.2.6 and the corresponding translated CSP process S such that $P \rhd S$, presented in section 2.1.2.*

$$\begin{aligned} traces(P) &= traces(S) \\ initials(P) &= initials(S) \\ refusal(P) &= refusal(S) \\ failures(P) &= failures(S) \\ M \Uparrow \ &\ \textit{iff}\ S \Uparrow \\ div(P) &= div(S) \end{aligned}$$

Proof. From the reflexive, transitive closure of of relations established in theorem 6.1.15 and theorem 6.1.9. □

Corollary 6.1.17. *Consider well-formed processes P, Q where $P \rhd S$, $Q \rhd T$. For the semantic models $\mathtt{M} \in \{\mathtt{T}, \mathtt{F}, \mathtt{FD}\}$, then*

$$P \sqsubseteq_{M(ACSP)} Q \ \textit{iff}\ S \sqsubseteq_{M(CSP)} T$$

We show that our verification process is congruent. Proving assertion for a closed part of the system suffices to show that the same property holds for the whole system. In terms of the gallery example, this means that if we verify Requirements 2, 4 and 5 for the exhibition area, the assertion also holds when composing the exhibition area with the rest of the art gallery.

Theorem 6.1.18. *For ACSP well-formed processes P, Q, C and a semantic model $\mathtt{M} \in \{\mathtt{T}, \mathtt{F}, \mathtt{FD}\}$,*

$$P \sqsubseteq_{M(ACSP)} Q \quad \textit{implies} \quad P \underset{E}{\parallel} C \sqsubseteq_{M(ACSP)} Q \underset{E}{\parallel} C$$

Proof. We know that

$$P \rhd S \tag{6.23}$$

$$Q \rhd T \tag{6.24}$$

$$C \rhd G \tag{6.25}$$

From theorem 6.1.15, $P \sqsubseteq_{M(ACSP)} Q$ and eqs. (6.23) and (6.24) we obtain

$$S \sqsubseteq_{M(CSP)} T \tag{6.26}$$

From CSP's congruence theorem in [105], we know that for a set of events

$$S \underset{E}{\parallel} G \sqsubseteq_{M(CSP)} T \underset{E}{\parallel} G \tag{6.27}$$

By Pro. 6.1.10, we know that both $P \underset{E}{\parallel} C$ and $Q \underset{E}{\parallel} C$ are well-formed. This means that there is a process S', T' such that

$$P \underset{E}{\parallel} C \rhd S' \tag{6.28}$$

$$Q \underset{E}{\parallel} C \rhd T' \tag{6.29}$$

From the structure of the processes, we know that the rule tPar must have been used and from Pro. 6.1.13,

$$S' = S \underset{E}{\parallel} G \tag{6.30}$$

$$T' = T \underset{E}{\parallel} G \tag{6.31}$$

By theorem 6.1.15, we know that

$$P \underset{E}{\parallel} C \sqsubseteq_{M(ACSP)} Q \underset{E}{\parallel} C \tag{6.32}$$

□

Example 6.1.19 (Compositionality of Req. 1)**.** Recall, our encoding of ACSP processes S_1 and $Spec_1$ in section 5.2, where S_1 comprises the adaptation procedure Π_1 that adapts and monitors the components for *corridor 2* and *restoration area*. We verify that small part of the system trace refines the specification $Spec_1$.

$$Spec_1 \sqsubseteq_{T(ACSP)} S_1$$

Through the compositionality theorem, we infer that the assertion also holds when we consider the whole gallery,

$$Spec_1 \underset{E_{(c_2,s)}}{\parallel} GroundFloor \sqsubseteq_{T(ACSP)} S_1 \underset{E_{(c_2,s)}}{\parallel} GroundFloor$$

where the process *GroundFloor* is the encoding for the rest of the art gallery, i.e., the *exhibition area* and *stairs* connecting the two floors and the cyber components—*access point* and the *HVAC* component. From the component diagram in fig. 4.1, we know that the *upper floor* is connected to the rest of the gallery through doors between *corridor 2* (c_2)and the *stairs* (s) represented by the event set $E_{(c_2,s)}$. We show that for verifying Req. 1 it suffices to verify only *upper floor*. This reduces the state space of the verification problem significantly. ◊

The bisimulation relation between well-formed process in ACSP and CSP processes allows us to define algebraic laws in our process language. From the bisimulation between ACSP processes and CSP processes and algebraic laws in CSP presented in Def. 2.1.2, we draw down that the following algebraic laws hold in ACSP processes. Here, we prove that the relation = is a strong bisimulation over ACSP processes.

	ACSP Code	CSP Code
Upper Floor	66	80
Exhibition Area	123	138
Access Point & HVAC	62	73

Table 6.1: Difference in lines of codes when translating from ACSP to CSP

Corollary 6.1.20. *For well-formed ACSP processes P, Q, R the following equivalences hold.*

$$P = (P \underset{A}{\parallel} P) \qquad \langle \parallel - \texttt{Identity} \rangle$$

$$\text{provided } ev(P) \subseteq A$$

$$(P \ {}_X\|_Y \ Q) = (Q \ {}_Y\|_X \ P) \qquad \langle {}_A\|_B - \texttt{symm} \rangle$$

$$(P \ {}_X\|_Y \ Q) \ {}_{X\cup Y}\|_Z \ R = P \ {}_X\|_{Y\cup Z} \ (Q \ {}_Y\|_Z \ R) \qquad \langle {}_A\|_B - \texttt{assoc} \rangle$$

$$(P \ {}_A\|_B \ Q) \setminus Z = (P \setminus Z \cap A) \ {}_A\|_B \ (Q \setminus Z \cap B) \qquad \langle \texttt{hide} - {}_A\|_B - \texttt{dist} \rangle$$

$$\text{provided } A \cap B \cap Z = \emptyset$$

$$\left(P \underset{A}{\parallel} Q\right) \setminus Z = (P \setminus Z) \underset{A}{\parallel} (Q \setminus Z) \qquad \langle \texttt{hide} - \underset{A}{\parallel} - \texttt{dist} \rangle$$

$$\text{provided } A \cap Z = \emptyset$$

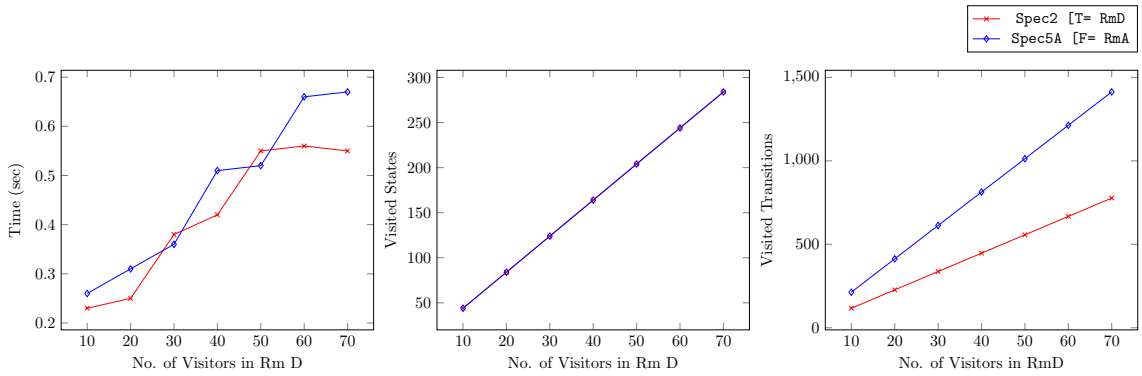
6.2 First Evaluation of the Verification Technique

In table 6.1, we depict the change in lines of code between the processes modelled in our language and the translated code in FDR for the *upper floor*, *exhibition area* and *cyber components*. Translating the system components of the *Restoration Area* and *Corridor 2*, modelled in our language as process ($\star$) in section 5.2, leads to an FDR file of 80 lines of code. In our example, the difference is minimal because adaptations alternates between two or three processes.

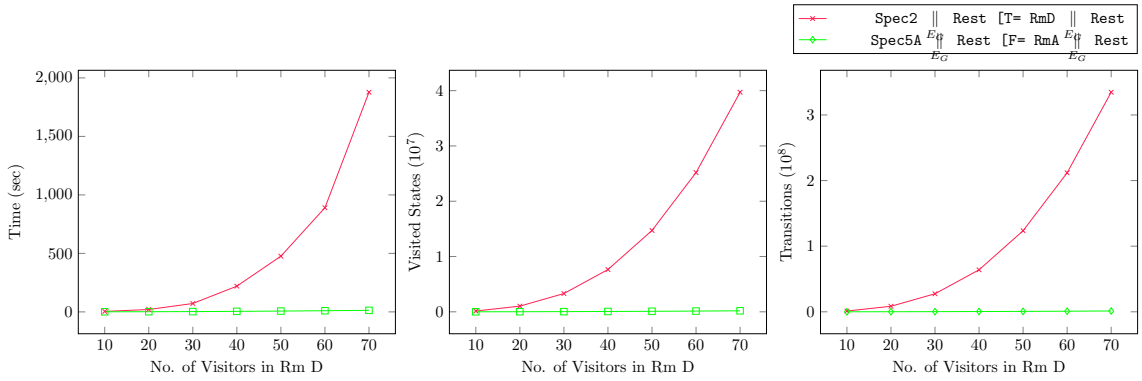
The verification of ($\star$) was done by encoding simple specification automata in CSP accepting a language of correct traces, and then showing that the translation of ($\star$) has a subset of these traces using FDR trace assertions. One of these specification processes does not include traces where a visitor accesses the restoration area without a guard present. Another shows that the guard does not leave the *Corridor 2* if there are still visitors in the second floor.

We investigate the scalability of our verification approach by measuring the order of growth of running time in seconds, number of states and transitions for verifying the specifications $Spec_2$, $Spec_4$ and $Spec_{5A}$. We verify all three assertions by first considering only the selected subset of components and then by considering the whole art gallery and see how topology-driven modelling help us attain a tractable verification approach for SA CPSs. We acknowledge that the standard verification approach may differ from the one used in the comparison presented in this section, and further experiments may be required to build a complete picture of the scalability of the new approach.

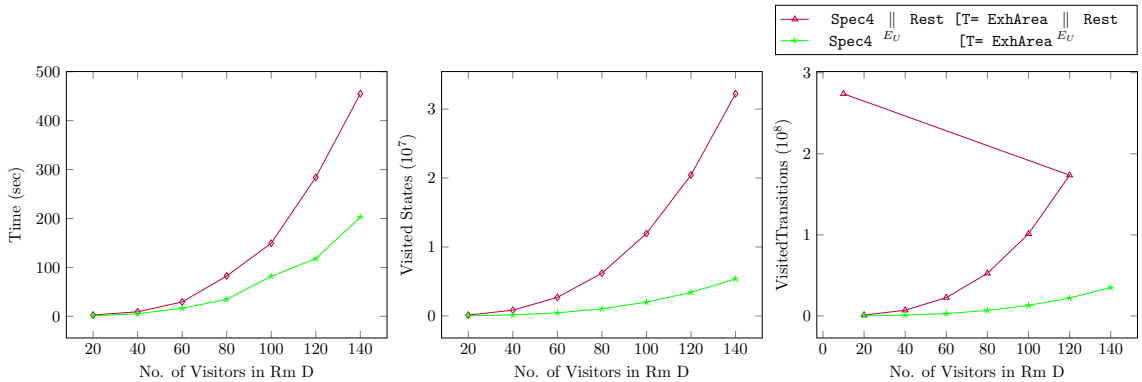
We run the three assertions in both contexts independently in FDR for an increasing number of allowed visitors in *room D* (Req. 2) and *exhibition area* (Req. 4). For simplicity, we assume that the number of visitors in the *exhibition area* is twice the number of allowed visitors in *room D*. For each verification, we measure the running time (in seconds) using Unix *time* command and measure the number of visited states and transition from FDR output. We run the experiment on a personal computer having 8 cores running at 3.4 GHz with 8GB of DDR RAM and on a server that has 56 cores running at 2.2 GHz with 256GB of RAM. The results match and therefore we



(a) Experimental results for verifying Requirements 2 and 5 for *room A* using FDR including only the relevant components. In the graphs, the x-axis specify the number of visitors allowed in *room D* at any time. FDR minimization is turned on for this experiment.

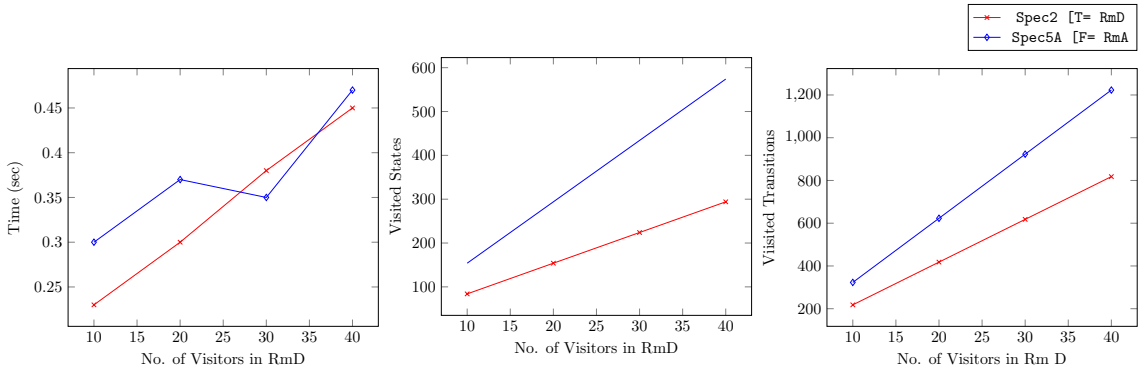


(b) Experimental results for verifying Requirements 2 and 5 for *room A* using FDR including all components in the art gallery. The green line corresponding to the verification of $Spec_{5A}$ grows slower than the red line ($Spec_2$)

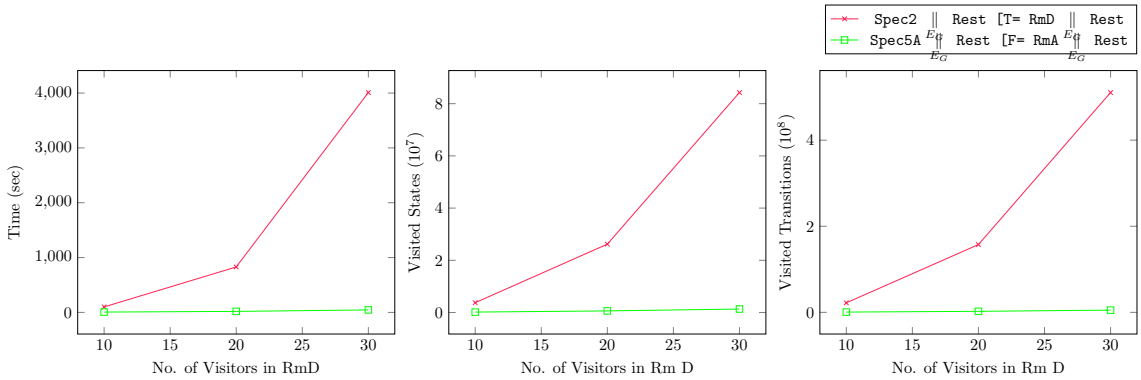


(c) Experimental results for verifying Req. 4 in isolation. The x-axis specify the number of visitors we allow in the *exhibition area*. We compare how topology-driven modelling performs in comparison to verifying the whole model.

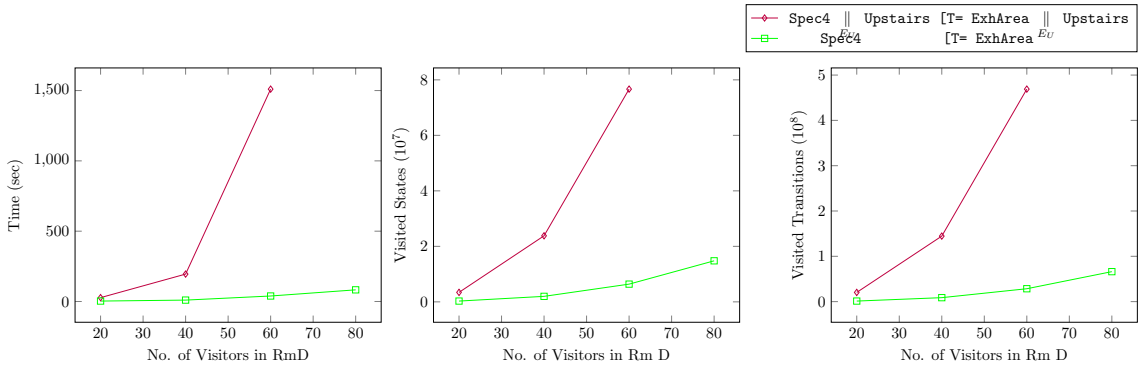
Figure 6.3: Experimental results for topology-guided modelling of the exhibition area requirements with FDR.



(a) Experimental results for verifying Requirements 2 and 5 for *room A* when FDR minimization is turned off and we include only the relevant components. In the graphs, the x-axis specify the number of visitors allowed in *room D* at any time. FDR minimization is turned on for this experiment.



(b) Experimental results for verifying Requirements 2 and 5 for *room A* when FDR minimization is disabled and we include all components in the art gallery. The green line corresponding to the verification of $Spec_{5A}$ grows slower than the red line ($Spec_2$).



(c) Experimental results for verifying Req. 4 when FDR minimization is disabled. The x-axis specify the number of visitors we allow in the *exhibition area*. We compare how topology-driven modelling performs in comparison to verifying the whole model.

Figure 6.4: Experimental results for verifying topology-guided modelling of exhibition area requirements with FDR minimization disabled.

only show the results derived from the execution on the server machine.

In fig. 6.3a for $Spec_2, Spec_{5A}$ and fig. 6.3c for $Spec_4$, we show the order of growth of running time (seconds), number of visited states and transitions in the assertion when the verification comprises only of the selected subset of components. In fig. 6.3b for $Spec_2, Spec_{5A}$ and fig. 6.3c for $Spec_4$, we show the same order of growths for the same assertions but considering the whole art gallery. From the figure, it is clear that topology-driven modelling scales more. In particular, consider assertion `Spec2 [T= RmD` in which we allow 70 visitors in *room D* is verified in 0.55 seconds after having visited 284 states and 777 transitions. When the same assertion is composed in parallel with the rest of the art gallery, the verification takes 1,877.46 seconds and needs to visit 39,711,720 states and 334,594,252 transitions. This improvement in running time is because the processes are much smaller and thus FDR needs to check much fewer states and transitions to verify assertions.

Our verification approach leverages a well-established verification tool—FDR, which allows us to subsume years of experience in optimizing the verification process. For instance, many of the transitions are internal transitions encoding adaptation transitions, which FDR eliminates using a normalization command. We investigate the extent the normalization command helps our verification approach performance and its impact on the order of growth by running the same assertions (both considering only the selected subset of components and the whole art gallery) and disabling any minimization techniques performed by FDR. The results are shown in fig. 6.4. Similar to the previous figure, we show the assertions verified by considering only the selected subset of components fig. 6.4a for $Spec_2, Spec_{5A}$ and fig. 6.4c for $Spec_4$ and the assertions with whole art gallery encoded in fig. 6.4b for $Spec_2, Spec_{5A}$ and fig. 6.4c for $Spec_4$. Even though the order of growth is preserved, FDR minimization allows our verification approach to scale more. Consider the assertion `Spec2 [T= RmD` again, with no minimization, FDR takes 45.45 seconds to verify 30 visitors in the *room D* after visiting 1,337,712 states and 5,165,393 transitions. The processes after minimization are noticeably smaller, which means FDR has fewer states and transitions to visit in order to verify assertions leading to better running times. This is a clear advantage of leveraging existing tools that has established optimization techniques to improve the performance of verification. Moreover, FDR allows us to run the verification in a GUI or command line environment and also verification is automatically parallelized and commands to distribute the verification task to distributed clusters are provided.

6.3 Summary

As the state space of *clusters* is typically smaller compared to that of the whole system, we can use formal verification tools, such as FDR, to check that the components behaviour and the adaptation procedure identified by the system designer can satisfy given requirements. We also provide an adequate translation from a subset of our language to FDR to perform verification of self-adaptive CPSs. We showcase our approach using a substantive art gallery example. Our results demonstrate that our approach has the benefit of reducing the memory and time required to verify properties of the self-adaptive CPSs. Our technique for discovering a good level of granularity for an adaptation procedure that ensures satisfaction of system requirements can reduce the size of components that need to be verified.

Chapter 7

Step 5 & 6: Composition and Re-verification of Overlapping Adaptation Procedures

In our proposed methodology, we have thus far explored the satisfaction of requirements by verifying the correctness of the adaptation procedures designed to enforce them *in isolation*, considering each adaptation procedure with only the components in its scope. When adaptation procedures have disjoint scopes then this verification is sufficient to ensure requirement satisfaction in the entire system. We have proved this result based on a compositionality theorem (theorem 6.1.18 in section 6.1.3). However, when adaptation procedure scopes overlap, there is the potential of *interference* between them. In this case we may need to re-verify that each requirement is satisfied when all adaptation procedures, and all system components, are composed back together.

In this chapter we develop a theory of compositionality results showing that interference is not possible in certain types of scope overlaps, thus allowing us to skip re-verification tasks when composing the system together. We do this by expanding on Step 5 from our proposed methodology in section 4.1 to systematically merge together clusters and providing theoretical non-interference results. We summarise these results in table 7.1. Note that $S \otimes S'$ denotes the merging of two clusters, formally defined in Def. 7.2.1.

7.1 Examples

Prior to presenting a general approach for systematically identifying interfering adaptation procedures, we list in table 7.1, different types of overlaps and how each type affects the satisfaction of the requirements. As a convention in the dissertation, we write Π_A^M to mean the adaptation procedure Π in which the sets of locations A and M are adapted and monitored by Π respectively. We adapt the same convention for clusters. In a cluster S_A^M locations A and M are adapted and monitored respectively.

The first row, *case 0*, represents when there are no overlaps between the clusters. This case is discussed in Chapter 5 where the specifications are independent of each other, e.g., Requirements 1 to 3. For this type of overlap, we later prove that the satisfaction of the two requirements by the composition follows from the compositionality theorem and thus the verification of the two requirements in isolation suffices to ensure their satisfaction.

In the next row, *Case 1*, both adaptation procedures monitor a common component $M_1 \cap$

Case	Clusters Overlap Type	Assertions that Follow ☺	Need to Reverify ☹
0	No Overlaps $(A_1 \cup M_1) \cap (A_2 \cup M_2) = \emptyset$	$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_1 $S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_2	–
1	Monitored $A_1 \cap (A_2 \cup M_2) = \emptyset$ $A_2 \cap (A_1 \cup M_1) = \emptyset$ $M_1 \cap M_2 \neq \emptyset$	$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_1 $S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_2	–
2	Adapted & Monitored $A_1 \cap (A_2 \cup M_2) = \emptyset$ $M_1 \cap A_2 \neq \emptyset$	$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_2	$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_1
3	Adapted & Monitored $A_1 \cap M_2 \neq \emptyset$ $A_2 \cap M_1 \neq \emptyset$ $A_1 \cap A_2 = \emptyset$	–	$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_1 $S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_2
4	Adapted $A_1 \cap A_2 \neq \emptyset$	–	$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_1 $S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ satisfies R_2

Table 7.1: Examples of different ways the clusters $S_{A_1}^{M_1}$ and $S_{A_2}^{M_2}$ can overlap and how the composition of the clusters can affect the satisfaction of requirements. Here, we assume that in Step 4 from our methodology we verified $S_{A_1}^{M_1}$ satisfies R_1 and $S_{A_2}^{M_2}$ satisfies R_2.

$M_2 \neq \emptyset$. Neither of the adaptation procedures change the behaviour of shared components—$A_1 \cap (A_2, M_2) = A_2 \cap (A_1, M_1) = \emptyset$. For this type of overlap, we later prove that the satisfaction of the two requirements also follows from the compositionality theorem.

Example 7.1.1. Consider the well-formed clusters S_a^b, S_c^b and ACSP specifications $Spec_1, Spec_2$ verified in isolation through the assertions,

$$Spec_1 \sqsubseteq_{M(ACSP)} (\nu\, a, b)\left((a\langle A\rangle \parallel b\langle B\rangle) \parallel \Pi_a^b\right) = S_a^b$$
$$Spec_2 \sqsubseteq_{M(ACSP)} (\nu\, c, b)\left((c\langle C\rangle \parallel b\langle B\rangle) \parallel \Pi_c^b\right) = S_c^b$$

Neither adaptation procedure change the behaviour at location b. Interference between the overlapping adaptation procedures Π_a^b and Π_c^b is not possible. We later show that the satisfaction of $Spec_1$ and $Spec_2$ follows from the compositionality theorem. ◊

The next row, *case 2*, the scopes overlap because $A_2 \cap M_2 \neq \emptyset$. This means that the adaptation procedure $\Pi_{A_2}^{M_2}$ in the cluster $S_{A_2}^{M_2}$ changes the behaviour of location b that is relevant to the satisfaction of $Spec_1$. We need to verify, using FDR, that the satisfaction of $Spec_1$ is preserved, when location b is adapted by $\Pi_{A_2}^{M_2}$. Because $S_{A_1}^{M_1}$ does not change the behaviour of any components relevant to the satisfaction of $Spec_2$, we later prove that the satisfaction of $Spec_2$ when $S_{A_1}^{M_1}$ and $S_{A_2}^{M_2}$ are composed follows from the compositionality theorem.

Example 7.1.2. Let us now consider Requirements 2 and 4 from the art gallery example in section 1.2.

Requirement 2: No more than 10 visitors should be in *Room D* at the same time.

Requirement 4: No more than 20 visitors (in total) should be in the exhibition area.

The requirements are meant to be satisfied by Π_2 and Π_4, respectively, as explained in section 5.2. The adaptation procedure Π_2 adapts the behaviour of *room D* whereas the adaptation procedure Π_4 adapts the behaviour of *room A* and monitors *rooms B* and *D*. For the requirements we verified that the assertions below hold [1]

$$Spec_2 \sqsubseteq_{T(ACSP)} (\nu\, l_d)\left(l_d\langle R'_D\rangle \underset{E_D}{\parallel} \Pi_2\right)$$

$$Spec_4 \sqsubseteq_{T(ACSP)} (\nu\, l_a, l_b, l_d)\left(\left(l_A\langle R'_A\rangle \underset{E_{(A,B)}}{\parallel} l_B\langle R'_B\rangle \underset{E_{(B,D)}}{\parallel} l_D\langle R'_D\rangle\right) \underset{E_A,E_B,E_D}{\parallel} \Pi_4\right)$$

The verification of Req. 4 assumes that the behaviour of *room D* does not change. This is not the case because Π_2 adapts the behaviour of *room D* to ensure the satisfaction of Req. 2. We thus need to re-verify Req. 4 replacing $l_D\langle R'_D\rangle$ with the process $l_D\langle R'_D\rangle \underset{E_d}{\parallel} \Pi_2$ to make sure that Req. 4 is satisfied with the accurate implementation of *room D*,

$$Spec_4 \sqsubseteq_{T(ACSP)} (\nu\, l_a, l_b)\left(\left(l_A\langle R'_A\rangle \underset{E_{(A,B)}}{\parallel} l_B\langle R'_B\rangle\right) \underset{E_{(B,D)}}{\parallel} (\nu\, l_d)\left(l_D\langle R'_D\rangle \underset{E_d}{\parallel} \Pi_2\right)\right) \underset{E_A,E_B,E_D}{\parallel} \Pi_4 \quad \texttt{(REVERIFY)}$$

The satisfaction of Req. 2 is not affected by the behaviour of *room A* (the component adapted by Π_4). Its satisfaction when Π_2 and Π_4 are composed together, follows from the compositionality theorem. We can define a *closed* well-formed process such that we can compose it in parallel with the assertion as a context. The assertion below follows by compositionality,

$$(\nu\, l_a, l_b)\left(\left(\left(l_A\langle R'_A\rangle \underset{E_{(A,B)}}{\parallel} l_B\langle R'_B\rangle\right) \underset{E_A,E_B}{\parallel} \Pi_4\right) \underset{E_{(B,D)}\cup E_d}{\parallel} Spec_2 \quad \sqsubseteq_{T(ACSP)}\right.$$

$$\left.(\nu\, l_a, l_b)\left(\left(\left(l_A\langle R'_A\rangle \underset{E_{(A,B)}}{\parallel} l_B\langle R'_B\rangle\right) \underset{E_A,E_B}{\parallel} \Pi_4\right) \underset{E_{(B,D)}\cup E_d}{\parallel} (\nu\, l_d)\left(l_d\langle R'_D\rangle \underset{E_d}{\parallel} \Pi_2\right)\right) \quad \texttt{(FOLLOWS)}\right) \quad \Diamond$$

In the next row, *case 3*, the adaptation procedures may potentially interfere with each other. In this overlap type, both adaptation procedures adapt locations that are monitored by the other. The satisfaction of R_1 needs to be reverified because a component in M_1 is adapted by $S^{M_2}_{A_2}$— $A_2 \cap M_1 \neq \emptyset$. Dually, the satisfaction of R_2 needs to be reverified because a component in M_2 is adapted by $S^{M_1}_{A_1}$.

In the last row, *case 4*, the adaptation procedures adapt common components. Because adaptation is arbitrary we need to ensure that different adaptations do not conflict and/or introduce violations.

Example 7.1.3. Recall Requirements 2 and 5 for *room D* in the art gallery example.

Requirement 2: No more than 10 visitors should be in *Room D* at the same time.

Requirement 5 for room D: In an emergency, all doors should be open

In section 5.2. we define the adaptation procedures Π_2 and Π_{5D} and verified that the adaptation procedures indeed satisfy Requirements 2 and 5 for *room D* respectively. More precisely, we verified

[1] For simplicity, we omit the internal state processes for components

the assertions below,

$$Spec_2 \quad \sqsubseteq_{T(ACSP)} \quad (\nu\, l_d)\left(l_d\langle R'_D\rangle \underset{E_d}{\parallel} \Pi_2\right)$$

$$Spec_{5D} \quad \sqsubseteq_{F(ACSP)} \quad (\nu\, l_d)\left(l_d\langle R'_D\rangle \underset{E_d}{\parallel} \Pi_{5D}\right)$$

Because both adaptation procedures Π_2 and Π_{5D} adapt the behaviour of the named process at l_d, their composition may potentially introduce interference and/or violations. Using FDR, we can infer that the composition of Π_2 and Π_{5D} violates Req. 5 for *room D*. In the case of an emergency, once there are 10 visitors in *room D*, Π_2 closes the entrance to *room D*, which violates Req. 5. We resolve the conflict by updating Π_2 to suppress adaptation during an emergency.

The specifications of Requirements 2 and 5 in the processes $Spec_2$ and $Spec_{5D}$ also needs to be updated. Req. 5 is a liveness property, *visitors should not be refused exit from any room.* This means that any refusals in the implementation must be present in $Spec_{5D}$. The specification $Spec_2$ refuses entrance to *room D* when the number of visitors inside is 10. This refusal, even though occur in a non-emergency state, still needs to be added to $Spec_{5D}$. Req. 2 does not specify that it should only be applied in a non-emergency situation. This is reflected in specification $Spec_2$ where at most 10 visitors are allowed in *room D* irrespective of whether there is an emergency. For the verification to be effective the specification needs to mirror closely the requirements. Here, we need to update both Req. 2 and specification $Spec_2$ to only apply in a non-emergency. Assume $Spec'_2$ and $Spec'_{5D}$ are the updated specifications and Π'_2 is the updated adaptation procedure. Then, we reverify the composition of adaptation procedures Π'_2 and Π_{5D} against both$Spec'_2$ and $Spec'_{5D}$.

$$Spec'_2 \quad \sqsubseteq_{T(ACSP)} \quad (\nu\, l_d)\left(l_d\langle R'_D\rangle \underset{E_d}{\parallel} \left(\Pi'_2 \underset{E_d}{\parallel} \Pi_{5D}\right)\right) \qquad \texttt{(REVERIFY)}$$

$$Spec'_{5D} \quad \sqsubseteq_{F(ACSP)} \quad (\nu\, l_d)\left(l_d\langle R'_D\rangle \underset{E_d}{\parallel} \left(\Pi'_2 \underset{E_d}{\parallel} \Pi_{5D}\right)\right) \qquad \texttt{(REVERIFY)}$$

$\Diamond$

We now present a technique to systematically compose a number of clusters together and identifying which specification needs to be reverified when the clusters are composed together and which follow from the compositionality theorem.

7.2 Cluster Composition

Our art gallery example comprises 6 requirements. As discussed earlier, some of the adaptation procedures adapt the behaviour of the same components, e.g., Π_2, Π_{5D} in Ex. 7.1.3, both adapt the behaviour of *room D*. Because of potential interference between overlapping adaptation procedures, we need to verify that the composition of adaptation procedures preserves the satisfaction of system requirements. In Chapters 5 and 6 we have verified that unary clusters (subset of system components together with one adaptation procedure) satisfy specifications, which describe requirements. Here we define a *merge* operator that composes together two clusters and returns a well-formed ACSP process which comprises the two adaptation procedures and the union of their components, put in parallel; the result is also a cluster. The operator allows us to systemically compose clusters together and obtain ACSP processes that can still be translated to FDR according to theorem 6.1.14. We then present a theory of overlaps between clusters, which shows the cases

where requirement satisfaction is preserved by the composition and the cases where re-verification is needed.

7.2.1 The *merge* Operation

Here, we define the *merge* operation which composes together two clusters and returns a *well-formed* cluster that can still be translated to FDR.

Definition 7.2.1 (Merge Operation)**.** Consider two clusters $S_{A_1}^{M_1}$ and $S_{A_2}^{M_2}$. The merge operation, written as $S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$, returns a well-formed process that takes the union of the adapted components, monitored components and the adaptation procedures. We here overview each element of the cluster. Firstly, the set of adapted components in a merged cluster is the union of the individual adapted components,

$$A_\otimes = (A_1 \cup A_2)$$

Secondly, the set of monitored components for $S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ includes the monitored components from both adaptation procedures that are not adapted by any of the adaptation procedures. If a component is monitored by one of the adaptation procedures but is adapted by the other adaptation procedure, the component becomes part of the adapted components.

$$M_\otimes = (M_1 \cup M_2) \setminus (A_1 \cup A_2)$$

Finally, the merge operation composes (in parallel) the adaptation procedures, synchronizing them over first-order events of common components

$$\Pi_{A_\otimes}^{M_\otimes} \quad = \quad \Pi_{A_1}^{M_1} \underset{(E_{A_1}, E_{M_1}) \cap (E_{A_2}, E_{M_2})}{\parallel} \Pi_{A_2}^{M_2}$$

The cluster of the composition is thus defined as

$$S_{A_\otimes}^{M_\otimes} = (\nu\, A_\otimes, M_\otimes) \left(\left(\Big\| _{l \in A_\otimes, M_\otimes} \; l_l\langle P_l \rangle \right) \underset{E_{A_\otimes}, E_{M_\otimes}}{\parallel} \Pi_{A_\otimes}^{M_\otimes} \right)$$

◊

Example 7.2.2. Recall the clusters S_2 and S_4 from Ex. 7.1.2,

$$\begin{aligned} S_2 =& (\nu\, l_D) \left(l_D\langle R'_D \rangle \underset{E_D}{\parallel} \Pi_2 \right) \\ S_4 =& (\nu\, l_B, l_D) \left(\left(l_A\langle R'_A \rangle \underset{E_{(A,B)}}{\parallel} l_B\langle R'_B \rangle \underset{E_{(B,D)}}{\parallel} l_D\langle R'_D \rangle \right) \underset{E_D, E_B, E_A}{\parallel} \Pi_4 \right) \end{aligned}$$

The merged cluster $S_2 \otimes S_4$ is

$$\begin{aligned} \Pi_{l_A, l_D}^{l_B} &= \Pi_2 \underset{E_D}{\parallel} \Pi_4 \\ S_2 \otimes S_4 &= (\nu\, l_A, l_B, l_D) \left(\left(l_A\langle R'_A \rangle \underset{E_{(A,B)}}{\parallel} l_B\langle R'_B \rangle \underset{E_{(B,D)}}{\parallel} l_D\langle R'_D \rangle \right) \underset{E_A, E_B, E_D}{\parallel} \left(\Pi_{l_A, l_D}^{l_B} \right) \right) \end{aligned}$$

The adapted components are now locations $\{l_A, l_D\}$ and location l_B is the monitored component. ◊

Case	Clusters Overlap Type	Assertions that Follow ☺	Need to Reverify ☹
0	No Overlaps $(A_1, M_1) \cap (A_2, M_2) = \emptyset$	$Spec_1 \underset{E}{\parallel} S_{A_2}^{M_2} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ $Spec_2 \underset{E}{\parallel} S_{A_1}^{M_1} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$	–
1	Monitored $A_1 \cap (A_2, M_2) = \emptyset$ $A_2 \cap (A_1, M_1) = \emptyset$ $M_1 \cap M_2 \neq \emptyset$	$Spec_1 \underset{E}{\parallel} S_{A_2}^{M_2 \setminus M_1} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ $Spec_2 \underset{E}{\parallel} S_{A_1}^{M_1 \setminus M_2} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$	–
2	Adapted & Monitored $A_1 \cap (A_2, M_2) = \emptyset$ $M_1 \cap A_2 \neq \emptyset$	$Spec_2 \underset{E}{\parallel} S_{A_1}^{M_1 \setminus M_2} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$	$Spec_1' \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$
3	Adapted & Monitored $A_1 \cap M_2 \neq \emptyset$ $A_2 \cap M_1 \neq \emptyset$ $A_1 \cap A_2 = \emptyset$	–	$Spec_1' \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ $Spec_2' \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$
4	Adapted $A_1 \cap A_2 \neq \emptyset$	–	$Spec_1' \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$ $Spec_2' \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$

Table 7.2: Clusters Overlaps and how their composition affect the satisfaction of requirements. Here, we assume that in Step 4 from our methodology we verified $S_{A_1} \sqsubseteq Spec_1$ and $S_{A_2}^{M_2} \sqsubseteq Spec_2$. Composition may require us to update specifications. In cases where the satisfaction is in the *follow* column, we have a mechanical way to update the specification, otherwise this is left up to the system designer.

We show that the merge operator defined in Def. 7.2.1 preserves the well-formed property.

Theorem 7.2.3. *Given well-formed clusters S_1 and S_2, $S_1 \otimes S_2$ is well-formed.*

Proof. Follows directly from Lem. B.0.13 □

Lemma 7.2.4. *We show that the following properties holds for well-formed clusters S_1, S_2 and S_3,*

$$S_1 \otimes S_2 = S_2 \otimes S_1 \qquad \langle \otimes - \texttt{sym} \rangle$$
$$S_1 \otimes (S_2 \otimes S_3) = (S_1 \otimes S_2) \otimes S_3 \qquad \langle \otimes - \texttt{assoc} \rangle$$

Proof. Follows from Lem. B.0.17. □

7.2.2 What needs to be re-verified?

In table 7.2, we examine how different types of overlaps between two adaptation procedures scopes affect the satisfaction of the two requirements. In particular, we analyse different types of overlaps between monitored and adapted components to identify interfering adaptation procedures. For overlaps where interference is not possible, we prove that the satisfaction of the requirements follows from the compositionality theorem. Moreover, the composition of clusters may require us

to update the encoding of specifications. For non-interfering composition, where the satisfaction of specification follows from the compositionality theorem, we outline the encoding of the updated specification, otherwise for potentially interfering composition, this task is left up to the system designer.

The first row in table 7.2, *case 0*, applies when the components in the clusters do not overlap. Here, the satisfaction of both requirements when the two clusters are composed together follow from the compositionality theorem. Pro. 7.2.5 also dictates how the specifications are to be updated. In this type of overlap, the specification composes in parallel the other cluster, e.g., $Spec_1' = Spec_1 \underset{E}{\parallel} S_{A_2}^{M_2}$ and that the parallel composition of clusters is equivalent (up to strong bisimulation) to the application of the merge operation.

Proposition 7.2.5. *$Spec_1 \sqsubseteq S_{A_1}^{M_1}$ and $Spec_2 \sqsubseteq S_{A_2}^{M_2}$ such that $(A_1, M_1) \cap (A_2, M_2) = \emptyset$ implies*

$$Spec_1' = Spec_1 \underset{E}{\parallel} S_{A_2}^{M_2} \quad \sqsubseteq \quad S_{A_1}^{M_1} \underset{E}{\parallel} S_{A_2}^{M_2} = S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} \qquad \texttt{(FOLLOWS)}$$

$$Spec_2' = Spec_2 \underset{E}{\parallel} S_{A_1}^{M_1} \quad \sqsubseteq \quad S_{A_2}^{M_2} \underset{E}{\parallel} S_{A_1}^{M_1} = S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} \qquad \texttt{(FOLLOWS)}$$

where $E = (E_{A_2}, E_{M_2}) \cap (E_{A_1}, E_{M_1})$ is the set of events that connect components in clusters $S_{A_1}^{M_1}$ and $S_{A_2}^{M_2}$

Proof. Follows directly from Lem. B.0.15 □

In the next row, *case 1*, the adaptation procedures monitor common components—$M_1 \cap M_2 \neq \emptyset$. The satisfaction of both $Spec_1$ and $Spec_2$ is preserved because the adaptation procedures adapt disjoint components—$A_1 \cap (A_2, M_2) = \emptyset$ and $A_2 \cap (A_1, M_1) = \emptyset$. The adaptation procedure in $S_{A_1}^{M_1}$ does not change the behaviour of any components that affect the satisfaction of $Spec_2$ and vice versa.

In Pro. 7.2.6, we prove the satisfaction of both specifications is preserved and also dictate how the specifications are to be updated as a result of the composition.

Proposition 7.2.6. *$Spec_1 \sqsubseteq S_{A_1}^{M_1}$ and $Spec_2 \sqsubseteq S_{A_2}^{M_2}$ such that $A_1 \cap (A_2, M_2) = \emptyset$, $A_2 \cap (A_1, M_1) = \emptyset$ implies*

$$Spec_1' = Spec_1 \underset{E}{\parallel} S_{A_2}^{M_2 \setminus M_1} \quad \sqsubseteq \quad S_{A_1}^{M_1} \underset{E}{\parallel} S_{A_2}^{M_2 \setminus M_1} = S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} \qquad \texttt{(FOLLOWS)}$$

$$Spec_2' = Spec_2 \underset{E}{\parallel} S_{A_1}^{M_1 \setminus M_2} \quad \sqsubseteq \quad S_{A_2}^{M_2} \underset{E}{\parallel} S_{A_1}^{M_1 \setminus M_2} = S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} \qquad \texttt{(FOLLOWS)}$$

where $E = (E_{A_2}, E_{M_2}) \cap (E_{A_1}, E_{M_1})$ is the set of events that connect components in clusters $S_{A_1}^{M_1}$ and $S_{A_2}^{M_2}$. The cluster $S_{A_2}^{M_2 \setminus M_1}$ excludes the locations in M_1 from its definition to remove duplicate locations.

Proof. Follows directly from Lem. B.0.15 □

In the next row, *case 2* (and its omitted symmetric case), one of the clusters adapts the behaviour of a location monitored by the other. In this case, we need to reverify the satisfaction of $Spec_1$ to ensure its satisfaction is preserved when the locations in $A_2 \cap M_1$ are adapted by $S_{A_2}^{M_2}$. The cluster $S_{A_1}^{M_1}$ does not change the behaviour of locations that affect the satisfaction of $Spec_2$ and thus we prove in Pro. 7.2.7 the satisfaction of $Spec_2$ follows from the compositionality theorem. The proposition define how the specification $Spec_2$ changes because of the composition, however the update of $Spec_1$ is left up to the designer as potential interference may require the requirement or adaptation procedure to be changed.

Proposition 7.2.7. *$Spec_1 \sqsubseteq S_{A_1}^{M_1}$ and $Spec_2 \sqsubseteq S_{A_2}^{M_2}$ such that $A_1 \cap (A_2, M_2) = \emptyset$ implies*

$$Spec_2' = Spec_2 \underset{E}{\parallel} S_{A_1}^{M_1 \setminus (M_2, A_2)} \quad \sqsubseteq \quad S_{A_2}^{M_2} \underset{E}{\parallel} S_{A_1}^{M_1 \setminus (M_2, A_2)} = S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} \qquad (\texttt{FOLLOWS})$$

where $E = (E_{A_2}, E_{M_2}) \cap (E_{A_1}, E_{M_1})$ is the set of events that connect components in clusters $S_{A_1}^{M_1}$ and $S_{A_2}^{M_2}$

Proof. Follows directly from Lem. B.0.15 □

In the next row, *case 3*, both clusters adapt the behaviour of a component monitored by the other. This kind of overlap means that both specifications needs to be reverified.

The last row, *case 4*, applies when both adaptation procedures adapt and change the behaviour of common components. We need to verify that the adaptation procedures do not interfere with each other and introduce conflicting adaptations. In this case, the specifications may need to be updated but this task is done manually by the system designer.

The conditions in table 7.2 ensures that only one row applies for a single composition.

7.2.3 Composing multiple overlapping Scopes

In the art gallery exhibition area, we may need to compose together clusters corresponding to five adaptation procedures—$\{S_2, S_4, S_{5A}, S_{5B}, S_{5D}\}$. The *merge* operator composes two clusters and returns a new cluster. The returned cluster can again be composed with another cluster using the *merge* operator. Therefore, the composition of multiple clusters is derived through the repeated application of the *merge* operator. At each composition, we verify the satisfaction of the relevant specifications. This way, we detect conflicts early on and can resolve conflicts before composing even more clusters.

The order of composition affects the verification efforts required to verify the satisfaction of requirements when clusters are composed together. The verification effort is determined by the number of assertions that needs to be verified, the computational and memory cost of each assertion and the efforts needed to resolve discovered conflicts.

There is no clear indication which ordering strategy minimizes the verification efforts required. The investigation of the effect of composition ordering on the verification efforts is left as future work. A system designer may decide to first compose "closer" cluster, because closer clusters have a higher probability of conflicting. This may be determined by some defined distance function. This approach increases the likelihood that conflicts are detected and resolved early. Alternatively, a system designer may compose the most distant clusters first because it is computational less costly.

In this thesis, we consider two case-studies: the art gallery and smart stadium. In each case-study, we implement a different composition order that minimizes our verification efforts. For the art gallery case-study, we aim to minimize the number of verification tasks. We acknowledge that simply counting the number of verification tasks may not necessarily reflect the verification effort or computation time and hence we leave the definition of a systematic ordering for composition as future work.f In table 7.3, we depict the number of assertions required to verify the satisfaction of Requirements 2, 4 and 5 for *rooms A,B, D* with two different orderings of composition. In the left table we opt for a natural ordering of composition. This order require us to reverify each specification with each composition. All compositions overlap over adapted components, which is *case 4* in table 7.2 requiring us to reverify all specifications. In the right table, we minimize the number of times we verify the same specifications. For example, we do not compose S_4 with

Step	Clusters	Overlap Type	Verification Tasks	Count
1	S_{5A}	–	$Spec_{5A} \sqsubseteq_{F(ACSP)} S_{5A}$	1
2	S_{5B}	–	$Spec_{5B} \sqsubseteq_{F(ACSP)} S_{5B}$	1
3	S_{5D}	–	$Spec_{5D} \sqsubseteq_{F(ACSP)} S_{5D}$	1
4	S_2	–	$Spec_2 \sqsubseteq_{T(ACSP)} S_2$	1
5	S_4	–	$Spec_4 \sqsubseteq_{T(ACSP)} S_4$	1
6	$S_{\otimes_6} = S_2 \otimes S_4$	4	$Spec'_2 \sqsubseteq_{T(ACSP)} S_{\otimes_6}$ $Spec_4 \sqsubseteq_{T(ACSP)} S_{\otimes_6}$	2
7	$S_{\otimes_7} = S_{5A} \otimes S_{\otimes_6}$	4	$Spec''_2 \sqsubseteq_{T(ACSP)} S_{\otimes_7}$ $Spec'_4 \sqsubseteq_{T(ACSP)} S_{\otimes_7}$ $Spec'_{5A} \sqsubseteq_{F(ACSP)} S_{\otimes_7}$	3
8	$S_{\otimes_8} = S_{5B} \otimes S_{\otimes_7}$	4	$Spec''_2 \sqsubseteq_{T(ACSP)} S_{\otimes_8}$ $Spec'_4 \sqsubseteq_{T(ACSP)} S_{\otimes_8}$ $Spec'_{5A} \sqsubseteq_{F(ACSP)} S_{\otimes_8}$ $Spec'_{5B} \sqsubseteq_{F(ACSP)} S_{\otimes_8}$	4
9	$S_{\otimes_9} = S_{5D} \otimes S_{\otimes_8}$	2	$Spec'''_2 \sqsubseteq_{T(ACSP)} S_{\otimes_9}$ $Spec'_4 \sqsubseteq_{T(ACSP)} S_{\otimes_9}$ $Spec'_{5A} \sqsubseteq_{F(ACSP)} S_{\otimes_9}$ $Spec'_{5B} \sqsubseteq_{F(ACSP)} S_{\otimes_9}$ $Spec'_{5D} \sqsubseteq_{F(ACSP)} S_{\otimes_9}$	5
			Total	**18**

Step	Clusters	Overlap Type	Verification Tasks	Count
1	S_{5A}	–	$Spec_{5A} \sqsubseteq_{F(ACSP)} S_{5A}$	1
2	S_{5B}	–	$Spec_{5B} \sqsubseteq_{F(ACSP)} S_{5B}$	1
3	S_{5D}	–	$Spec_{5D} \sqsubseteq_{F(ACSP)} S_{5D}$	1
4	S_2	–	$Spec_2 \sqsubseteq_{T(ACSP)} S_2$	1
5	S_4	–	$Spec_4 \sqsubseteq_{T(ACSP)} S_4$	1
6	$S_{\otimes_6} = S_{5D} \otimes S_2$	4	$Spec'_{5D} \sqsubseteq_{F(ACSP)} S_{\otimes_6}$ $Spec'_2 \sqsubseteq_{T(ACSP)} S_{\otimes_6}$	2
7	$S_{\otimes_7} = S_{5B} \otimes S_{\otimes_6}$	0	–	0
8	$S_{\otimes_8} = S_{5A} \otimes S_4$	4	$Spec'_{5A} \sqsubseteq_{F(ACSP)} S_{\otimes_8}$ $Spec'_4 \sqsubseteq_{T(ACSP)} S_{\otimes_8}$	2
9	$S_{\otimes_9} = S_{\otimes_8} \otimes S_{\otimes_7}$	2	$Spec'_{5A} \sqsubseteq_{F(ACSP)} S_{\otimes_9}$ $Spec'_4 \sqsubseteq_{T(ACSP)} S_{\otimes_9}$	2
			Total	**11**

Table 7.3: Example of how the ordering affect the number of assertions that needs to be re-verified. In the table we illustrate how specifications are affected by the composition of adaptation procedures. Refer to section 7.3 for the encoding of the specifications.

$S_2 \otimes S_{5D}$ because *room B* monitored in S_4 is adapted by S_{5B}. We first compose S_{5B} with $S_2 \otimes S_{5D}$ before adding S_4. This way, we only verify the correctness of S_4 once, rather than when composing $S_2 \otimes S_{5D}$ and again when composing S_{5B}.

As we shall see in a later chapter, adaptation procedures in the stadium case-study are all defined at the same level of granularity comprising of the same adapted components and mostly the same monitored components[2]. Because this matches with *case 4* in table 7.2, which requires us to reverify all specifications with every composition, the ordering there does not impact heavily the verification efforts required.

7.3 Cluster Composition in the Art Gallery

We now apply our composition approach to the art gallery example. Recall, in section 5.2 we verify that the unary clusters $S_1 - S_6$ satisfies Requirements 1 to 6 respectively. We now verify that requirements satisfaction is preserved when all adaptation procedures are composed together. We first group the art gallery specifications into three: the exhibition area which includes $\{S_2, S_4, S_{5A}, S_{5B}, S_{5D}\}$, the access point with overlaps from $\{S_3, S_6\}$ and the restoration area which includes the clusters $\{S_1, S_{5RA}, S_{5C_2}\}$ and then compose the groups of *merged* clusters together to infer that all requirements in the art gallery are satisfied when all adaptation procedures and system components are composed together.

7.3.1 The Exhibition Area

We first discuss the composition of clusters defined over the exhibition area. We adapt the ordering shown in table 7.3 (right) to reduce the number of reverifications needed.

[2]refer to table 9.1 for a depiction of overlaps in the stadium case-study.

Composition of S_2 and S_{5D} Here, we verify the composition of S_2 and S_{5D}, where both adapt the component *room D*. From table 7.2, this matches *case 4*, which means that we need to re-verify, using FDR, that both $Spec_2$ and $Spec_{5D}$ are still satisfied by $S_2 \otimes S_{5D}$.

$$Spec_2 \sqsubseteq_{T(ACSP)} S_2 \otimes S_{5D} \quad \text{(REVERIFY)}$$
$$Spec_{5D} \sqsubseteq_{F(ACSP)} S_2 \otimes S_{5D} \quad \text{(REVERIFY)}$$

As explained in Ex. 7.1.3, this composition violates both specifications. We thus need to resolve the violations before continuing the composition process.

The specification $Spec_{5D}$ is violated because it is a liveness property and all refusals in the implementation need to be included in the specification. This should include refusals by the adaptation procedure Π_2 during a non-emergency. We therefore update the encoding of specification $Spec_{5D}$ to be the cluster S_2 in a non-emergency situation and switch to $Spec_{5D}$ on the *emergency* event.

$$Spec'_{5D} = S_2 \triangle (emergency \rightarrow RUN(E_D))$$

The specification $Spec'_{5D}$ is still violated because the adaptation procedure Π_2 refuses further visitors' entry to *room D* if the number of visitors in *room D* reaches 10, even in an emergency situation. We thus update the adaptation procedure Π_2 to suppress adaptations during an emergency. This change is within the adaptation pattern, where we do not trigger adaptations after an *emergency* event.

$$P'_2(n) = \Box \begin{cases} vis_{(EB,ED)} \rightarrow \text{if } n+1 \geq 10 \text{ then } \star\langle n+1\rangle \rightarrow ack \rightarrow P'_2(n+1) \text{ else } P'_2(n+1) \\ vis_{(ED,c)} \rightarrow \text{if } n-1 \geq 9 \text{ then } \star\langle n-1\rangle \rightarrow ack \rightarrow P'_2(n-1) \text{ else } P'_2(n-1) \\ emergency \rightarrow RUN(E_D) \end{cases}$$

Assume S'_2 is the updated cluster which replaces the adaptation pattern in S_2 with P'_2. We verify (in isolation) the updated adaptation procedure satisfies $Spec_2$ as explained in Step 4 in our proposed methodology. The specification $Spec_2$ is violated because after an emergency event, the implementation allows more than 10 visitors in *room D*. Because this is actually correct behaviour, this points to a conflict in the requirements. Specifications mirror faithfully requirements and any changes to the specification need to confirmed against the requirements. Here, we update Req. 2 to state

Requirement 2. (revised) In a non-emergency, no more than 10 visitors should be in *Room D* at the same time. ◊

We now update $Spec_2$ to only describe the behaviour of *room D* in a non-emergency situation as explained in the requirement. After an *emergency* event, the process encodes the most permissive process where all events from *room D* are accepted indefinitely.

$$Spec'_2 = (Spec_2 \triangle (emergency \rightarrow RUN(E_D)))$$

Using FDR, we now verify that Requirements 2 and 5 for *room D* are satisfied by the composition $S'_2 \otimes S_{5D}$ through the assertions below.

$$Spec'_2 \sqsubseteq_{T(ACSP)} S'_2 \otimes S_{5D} \quad \text{(REVERIFY)}$$
$$Spec'_{5D} \sqsubseteq_{F(ACSP)} S'_2 \otimes S_{5D} \quad \text{(REVERIFY)}$$

Composition of $S_2' \otimes S_{5D}$ and S_{5B} We next compose $S_2' \otimes S_{5D}$ and S_{5B}. By looking at the components in the clusters, we know that *room D* is adapted by $S_2' \otimes S_{5D}$ and *room B* is adapted by S_{5B}. This means that the two clusters do not overlap which matches *case 0* in table 7.2. From Pro. 7.2.5, the satisfaction of the specifications $Spec_2', Spec_{5D}'$ and $Spec_{5B}$ after composition follows from the compositionality theorem.

$$Spec_{5B}' = \left(Spec_{5B} \underset{\substack{E_{(B,D)} \\ emergency}}{\parallel} S_2' \otimes S_{5D} \right) \sqsubseteq_{F(ACSP)} \left(S_{5B} \underset{\substack{E_{(B,D)} \\ emergency}}{\parallel} S_2' \otimes S_{5D} \right) = S_{5B} \otimes S_2' \otimes S_{5D} \quad \texttt{(FOLLOWS)}$$

$$Spec_{5D}'' = \left(Spec_{5D}' \underset{\substack{E_{(B,D)} \\ emergency}}{\parallel} S_{5B} \right) \sqsubseteq_{F(ACSP)} \left(S_2' \otimes S_{5D} \underset{\substack{E_{(B,D)} \\ emergency}}{\parallel} S_{5B} \right) = S_{5B} \otimes S_2' \otimes S_{5D} \quad \texttt{(FOLLOWS)}$$

$$Spec_2'' = \left(Spec_2' \underset{\substack{E_{(B,D)} \\ emergency}}{\parallel} S_{5B} \right) \sqsubseteq_{T(ACSP)} \left(S_2' \otimes S_{5D} \underset{\substack{E_{(B,D)} \\ emergency}}{\parallel} S_{5B} \right) = S_{5B} \otimes S_2' \otimes S_{5D} \quad \texttt{(FOLLOWS)}$$

Composition of S_{5A} and S_4 We now compose the clusters S_{5A} and S_4. Since both clusters adapt *room A*, this scenario matches *case 4* in table 7.2. This means that both specifications $Spec_{5A}$ and $Spec_4$ need to be reverified to ensure both are satisfied by $S_4 \otimes S_{5A}$. We need to update $Spec_{5A}$ to include *rooms B,D* so it matches the scope of $S_4 \otimes S_{5A}$.

$$Spec_{5A} = Spec_{5A} \underset{E_{(A,B)}}{\parallel} R_B' \underset{E_{(B,D)}}{\parallel} R_D'$$

We verify the assertions,

$$Spec_{5A} \sqsubseteq_{F(ACSP)} S_4 \otimes S_{5A} \quad \texttt{(REVERIFY)}$$

$$Spec_4 \sqsubseteq_{T(ACSP)} S_4 \otimes S_{5A} \quad \texttt{(REVERIFY)}$$

Similar to the composition of S_2' and S_{5D}, both specifications are violated. We need to change $Spec_{5A}$ to include refusals enforced by the adaptation procedure Π_4 in a non-emergency situation, because $Spec_{5A}$ describes a liveness property.

$$Spec_{5A}' = (S_4 \triangle (emergency \rightarrow RUN(E_A, E_B, E_D)))$$

We also need to update the adaptation procedure Π_4 to suppress adaptations after the *emergency* event by changing its adaptation pattern,

$$P_4'(n) = \Box \begin{cases} vis_{(c,EA)} \rightarrow \text{if}(n+1 \geq 16) \text{ then } \star\langle n+1 \rangle \rightarrow ack \rightarrow P_4'(n+1) \text{ else } P_4'(n+1) \\ vis_{(EA,c)} \rightarrow \text{if}(n = 20) \text{ then } \star\langle n-1 \rangle \rightarrow ack \rightarrow P_4(n-1) \text{ else if } n > 0 \text{ then } P_4'(n-1) \text{ else } P_4'(0) \\ vis_{(ED,c)} \rightarrow \text{if}(n = 20) \text{ then } \star\langle n-1 \rangle \rightarrow ack \rightarrow P_4(n-1) \text{ else if } n > 0 \text{ then } P_4'(n-1) \text{ else } P_4'(0) \\ vis_{(EB,ED)} \rightarrow P_4'(n) \\ vis_{(EA,EB)} \rightarrow P_4'(n) \\ vis_{(EB,EA)} \rightarrow P_4'(n) \\ emergency \rightarrow RUN(E_D, E_B, E_A) \end{cases}$$

Assume S'_4 is the updated cluster, where the adaptation pattern is replaced by P'_4. We verify the satisfaction of $Spec_4$ by S'_4 (in isolation). The specification $Spec_4$ is violated because it does not account for the emergency state as it is not included in Req. 4. We therefore update both Req. 4 and the specification $Spec_4$ to include the non-emergency prerequisite.

Requirement 4. (revised) In a non-emergency, no more than 20 visitors (in total) should be in the exhibition areas at the same time. ◊

We now update $Spec_4$ such that it accepts all movements after the *emergency* event, to mirror as closely as possible the requirement,

$$Spec'_4 = (Spec_4 \triangle (emergency \rightarrow RUN(E_A, E_B, E_D)))$$

Using FDR, we verify that Requirements 4 and 5 for *room A* are both satisfied by $S_4 \otimes S_{5A}$ through the following assertions,

$$Spec'_{5A} \sqsubseteq_{F(ACSP)} S'_4 \otimes S_{5A} \quad \texttt{(REVERIFY)}$$
$$Spec'_4 \sqsubseteq_{T(ACSP)} S'_4 \otimes S_{5A} \quad \texttt{(REVERIFY)}$$

Composition of $S'_2 \otimes S_{5D} \otimes S_{5B}$ and $S'_4 \otimes S_{5A}$ Lastly, we compose all clusters in the exhibition area together. By looking at the components in each cluster, we infer that $S'_2 \otimes S_{5D} \otimes S_{5B}$ adapts *rooms B, D*, whereas $S'_4 \otimes S_{5A}$ adapts *room A* and monitors *rooms B, D*. This matches *case 2*, where the satisfaction of specifications $Spec''_2$, $Spec''_{5D}$ and $Spec'_{5B}$ follows from the compositionality theorem but we need to reverify]AAB[using FDR] the satisfaction of the specifications $Spec'_4$ and $Spec'_{5A}$ because the components monitored are adapted by the other cluster. We verify that

$$Spec'_4 \sqsubseteq_{T(ACSP)} ((S'_2 \otimes S_{5D}) \otimes S_{5B}) \otimes (S'_4 \otimes S_{5A}) \quad \texttt{(REVERIFY)}$$
$$Spec'_{5A} \sqsubseteq_{F(ACSP)} ((S'_2 \otimes S_{5D}) \otimes S_{5B}) \otimes (S'_4 \otimes S_{5A}) \quad \texttt{(REVERIFY)}$$

From Pro. 7.2.7, we know that the assertions below follow from compositionality,

$$Spec'''_2 = \left(Spec''_2 \underset{E_B,E_D}{\parallel} S'_4 \otimes S_{5A}{}^{\backslash l_B, l_D}\right) \sqsubseteq_{T(ACSP)} \left(S'_2 \otimes S_{5D} \otimes S_{5B} \underset{E_B,E_D}{\parallel} S'_4 \otimes S_{5A}{}^{\backslash l_B, l_D}\right) = S'_2 \otimes S_{5D} \otimes S_{5B} \otimes S'_4 \otimes S_{5A} \quad \texttt{(FOLLOWS)}$$
$$Spec'''_{5D} = \left(Spec''_{5D} \underset{E_B,E_D}{\parallel} S'_4 \otimes S_{5A}{}^{\backslash l_B, l_D}\right) \sqsubseteq_{F(ACSP)} \left(S'_2 \otimes S_{5D} \otimes S_{5B} \underset{E_B,E_D}{\parallel} S'_4 \otimes S_{5A}{}^{\backslash l_B, l_D}\right) = S'_2 \otimes S_{5D} \otimes S_{5B} \otimes S'_4 \otimes S_{5A} \quad \texttt{(FOLLOWS)}$$
$$Spec''_{5B} = \left(Spec'_{5B} \underset{E_B,E_D}{\parallel} S'_4 \otimes S_{5A}{}^{\backslash l_B, l_D}\right) \sqsubseteq_{F(ACSP)} \left(S'_2 \otimes S_{5D} \otimes S_{5B} \underset{E_B,E_D}{\parallel} S'_4 \otimes S_{5A}{}^{\backslash l_B, l_D}\right) = S'_2 \otimes S_{5D} \otimes S_{5B} \otimes S'_4 \otimes S_{5A} \quad \texttt{(FOLLOWS)}$$

7.3.2 The Access Point

Here, we verify that the satisfaction of Requirements 3 and 6 is preserved by the composition of the clusters S_3 and S_6. In Req. 3, we disconnect the *HVAC* if a visitor is connected to the *access point*. In the presence of a pending update, we however temporarily disconnect visitors to give enough time for the *HVAC* to install the updates. In section 5.2, we verified the following two assertions,

$$Spec_3 \sqsubseteq_{T(ACSP)} (\nu\, ap, h)\left(\left(ap\langle A0\rangle \underset{E_{(ap,h)}}{\parallel} h\langle H\rangle\right) \underset{E_{ap},E_H}{\parallel} \Pi_3\right)$$
$$Spec_6 \sqsubseteq_{T(ACSP)} (\nu\, ap, h)\left(\left(ap\langle A0\rangle \underset{E_{(ap,h)}}{\parallel} h\langle H\rangle\right) \underset{E_{ap},E_H}{\parallel} \Pi_6\right)$$

Both adapt the *access point* component and thus their composition matches *case 4* from table 7.2. We use the *merge* operator to compose and verify the assertions again.

$$Spec_3 \sqsubseteq_{T(ACSP)} (\nu\, ap, h)\left(\left(ap\langle A0\rangle \underset{E_{(ap,h)}}{\parallel} h\langle H\rangle\right) \underset{E_{ap},E_H}{\parallel} \left(\Pi_3 \underset{E_{AP},E_H}{\parallel} \Pi_6\right)\right) = S_3 \otimes S_6 \quad \texttt{(REVERIFY)}$$

$$Spec_6 \sqsubseteq_{T(ACSP)} (\nu\, ap, h)\left(\left(ap\langle A0\rangle \underset{E_{(ap,h)}}{\parallel} h\langle H\rangle\right) \underset{E_{ap},E_H}{\parallel} \left(\Pi_3 \underset{E_{AP},E_H}{\parallel} \Pi_6\right)\right) = S_3 \otimes S_6 \quad \texttt{(REVERIFY)}$$

7.3.3 The Restoration Area

Here, we have three overlapping clusters $\{S_1, S_{5C_2}, S_{5RA}\}$.

Composition of S_{5C_2} and S_{5RA} We first compose the clusters S_{5C_2} and S_{5RA}, where S_{5C_2} adapts *corridor 2* and S_{5RA} adapts the *restoration area*. From table 7.2, this is *case 0*, where components do not overlap. From Pro. 7.2.5, we know that the satisfaction of both $Spec_{5C_2}$ and $Spec_{5RA}$ by $S_{5C_2} \otimes S_{5RA}$ follows from the compositionality theorem.

$$Spec'_{5C_2} = \left(Spec_{5C_2} \underset{\substack{E_{(c_2,ra)}\\ emergency}}{\parallel} S_{5RA}\right) \sqsubseteq_{F(ACSP)} \left(S_{5C_2} \underset{\substack{E_{(c_2,ra)}\\ emergency}}{\parallel} S_{5RA}\right) = S_{5C_2} \otimes S_{5RA} \quad \texttt{(FOLLOWS)}$$

$$Spec'_{5RA} = \left(Spec_{5RA} \underset{\substack{E_{(c_2,ra)}\\ emergency}}{\parallel} S_{5C_2}\right) \sqsubseteq_{F(ACSP)} \left(S_{5RA} \underset{\substack{E_{(c_2,ra)}\\ emergency}}{\parallel} S_{5C_2}\right) = S_{5C_2} \otimes S_{5RA} \quad \texttt{(FOLLOWS)}$$

Composition of S_1 and $S_{5C_2} \otimes S_{5RA}$ We now compose the clusters S_1 and $S_{5C_2} \otimes S_{5RA}$. Both clusters adapt the behaviour of *corridor 2* and the *restoration area*. From table 7.2, this matches *case 4*, where all specifications in the two clusters needs to be reverified. We verify, using FDR, that the assertions below hold,

$$Spec'_{5C_2} \quad \sqsubseteq_{F(ACSP)} \quad S_1 \otimes S_{5C_2} \otimes S_{5RA} \quad \texttt{(REVERIFY)}$$

$$Spec'_{5RA} \quad \sqsubseteq_{F(ACSP)} \quad S_1 \otimes S_{5C_2} \otimes S_{5RA} \quad \texttt{(REVERIFY)}$$

$$Spec_1 \quad \sqsubseteq_{T(ACSP)} \quad S_1 \otimes S_{5C_2} \otimes S_{5RA} \quad \texttt{(REVERIFY)}$$

Akin to the composition of the *exhibition area*, the specifications are violated. The specifications $Spec'_{5C_2}$ and $Spec'_{5RA}$ are liveness properties and thus need to include refusals enforced by other adaptation procedures in a non-emergency situation.

$$Spec''_{5C_2} = Spec''_{5RA} = (S_1 \triangle (emergency \rightarrow RUN(E_{ra}, E_{c_2})))$$

We also need to update the adaptation procedure Π_1 to suppress adaptations after the *emergency* event as adaptations from Π_{5C_2} and Π_{5RA} take precedence

$$P_1(v,g) = \Box \begin{cases} vis_{(c_2,\mathrm{ra})} \rightarrow & \text{if } v = 0 \text{ then } \star\langle v+1, g\rangle \rightarrow ack \rightarrow P_1(v+1,g) \\ & \quad \text{else } P_1(v+1,g) \\ vis_{(\mathrm{ra},c_2)} \rightarrow & \text{if } v = 1 \text{ then } \star\langle v-1, g\rangle \rightarrow ack \rightarrow P_1(v-1,g) \\ & \quad \text{else if } n > 0 \text{ then } P_1(v-1,g) \\ & \quad \text{else } P_1(0) \\ grd_{(c_2,s)} \rightarrow & \text{if } g > 0 \text{ then } \star\langle v, g-1\rangle \rightarrow ack \rightarrow P_1(v,g-1) \\ & \quad \text{else } P_1(v,g-1) \\ grd_{(s,c_2)} \rightarrow & \star\langle v, g+1\rangle \rightarrow ack \rightarrow P_1(v,g+1) \\ e \rightarrow & P_1(v,g) \text{ where } e \in E_{ra}, E_{c_2} \backslash \{vis_{(c_2,\mathrm{ra})}, vis_{(\mathrm{ra},c_2)}, grd_{(c_2,s)}, grd_{(s,c_2)}\} \\ emergency \rightarrow & RUN(E_{ra}, E_{c_2}) \end{cases}$$

Assume that S_1' is the updated cluster where the adaptation pattern in S_1 is replaced by the process above. Akin to Requirements 2 and 4, the specification $Spec_1$ is violated because Req. 1 does not constraint the requirement to non-emergency state only. We thus update Req. 1 to state,

Requirement 1. (revised) In a non-emergency, visitors should not interfere with the restoration process ◊

We now update $Spec_1$ to be as permissive as possible after an *emergency* event, by allowing all events from the *restoration area* and *corridor 2* indefinitely,

$$Spec_1' = (Spec_1 \triangle (emergency \rightarrow RUN(E_{ra}, E_{c_2})))$$

Using FDR, we verify that Requirements 1 and 5 for *corridor 2* and *restoration area* are satisfied when all adaptation procedures are composed together, by verifying the assertions below,

$$\begin{aligned} Spec_{5C_2}'' &\sqsubseteq_{F(ACSP)} S_1' \otimes S_{5C_2} \otimes S_{5RA} && \texttt{(REVERIFY)} \\ Spec_{5RA}'' &\sqsubseteq_{F(ACSP)} S_1' \otimes S_{5C_2} \otimes S_{5RA} && \texttt{(REVERIFY)} \\ Spec_1' &\sqsubseteq_{T(ACSP)} S_1' \otimes S_{5C_2} \otimes S_{5RA} && \texttt{(REVERIFY)} \end{aligned}$$

7.3.4 The entire Art Gallery

Lastly, we show that the satisfaction of all requirements hold when all adaptation procedures are composed together. Here, we compose the adaptation procedures in exhibition area, access point and restoration area. We show that the satisfaction of requirements by the composition of all adaptation procedures follow from the compositionality theorem. Assume the following processes,

$$S_{EA} = S_2' \otimes S_{5D} \otimes S_{5B} \otimes S_4' \otimes S_{5A} \quad \text{and} \quad S_{AP} = S_3 \otimes S_6, \quad \text{and} \quad S_{RA} = S_1' \otimes S_{5C_2} \otimes S_{5RA}$$

Composition of S_{RA} and S_{AP} By looking at the components in each cluster, we know that the two clusters do not overlap, which matches *case 0* from table 7.2.

From Pro. 7.2.5, we infer that

$$Spec''_1 = S_{AP} \,|||\, Spec'_1 \sqsubseteq_{T(ACSP)} S_{AP} \,|||\, S_{RA} = S_{AP} \otimes S_{RA} \quad \texttt{(FOLLOWS)}$$
$$Spec'''_{5RA} = S_{AP} \,|||\, Spec''_{5RA} \sqsubseteq_{F(ACSP)} S_{AP} \,|||\, S_{RA} = S_{AP} \otimes S_{RA} \quad \texttt{(FOLLOWS)}$$
$$Spec'''_{5C_2} = S_{AP} \,|||\, Spec''_{5C_2} \sqsubseteq_{F(ACSP)} S_{AP} \,|||\, S_{RA} = S_{AP} \otimes S_{RA} \quad \texttt{(FOLLOWS)}$$
$$Spec'_3 = Spec_3 \,|||\, S_{RA} \sqsubseteq_{T(ACSP)} S_{AP} \,|||\, S_{RA} = S_{AP} \otimes S_{RA} \quad \texttt{(FOLLOWS)}$$
$$Spec'_6 = Spec_6 \,|||\, S_{RA} \sqsubseteq_{T(ACSP)} S_{AP} \,|||\, S_{RA} = S_{AP} \otimes S_{RA} \quad \texttt{(FOLLOWS)}$$

Composition of $S_{AP} \otimes S_{RA}$ and S_{EA} By looking at the components in each cluster, we know that the two cluster do not overlap and it matches *case 0* from table 7.2. From Pro. 7.2.5, we know that the requirements satisfaction follows from the compositionality theorem, akin to the previous composition.

$$Spec'''_1 = Spec''_1 \,|||\, S_{EA} \sqsubseteq_{T(ACSP)} (S_{AP} \otimes S_{RA}) \,|||\, S_{EA} = (S_{AP} \otimes S_{RA}) \otimes S_{EA} \quad \texttt{(FOLLOWS)}$$
$$Spec^{IV}_{5RA} = Spec'''_{5RA} \,|||\, S_{EA} \sqsubseteq_{F(ACSP)} (S_{AP} \otimes S_{RA}) \,|||\, S_{EA} = (S_{AP} \otimes S_{RA}) \otimes S_{EA} \quad \texttt{(FOLLOWS)}$$
$$Spec^{IV}_{5C_2} = Spec'''_{5C_2} \,|||\, S_{EA} \sqsubseteq_{F(ACSP)} (S_{AP} \otimes S_{RA}) \,|||\, S_{EA} = (S_{AP} \otimes S_{RA}) \otimes S_{EA} \quad \texttt{(FOLLOWS)}$$
$$Spec''_3 = Spec'_3 \,|||\, S_{EA} \sqsubseteq_{T(ACSP)} (S_{AP} \otimes S_{RA}) \,|||\, S_{EA} = (S_{AP} \otimes S_{RA}) \otimes S_{EA} \quad \texttt{(FOLLOWS)}$$
$$Spec''_6 = Spec'_6 \,|||\, S_{EA} \sqsubseteq_{T(ACSP)} (S_{AP} \otimes S_{RA}) \,|||\, S_{EA} = (S_{AP} \otimes S_{RA}) \otimes S_{EA} \quad \texttt{(FOLLOWS)}$$
$$Spec^{IV}_2 = Spec'''_2 \,|||\, (S_{AP} \otimes S_{RA}) \sqsubseteq_{T(ACSP)} S_{EA} \,|||\, (S_{AP} \otimes S_{RA}) = (S_{AP} \otimes S_{RA}) \otimes S_{EA} \quad \texttt{(FOLLOWS)}$$
$$Spec^{IV}_{5D} = Spec'''_{5D} \,|||\, (S_{AP} \otimes S_{RA}) \sqsubseteq_{F(ACSP)} S_{EA} \,|||\, (S_{AP} \otimes S_{RA}) = (S_{AP} \otimes S_{RA}) \otimes S_{EA} \quad \texttt{(FOLLOWS)}$$
$$Spec'''_{5B} = Spec''_{5B} \,|||\, (S_{AP} \otimes S_{RA}) \sqsubseteq_{F(ACSP)} S_{EA} \,|||\, (S_{AP} \otimes S_{RA}) = (S_{AP} \otimes S_{RA}) \otimes S_{EA} \quad \texttt{(FOLLOWS)}$$

7.4 Revisiting the Evaluation of the Verification Technique

We investigate again the scalability of our verification approach by measuring the time, number of states and number of transitions for the *exhibition area* requirements: $Spec_2$, $Spec_4$, $Spec_{5B}$, $Spec_{5D}$ and $Spec_{5A}$. We measure the running time, number of visited states and transitions to verify each assertion with increasing number of visitors allowed in *room D* (in Req. 2) and *exhibition area* (in Req. 4). For simplicity, we assume the number of visitors allowed in the *exhibition area* is twice the number of visitors in *room D*. The experiments were run on a personal computer having 8 cores running at 3.4 GHz with 8GB of DDR RAM and on a server that has 56 cores running at 2.2 GHz with 256GB of RAM. The results match and therefore we only show the results derived from the execution run on the server machine. For each assertion, we measure the running time (in seconds) using the Unix *time* command and from FDR output we infer the number of states and transitions visited.

In fig. 7.1, we summarize the results. Assertions that are localized to one component, i.e., $Spec_2$,$Spec_{5A}$, $Spec_{5B}$, $Spec_{5D}$ perform better than assertions spanning over the exhibition area, i.e., $Spec_4$. This affirms that topology-driven modelling is an effective compositional technique. In fig. 7.1 (top), we show that assertions over single component is approximately 100 times faster than assertions over the exhibition area shown in fig. 7.1 (bottom). This is because the processes are smaller in size and can be encoded with far fewer states and transitions, as shown in the states (middle) and transitions (right) graphs.

The verification of Req. 5 for *room A* is the least favourable to our technique. The verification of Req. 5 for *room A* by the composition of adaptation procedures exhibits the biggest increase in process size and running time between the two verification steps. This is because in the re-

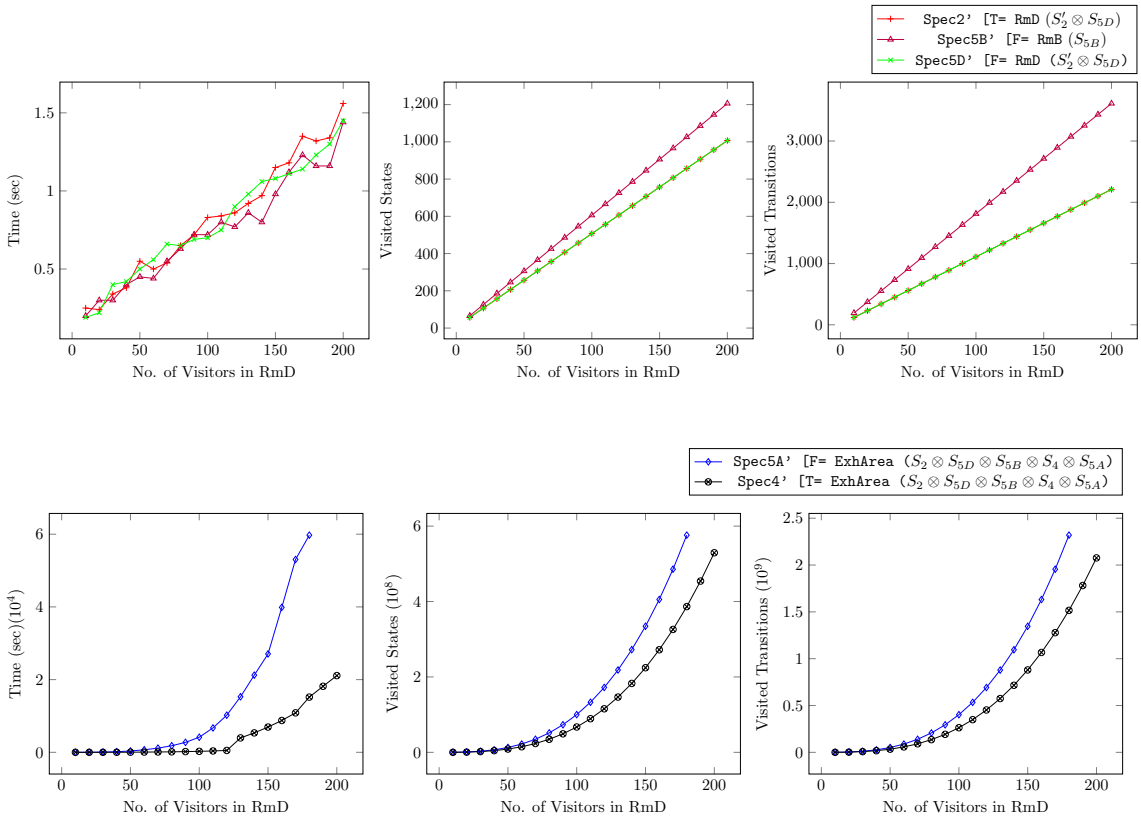


Figure 7.1: Experimental results for verifying exhibition area requirements when composing adaptation procedures. Here, we enable FDR minimization.

verification step the scope of verification had to be increased to include four other adaptation procedures and two other rooms

In fig. 7.2, we run the same experiment but disabling FDR minimization to investigate the extent FDR optimization improves the performance of our verification approach. Most of the transitions in our encodings are internal transitions encoding adaptation transitions. These transitions are removed by FDR through its normalization process. Comparing the graphs in figs. 7.1 and 7.2, one notices that the input size varies drastically, with minimization we are able to verify up to 200 visitors in *room D*, whereas without minimization we are only able to verify up to 80 visitors before the process runs out of memory. Consider, the assertion `Spec4 [T= ExhArea`, in which we allow 80 visitors in *room D* (and 160 visitors in the *exhibition area*), with minimization the verification takes 131.45 seconds and visits 34,658,711 states and 135,496,258 transitions, and with no FDR minimization, the verification takes 9,499.39 seconds and visits 364,019,512 states and 1,174,704,700 transitions. Because minimization reduces the size of processes, we exhibit better performance. This is a benefit of exploiting established, robust verification techniques that have years of experience in optimization.

We now compare the results in fig. 7.1 with the results derived in fig. 6.3c. The specifications $Spec_{5A}$ and $Spec_4$ scale much less in this verification step. In particular, the assertion `Spec5A [F= RmA` no longer scales (approximately) linearly. For 70 visitors, in the previous verification step the task is completed in 0.67 seconds after visiting 284 states and 1,413 transitions but now takes 3.08 seconds and FDR visits 20,873 states and 142,565 transitions. This is because we extend its scope in the verification as explained earlier. Partially overlapping adaptation procedures may impact the scalability of our verification approach negatively but topology-driven modelling and our theory of compositionality minimizes the extent to which adaptation procedures overlap. In

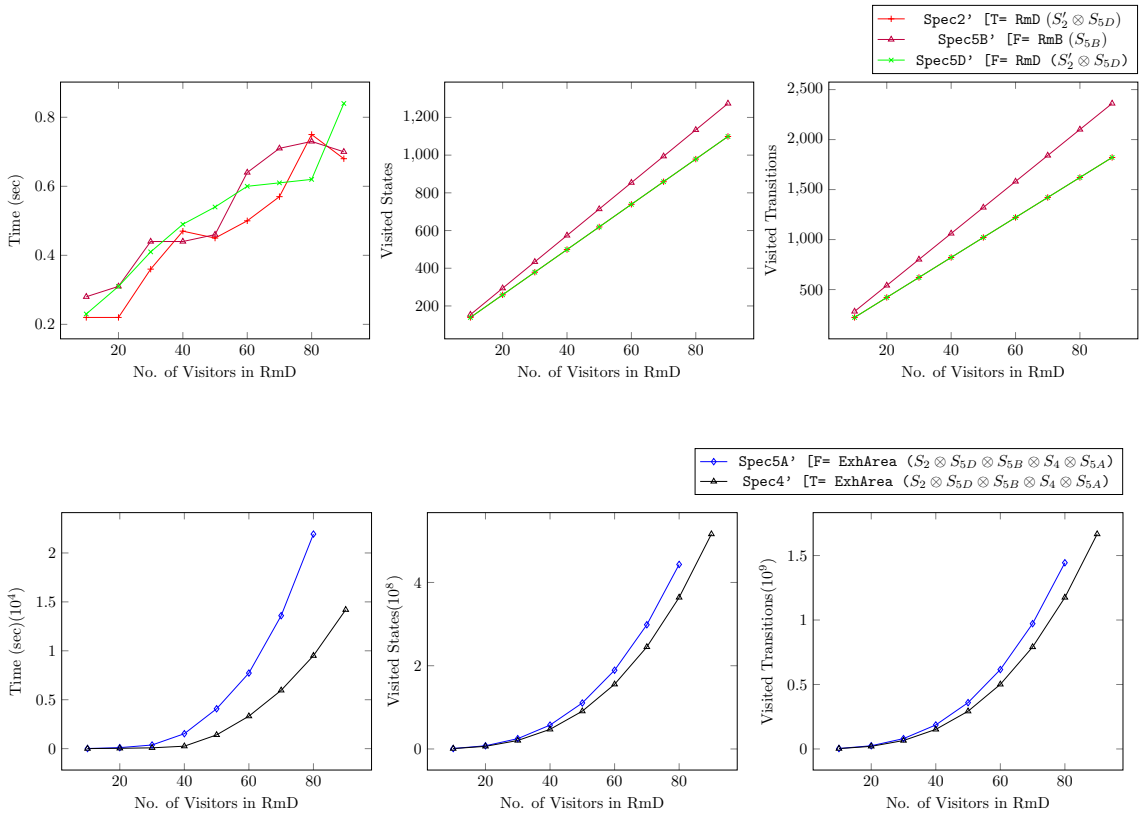


Figure 7.2: Experimental results for verifying requirements against the composition of adaptation procedures. Here, we disable FDR minimization.

our art gallery example, requirements are split into three non-overlapping areas - exhibition area, restoration area and access point, whose composition does not require verification efforts.

7.5 Summary

An SA CPS consists of a large and diverse collection of requirements. It is the responsibility of the system designer to present a model that satisfies *all* requirements. In this chapter, we discuss how the composition of adaptation procedures may affect requirements satisfaction and hence the need to verify satisfaction is preserved when all adaptation procedures and systems components are composed together. We identify overlap types where interference is not possible, e.g., when adaptation procedures scopes are disjoint or they only monitor common components, and for such cases, we develop on our theory of compositionality to prove that satisfaction in such cases follow from the compositionality theorem (theorem 6.1.18 in section 6.1.3). Requirements satisfaction still holds when adaptation procedures and previously ignored systems components are composed together. This minimizes the verification efforts required in Step 5 of our proposed methodology.

This chapter concludes our presentation of the proposed methodology to compositionally model and verify the satisfaction of security requirements in SA CPSs, motivated by an art gallery example. In our methodology, we propose a correspondence between requirements, specifications and adaptation procedures, which we called *requirement-driven adaptation*. We utilize the apparent *topology of CPSs and topological relations* to a systematic explore levels of granularity (i.e., different grouping of components) to define adaptation procedures that aims to ensure a requirement satisfaction. We also present an encoding for adaptation procedures that together with topology enables a system designer to experiment with different adaptation approaches to enforce a require-

ment. We discuss how overlaps between adaptation procedure scopes may potentially affect the satisfaction of requirements and that a system designer needs to reverify requirements when adaptation procedures are composed. For our methodology, we present a *theory of compositionality* that reduces the verification efforts required to enable tractable verification of SA CPSs. We also highlight how for our encoding of CPSs we *leverage existing, robust verification tools*. Here, we use FDR [59], a refinement-checker for CSP.

Chapter 8

A Translation Tool from ACSP$_M$ to CSP$_M$

We now overview an implementation of a tool that automates the translation to CSP at the core of our verification approach. Through this tool we aim to improve the usability of our framework by alleviating the task of encoding realistic SA CPSs. In line with the evolution of CSP, we first define a concrete syntax for our process language, which we call $ACSP_M$. $ACSP_M$ is a machine readable dialect of ACSP, where shorthand constructs and idioms inspired from functional programming languages, like Haskell and Miranda, have been added to expedite the task of encoding realistic problems [111]. The translation process is depicted in fig. 8.1, where for an $ACSP_M$ input file, we first construct the *in* and *out* functions required by the well-formed proof rules. With these functions, we then define a checker for $ACSP_M$ to verify if the input is well-formed, as defined in Def. 6.1.3. For well-formed inputs, we provide a translation to CSP$_M$, realizing the rules in fig. 6.2. The translation produces a CSP$_M$ file that is both saved on the user machine and launched automatically into FDR. The tool is developed using the parser generator Antlr [101].

In this chapter, we first overview the concrete syntax $ACSP_M$ and then discuss each main step in the translation process.

8.1 The Concrete Syntax for ACSP$_M$

We extend ACSP with a subset of the CSP$_M$ constructs. In our implementation, we started from the CSP grammar found in [49] and appended both ACSP constructs and a subset of the CSP$_M$ constructs. The full listing of the $ACSP_M$ grammar can be found online[1]. Here, we discuss the main constructs inspired from CSP$_M$ that have been included in $ACSP_M$.

Definitions

At the most basic level, FDR runs a series of assertions that checks if a process refines a specification or if a process is deterministic, deadlock-free or livelock-free for trace, failure and failure-divergence semantic models. An input file of $ACSP_M$ contains a sequence of process definitions and assertions.

We allow the input file to reference other files through the `include` command. This command acts as a macro with the source code of the reference file being copied into the main file.

[1]Code for the translator is available at `https://github.com/AimeeBorda/ACSP-Compiler`. The repository also contains the full encoding in $ACSP_M$ for the Art Gallery example and the second case-study to be presented in the next chapter, a smart stadium.

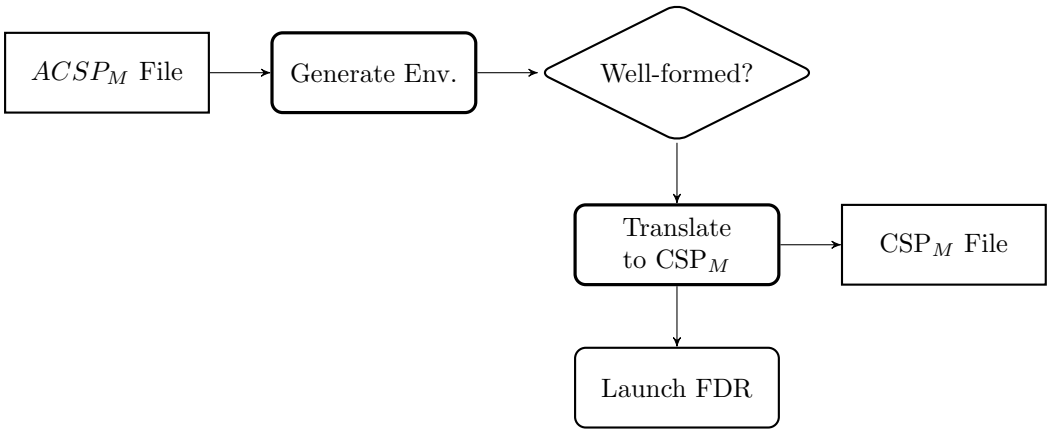


Figure 8.1: The translation process from $ACSP_M$ to CSP_M

Events Encoding

A notable difference between CSP and CSP_M that has been incorporated in $ACSP_M$, is the rich syntax for encoding events. In CSP_M, events are defined as channels with types. Recall the *goto* events from the art gallery example. A typical *goto* communicates not only that an agent moved from one room to another but also states the type of the agent, the source and destination of the movement, e.g., $vis_{(EA,EB)}$ broadcasts that a visitor moved from *room A* to *room B*. In $ACSP_M$, we define the data-types for agent types and rooms,

```
datatype Room = EA | EB | AP | ...
datatype Agent = Vis | Emp | Grd
```

A *goto* event comprises an agent `a : Agent` and rooms `f, t : Room`. We then define a channel `goto` such that

```
channel goto : Type.Room.Room
```

We encode the event $vis_{(EA,EB)}$ as `goto.Vis.EA.EB`.

Expressions

CSP_M also supports a rich syntax for set generation and boolean and numerical expressions. We incorporate this rich syntax in $ACSP_M$. In $ACSP_M$, we support enumerated sets, e.g., {|`goto`|} and {|`goto.x | x <- {Grd,Emp}`|} which returns all possible *goto* events and all employee movements respectively. In $ACSP_M$, was also assume a standard encoding of natural numbers and boolean expressions.

Processes

The goal of CSP is to verify concurrent processes. In CSP_M and $ACSP_M$, a richer syntax for process definition is introduced. We include in our tool a subset of the process definitions found in CSP_M. In comparison with ACSP, processes in $ACSP_M$ also include the interrupt, timeout, sequential, internal choice and boolean guard. An addition that we already assumed throughout this dissertation is that of named processes $P(\vec{x})$ as a shorthand notation for recursive definitions. This helps us decompose and better understand the encoding of complex processes.

The prefix process, $e \rightarrow P$, supports pattern matching. Consider, the process,

```
goto?_:  Agent $f:  {x   |  x <- Room, x != EA}?t:  diff(Room,EA) -> C(f,t)
```

Here, we pattern match the three parameters in a *goto* event. In this example, the continuation is independent of the type of the agents—underscore implies that the parameter is not bound to a variable. For the second and third parameters, the parameters are bound to `f` and `t` respectively and for both `f` and `t` we accept any room other than `EA`.

This is equivalent to composing the individual events with the (external) choice construct.

Higher-order Communication

In ACSP, we extend CSP with higher-order communication to concisely express adaptation. In the process `Example` below, we scope the location `l` around the process that composes in parallel a named process `l[A0]` with a process that performs a higher-order output on `l`, `l!<SKIP>` and evolves to process `P`. The two processes synchronize over the set of first-order events `Ev`,

```
Example = (new l)(l[A0] [| Ev |] l!<SKIP>.P)
```

Excluded CSP$_M$ Constructs

The implementation is a proof of concept for the translation and thus we did not incorporate all the constructs from CSP$_M$. In particular, we omit advanced hybrid constructs and replicated process definitions. CSP$_M$ also provides a richer syntax for data definition, which includes tuples, maps and lambda expressions. These have not been included in $ACSP_M$.

We now look at the three passes we perform on the input to generate the CSP$_M$ output: *environment generation*, *well-formedness checking* and *translation*.

8.2 Environment Generation

The rules for checking if a process is well-formed, depicted in fig. 6.1, depend on the definition of the *in* and *out* functions. These functions, given a process P, to return the free locations in named processes in P and the free locations in higher-order prefixes found in P respectively. Consider the example,

```
Example = (new l)(PA [| Ev |] PB)
PA = l[A0]
PB = PA ||| l!<SKIP>.A1
assert Example :[deterministic]
```

where we assume the definition of processes `A0` and `A1`. Here, inferring the *in* and *out* result for the `Example` process depends on the results from the processes `PA` and `PB`, which a parser, parsing from top-to-bottom, have not encountered yet. As a workaround to this problem, we construct a map *map* that for all process definitions P returns a tuple containing: the set of bound locations inside P, the set of locations in named processes inside P, the set of locations in higher-order prefixes inside P and all referenced processes inside P. Subsequent steps in the translation process infer the result of the *in* and *out* functions by the union of the recursive calls results of the referenced processes. For instance, for the input above, the *map* is

	Example	PA	PB
inExp	$\emptyset$	{l}	$\emptyset$
outExp	$\emptyset$	$\emptyset$	{l}
calls	{PA, PB}	$\emptyset$	{PA}
bound	{l}	$\emptyset$	$\emptyset$

In the next step, the check for well-formedness, we infer the results of $in(\texttt{Example})$ and $out(\texttt{Example})$ by the union of the recursive calls results from the *calls* processes removing bound locations.

$$\begin{aligned} in(\texttt{Example}) &= inExp(\texttt{PA,PB,Example}) - bound(\texttt{PA,PB,Example}) \\ &= \{l\} - \{l\} \end{aligned}$$

The implementation employs memoization techniques to handle circular process calls, e.g., the tool is able to check well-formedness of the processes below

```
J = (new l)(l[SKIP] ||| K)
K = (new h)(h[SKIP] ||| J)
```

8.3 Well-formedness Checking

In fig. 6.1, we define a set of rules to identify the class of well-formed processes. A process P is well-formed iff $\emptyset \vdash P$. Here, the input is a sequence of named processes and assertions. Consider the input,

```
Example = (new l)(PA [| Ev |] PB)
PA = l[A0]
PB = PA ||| l!<SKIP>.A1

assert Example :[deterministic]
```

Checking that all named processes at the top level of the file(s) are well-formed is too restricting. In this example, the processes `PA` and `PB` are named processes at the top level that are not well-formed, but we only intend to use the processes as part of another process, the `Example` process. We know this because there are no assertions about either `PB` or `PA` directly. For this reason, we instead check that all processes in assertions are well-formed.

For the process `Example` to be well-formed, the side-condition of wPar, in fig. 6.1, needs to be satisfied. The side-conditions require that `PA` and `PB` are well-formed with respect to the environment $\{l\}$. For every bound location, one of the sub-processes has the named process and the other the higher-order output. These side-conditions are determined by the `PA` and `PB` processes, which we have not encountered yet in our parsing. Recall in the previous pass over the code, we generate a *map* that implements the *in* and *out* for all named processes in the input. When parsing `Example`, we reference the *map* to check the satisfaction of the side-conditions of wPar. The example above is ruled out, because $in(\texttt{PA}) = \{l\}$ and $in(\texttt{PB}) = \{l\}$. Consider, a different encoding of the process `PB`,

```
PB = l!<SKIP>.SKIP ||| PB
```

In this example, the process `Example` is well-formed, because $in(\texttt{PB}) = \emptyset$. This satisfies the side-condition in wPar, even though we have a recursive call that performs higher-order outputs on `l`. All higher-order outputs on `l` are on the right-hand side of the interleaving in `Example` and the named location is on the left-hand side of the interleaving.

The checker also implements the rules wLoc and wSnd. The rules require the process `P` at a named location, e.g., `l[P]`, or in a higher-order output, e.g., `l!<P>`, to be well-formed with respect to the empty environment. The process `P` should not contain any free named locations or higher-order outputs. Consider the example,

```
G = (new l)(l!<H>.SKIP ||| l[P])
H = SKIP ||| I
I = l!<SKIP>.SKIP

assert G:[deterministic]
```

This example is ruled out because the process `H` has an higher-order output on `l` nested in `I`. Note also that the process `P` is undefined. For the purpose of this check, we assume that undefined identifiers are well-formed processes, because the identifier can be a reserved word or method in CSP$_M$, e.g., `normal`. The last example to highlight the functionality of the checker is

```
(new l,m,n)(l[P] ||| m!<R>.SKIP)
```

The above process is well-formed even though `n` is not used in the process, `m` does not have a named location and `l` is never adapted.

Limitations This tool is a proof-of-concept and thus error handling leaves much to be desired. Even though, the tools outputs to the console which construct or assertion violates the well-formed check, the tool does not indicate from which line the error originated. For this reason, we delay as many checks as possible to FDR, to exploit its rich error handling mechanisms.

We also require processes in parallel composition to be enclosed within brackets. Consider

```
Example = normal(l[P] ||| l!<R>.Q)
```

Here, the associativity of the interleaving is non-ambiguous. However, we still require the composition to be enclosed within another set of brackets. The `new` keyword, however, can be omitted if no locations are bound in the interleaving process.

8.4 Translation

Once, we verify that the input is well-formed, the tool produces a CSP$_M$ file containing the translated encoding. The translation follows the rules in fig. 6.2. Here, we must construct the injective map m from higher-order prefix to distinguished CSP events. Recall the process above

```
Example = (new r, l)(PA [| Ev |] PB)
PA = l[A0] ||| r[A0]
PB = l!<SKIP>.P ||| r!<SKIP>.SKIP

assert Example :[deterministic]
```

The first process `Example` contains the composition of `PA` and `PB`. From the rule tPar in fig. 6.2, the two processes synchronize on all encoded higher-order communications $A = \{m(l!R) \mid R \in Proc, l \in L\}$. This information is not known until the end of the translation phase, when we have parsed all possible higher-order prefixes. We utilize the CSP$_M$ enumerated set construct to avoid having to explicitly list all the identifiers communicated on a location l. For each location $l \in L$, we assume a `channel l` that will be defined at the end of the translation after all prefixes over location l have been identified. For the `Example` process, we only need to state that the processes `PA` and `PB` synchronize and hide the set of events $\{|$`r, l`$|\}$. We optimize the evaluation of the input (as recommended on the FDR website[59]) by encapsulating the hide process under a `normal` function. The process `Example` is translated to

```
normal((PA [| union(Ev, {|r,l|}) |] PB) \ {|r,l|})
```

We now translate the next process, `PA = l[A0] ||| r[A0]`. From tLoc in fig. 6.2, the processes `l[A0]` and `r[A0]` translate to $A0 \triangle rec(l)$ and $A0 \triangle rec(r)$ respectively where

$$rec(l) = \Box_{e \in ch(l)} \; e \rightarrow (T_e \triangle rec(l)) \text{ such that } p(e) \rhd T_e$$
$$rec(r) = \Box_{e \in ch(r)} \; e \rightarrow (T_e \triangle rec(r)) \text{ such that } p(e) \rhd T_e$$

Similarly, we cannot infer all the distinguished CSP events in $ch(l)$ and $ch(r)$ until the end of the translation. Once again, we utilize CSP$_M$ rich syntax for process definition to avoid having to explicit list all possible values of $ch(l)$ and $ch(r)$. We translate the processes $rec(l)$ and $rec(r)$ to

```
rec(l)   =   let R = l?id -> map(l,id) /\ R
rec(r)   =   let R = r?id -> map(r,id) /\ R
```

The process `map` will be defined at the end of the translation, which given a location and identifier returns the process communicated $p(l.0)$. We translate the processes `l[A0]` and `r[A0]` to

```
let R = l?id -> map(l,id) /\ R within A0 /\ R
let R = r?id -> map(r,id) /\ R within A0 /\ R
```

The process `PA` is translated using tPar to

```
normal((let R = l?id -> map(l,id) /\ R within A0 /\ R ||| let R = r?id -> map(r,id) /\ R
    within A0 /\ R)
```

We now translate the process `PB`. The processes `l!`$\langle SKIP \rangle$`.P` and `r!`$\langle SKIP \rangle$`.SKIP` translate to

```
l.0 -> P
r.0 -> SKIP
```

The events `l.0` and `r.0` are the distinguished CSP events encoding the higher-order outputs $l!SKIP$ and $r!SKIP$ respectively. We track all higher-order outputs in an order-preserving set m. The identifier communicated on l is the index of the process in the map. Using a set as our data-structure allow us to reuse identifier when the same processes are communicated multiple times. In our example, the map is ,

$$m = l \rightarrow [SKIP_{\text{CSP}_M}]$$
$$\quad\; r \rightarrow [SKIP_{\text{CSP}_M}]$$

At the end of the translation, the map m contains all higher-order prefixes and their distinguished CSP$_M$ event. We use this, to define the `map` process assumed in the translation of a named process $l\langle P \rangle$ which given a channel and identifier returns the communicated process. For our example input,

```
map(ch, id) = if ch == l and id == 0 then SKIP
              if ch == r and id == 0 then SKIP
              else SKIP
```

The last task of the translation is to define the channels that in the original encoding were locations. Here we only have the location l. From the mapping, we infer the range that will be communicated on l from the size of the set $m(l)$. This allows to define a refined data-type for the locations,

```
channel l : {0..0}
channel r : {0..0}
```

Input	Output	Valid

`Example = (new l)(PA [| Ev |] PB)`
`PA = l[A0]`
`PB = PA ||| l!<SKIP>.A1`
`assert Example :[deterministic]`
	`l are in left in and right in`	✓

`Test = (new l,m,n)(l[P] ||| m!<R>.SKIP)`
 	
`assert Test :[deterministic]`
	`transparent normal`
	
	`Test = normal((let R = l?id -> (map(l,`
	`id) /\ R)`
	`within (P/\ R)[|{| l,m,n|}|] m!0 ->`
	`SKIP) \ {| l,m,n|})`
	
	`assert Test :[deterministic]`
	
	`channel l : {0..0}`
	`channel m : {0..0}`
	`channel n : {0..0}`
	
	`map = \ chName,id @ if chName == m and`
	`id == 0 then R`
	`else SKIP`
		✓

`Test = (new l)(l[A] [| E |] m!<SKIP>.P)`
 	
`assert Test [T= SKIP`
	`error in assertion assert`
	`Example[T=SKIP is not well-formed`
		✓

Table 8.1: Some Validation Tests performed

Limitations

The `Let` construct may introduce scoped variables. Consider the example,

```
let x = 5 within l!<P(x)>.SKIP
```

The higher-order output `l!<P(x)>` is replaced by a distinguished event `l.0` and we define a global process `map` that given `l.0` returns `P(x)`. However, because we removed the process `P(x)` from the `let` context, the variable `x` becomes undefined in the new scope inside the `map` process, resulting in a syntax error. In this case, the translation still produces the CSP$_M$ file and still launches FDR but FDR will state that there is a syntax error because `x is undefined`.

8.5 Tool Validation

We validate the correctness of the tool in two steps: first we validate that the check for well-formedness is correct and then we validate the correctness of the translation.

We validate the well-formedness check using a series of unit tests that target different key points of the well-formedness check.

We validate the correctness of our translation through a serious of simple examples. We show

three such examples in table 8.1. We also manually verified the translation of the two case-studies presented in this thesis that of a smart art gallery and a smart stadium to be presented in the next chapter.

8.6 Summary

In this chapter, we overviewed the implementation of a translation tool from $ACSP_M$ to CSP_M. $ACSP_M$ is the concrete representation of ACSP. This extension is in line with CSP where CSP_M has been introduced to promote the use of CSP for verifying realistic problems. Here, we outline the main extensions inspired from CSP_M that we included in $ACSP_M$. We discussed the well-formed checking algorithm which realizes the proof rules in fig. 6.1. For well-formed processes, we define a translation to CSP_M following the proof rules in fig. 6.2. The outcome of a translation is a CSP_M file that is launched into FDR. In the next chapter, we evaluate the applicability of our methodology through a second case-study. There, we also use this tool to implement the check for well-formedness and translation to CSP_M for the second case-study.

Chapter 9

Case Study: A Smart Stadium

In this chapter, we evaluate our modelling and verification methodology for SA CPSs through a second case-study, that of a smart stadium inspired from [92]. We outline the goals of the evaluation and then overview the case-study and its requirements. For the evaluation, we follow our proposed methodology, presented in section 4.1, to model and verify the satisfaction of all requirements.

9.1 Evaluation Criteria

The goal of this evaluation is to investigate the applicability of our verification approach for realistic systems. In particular, we look to investigate the following criteria

Applicability of requirement-driven adaptation We address this criterion by modelling an alternative case-study, inspired from SA CPSs. We investigate how our modelling methodology for SA CPSs tackles the complexity of attaining a compositional model.

Applicability of topological knowledge and relations to systematically explore adaptation procedures. Topology guides the system designer to not only identify the main cyber and physical components of CPSs but also to systematically explore different grouping of components, that adaptation procedures control to optimally and correctly ensure the satisfaction of a requirement. We investigate how the topology and topological relations alleviate the task of identifying the components that affect the satisfaction of requirements.

Scalability of our verification technique. In our framework, we utilize FDR, an existing refinement-checker for CSP_M. By default, we have a GUI, that improves the usability of our verification technique, and enhanced performance, stemming from FDR parallel and distributed refinement-checking. We also investigate how the concrete syntax of our process language and the automated translation further improves the usability of our framework.

9.2 A Smart Stadium

The case study is inspired from a real-world IOT project on Croke Park in Dublin presented in [92]. We model a smart stadium that can host up to 86,000 visitors. The stadium is split into three areas: VIP, Upper area (U) and lower area (L). Each area is further split into sections numbered from $0 \ldots 35$. We write L_0 to be section 0 in the lower area. Each section has a capacity of 800 visitors. As shown in fig. 9.1, sections are connected to a common corridor that leads to an exit

Figure 9.1: The seating plan of Croke Park from [1]

and adjacent sections e.g., the section L_0 is connected to the *corridor*, section L_1 and section L_{35}. Ticket holders have access to a single section e.g., a ticket holder L_0 has access to section L_0.

For the purpose of this evaluation, we consider the following requirements.

Requirement A. Before a match, only ticket holders and employees have access to a section. ◊

Adaptation Procedure A.1. We satisfy this requirement by implementing a simple access control that before a match starts ensures only ticket holders and employee access each section. ■

Requirement B. After a match, visitors should be directed to the nearest exit to avoid congestion. ◊

Adaptation Procedure B.1. Exits can be reached from the corridor. We implement an access control that allows visitors to move from a section to the corridor and precludes visitors from moving to non-empty adjacent sections, as it increases congestion. If however an adjacent section is empty, then visitors are allowed to exit through the empty section. ■

Requirement C. In the case of a fire alarm within a section, the section should be evacuated. ◊

There is a fire-alarm installed in each section.

Adaptation Procedure C.1. During an evacuation, no visitor should be able to enter the section. Visitors are also allowed to exit the section by moving to the corridor leading to an exit or to an adjacent section, provided it is not evacuating as well. ■

The evacuation of the stadium is modelled by triggering the fire alarm in all the sections.

During a match, the following requirements also apply.

Requirement D. To have a non-intrusive access control, visitors are allowed to roam to other non-empty sections. ◊

Visitors attending a match would likely be seated before the match starts and thus empty seats are free to be taken by other visitors aiming to get a better view of the stadium. Empty sections in the lower and upper areas should however be closed to reduce energy usage across the stadium.

Adaptation Procedure D.1. To satisfy the requirement, we define an adaptation procedure that changes the access control to allow visitors to enter if the section is not empty, otherwise access control is limited to ticket holders. ■

Requirement E. On a windy day, the system should attempt to empty the upper area. ◊

Adaptation Procedure E.1. The system should immediately change the access control such that only ticket holders and employees can enter exposed sections. The system should also attempt to open an *empty* lower section, so visitors (in exposed sections) can find another more sheltered seat. If an alternative section (in the lower area) is identified, the exposed section is closed to visitors and visitors are directed towards the alternative section. However, as is often the case in a sold-out match, if there is not an empty lower section, only ticket holders can enter the upper area for the remainder of the match. ■

Requirement F. During a match, the system should aim to keep noise levels below a threshold. ◊

The threshold is determined according to the length of exposure and the time of the day. The threshold is lower after 10pm and during the resting period between 2pm and 3pm. Noise level in a section is measured through a noise-detector. The stadium has a noise-detector installed in each section. Noise may be reduced by allowing visitors to distribute themselves more evenly around the stadium. This only applies during a match because before and after a match, crowd control takes precedence over noise control.

Adaptation Procedure F.1. When noise levels are above the threshold, the system should try to gradually open empty sections and preclude non-ticket holders from entering the noisy section. ■

Requirement G. A section may be re-opened as a backup section if it is not noisy, not exposed to strong wind, empty and its fire alarm is off. ◊

Adaptation Procedure G.1. To satisfy this requirement, a section should only allow visitors to enter if another section is either too noisy, exposed to strong winds, not empty or the fire-alarm is on and none of these conditions are true for the section be opened. ■

Requirement H. To minimize energy usage, noise level sensors are switched off before and after match or if the section they are in is empty. ◊

Adaptation Procedure H.1. To satisfy this requirement, we switch off sensors before and after a match or if a section is empty during a match. Sensors should be switched on when during a match, visitors enter an empty section, i.e., the section is not longer empty. ■

Requirement I. Floodlights should be switched off during the day and if a section is empty. ◊

Each section has a floodlight that can be controlled remotely.

Adaptation Procedure I.1. To satisfy this requirement, we switch off all floodlights during the day when there is sunlight and a section's floodlight if the section remains empty by the start of a match. Unlike the noise sensor, if a section becomes temporarily empty during a match,which may happen if a section is scarcely filled, the lights should not be switched off as this may affect user experience. ■

9.2.1 Challenges

The stadium comprises 108 homogeneous sections. This regular structure of the stadium potentially allows us to replicate verification results achieved for a single section to the whole stadium. This may not necessarily be obvious because requirements may be defined at different levels of granularity. For instance, Req. E distinguishes between the upper area and lower area, Req. B

may include adjacent sections to fully capture congestion in a section and Requirements D to G may request alternative sections to be opened to visitors, which may require us to define a global adaptation procedure. These may affect how compositional our model is and how much we can reuse verification efforts. In the worst-case scenario the verification of some requirements may encompass the whole stadium.

Our approach entails having an adaptation procedure for each requirement. In a single section, we have seven requirements that together define the access control to a section. This dense level of overlap between adaptation procedures may highlight a potential limitation of our proposed requirement-driven adaptation. Moreover, the composition of seven adaptation procedures may drastically increase the potential of overlaps of adaptation procedures that require reverification. This is amplified if some components are globally defined, e.g., the *wind monitor* may be shared between all sections in the stadium.

Moreover, each section hosts up to 800 visitors. If all adaptation procedures need to track the number of visitors. The size of processes (number of states and transitions) is going to increase drastically which may lead to the verification approach becoming computationally infeasible.

Finally, in this example, we introduce a broader range of sensors and components, e.g., wind monitor, noise detectors and the notion of time. The sensors need to be encoded in our process language ACSP. Unlike the art gallery example, most components are cyber components, whose topology and topological relations are not obvious.

9.3 Step 1: Modelling the CPS

The stadium comprises 108 sections—36 upper sections (U), 36 lower sections (L) and 36 VIP sections (VIP), numbered from 0 to 35.

Each section has a physical connectivity with the corridor and adjacent sections, represented by the set of first-order events $\{\!|goto|\!\}$. A *goto* event has three parameters: the agent and the section from where the agent is leaving and to where the agent is going. An agent can be either an employee *emp* or ticket holder represented by the section identifier to which the ticket is valid. A section ranges over the corridor $Corr$ or a section identifier. The event $goto_{L_0,U_0,Corr}$ represents a visitor that has a ticket to section L_0 moving from section U_0 to the corridor. The component also broadcasts an *empty* event to other components in the same section once a section is empty. A section is also digitally connected to all other sections in the stadium as it may requests another section to be opened as backup and vice versa other sections may request the section to be reopened as backup. We define a family of events $open_{(s,t)}$ to represent a section s requests section t to be opened. We let $E_a = \{\!|goto, open, empty|\!\}$ represent the set of first-order events from the *access controller* component .

The stadium has installed in each section the cyber components: a *noise detector, fire alarm* and *floodlights controller*. Each component has a well-defined interface. The *noise detector, fire alarm and floodlights controller* broadcasts the events $noise_b$, $alarm_b$, $lights_b$ where b can be $T|F$ respectively. We let E_n, E_f and E_l to be the set of first-order events in the components respectively. [1]

Globally, we assume the cyber components *match status* and a *wind monitor*. The *match status* component broadcasts the set of first-order events $E_m = \{before, during, after\}$ to represent the status of the match: just before a match starts, when a match starts and once a match finished respectively. The *wind monitor* broadcasts a *wind* event when strong wind is detected. We refer

[1] For presentation reasons, we abbreviate *True* and *False* to T and F respectively

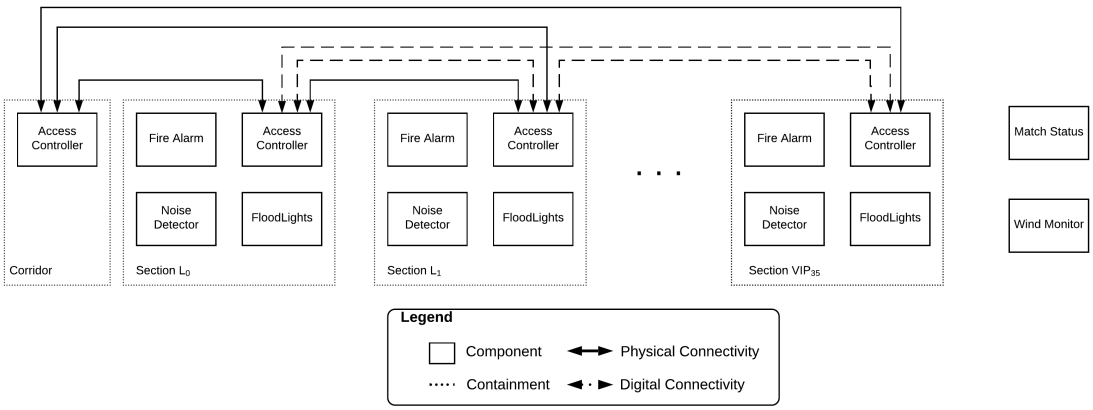


Figure 9.2: The component model for the stadium case-study. The model comprises 108 sections, a wind monitor and match status components and a corridor.

to this singleton set of events of the component as E_w. We depict the component model for the stadium in fig. 9.2.

9.4 Step 2-4: Encoding a Section with Adaptive CSP

In this section, we overview the next 3 steps of the methodology, where *for each requirement* we explore the space of the components to identify a grouping of components over which we encode an adaptation procedure that aims to ensure the satisfaction of the requirement. We verify using FDR that this encoding, which we call the *unary cluster*, refines, according to a semantic model, a specification process that describes the requirement. In this section, we show the encoding for the first three requirements, the encoding for the remaining requirements can be found in appendix D.[2]

In our encoding, we assume *Visitors* and *SectionID* to be the set of all ticket holders for the stadium and the set of sections identifiers respectively. We range over v for the set of ticket holders and s, s' for sections. For a section s, we also assume the set of adjacent sections is represented by adj, e.g., for a section L_0, the set $adj = \{Corr, L_{35}, L_1\}$. We range over a for adjacent sections. For simplicity, we omit the identifiers, i.e., subscripts from first-order events, when the event is an external choice over all identifiers, e.g., we write $goto \rightarrow P$ to mean

$$\Box_{\substack{v \in Visitors,\, Emp \\ a \in adj}} \begin{cases} goto_{v,s,a} \rightarrow P \\ goto_{v,a,s} \rightarrow P \end{cases}$$

Moreover, employees are free to roam any section and thus for simplicity we omit employee movements in our encoding. In the subsequent encodings, the section being modelled and verified is represented by the place holder s.

Requirement A: ***Before a match, only ticket holders have access to a section*** We define an adaptation procedure Π_A to guarantee the satisfaction of this requirement for section s.

For satisfying this requirement, we adapt the *access controller* to preclude non-ticket holders from entering s and monitor the *match status* controller to identify when a match is about to start. The *access controller* can be in one of two functionalities

[2]The full listing of the encoding in $ACSP_M$for this case-study and the Art Gallery case-study can be found in `https://github.com/AimeeBorda/ACSP-Compiler/tree/master/Examples`

1. If a match is not taking place, visitors are not allowed to enter any section. This is the default (initial) behaviour of the *access controller*. Note how we do not include the event $goto_{v,a,s}$ that encodes a visitors entrance into s, either through the corridor or an adjacent section,

$$SExit = \Box_{v \in Visitors} \left\{ goto_{v,s,Corr} \rightarrow SExit \right.$$

2. Only ticket holders can enter section s. Note how we only accept $goto_{s,\_,\_}$,

$$SOpen = \Box_{a \in adj} \begin{cases} goto_{s,a,s} & \rightarrow SOpen \\ goto_{s,s,a} & \rightarrow SOpen \end{cases}$$

The internal state of a section keeps track of the visitor movements to make sure that the number of visitors leaving a section is less than or equal to the number of visitors entering the section. We define a process Beh that monitors the movement of visitors. The process broadcasts an *empty* event once s becomes empty. This relieve subsequent adaptation patterns from having to track the movement of visitors to determine when a section is empty.

$$Beh(n) = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases} n > 0 \ \& \ goto_{v,s,a} & \rightarrow Beh(n-1) \\ \qquad\quad goto_{v,a,s} & \rightarrow Beh(n+1) \\ n = 0 \ \& \quad empty & \rightarrow Beh(n) \end{cases}$$

For an adaptation to take effect, we encapsulate the *access controller* component in the named location *ac*. An adaptation procedure adapts the behaviour of the *access controller* by performing a higher-order output to *ac*. The (adaptable) *access controller* for a section s is encoded in the ACSP process,

$$AccessController = ac\langle SExit \rangle \underset{\{|goto|\}}{\parallel} Beh(0)$$

We now define an adaptation procedure Π_A as the composition of an adaptation pattern P_A determining *when* to adapt and the encoding of an adaptation function F_A determining *how* to adapt. To satisfy this requirement, we adapt on the *before* event that marks the start of a match and arrival of visitors.

$$P_A = \Box \begin{cases} before & \rightarrow \star \rightarrow ack \rightarrow P_A \\ during & \rightarrow P_A \\ after & \rightarrow P_A \\ goto & \rightarrow P_A \\ open & \rightarrow P_A \\ empty & \rightarrow P_A \end{cases}$$

Adaptation pattern comprises solely first-order events. We can check using FDR that the adaptation pattern P_A only monitors the behaviour of the *access controller* and *match status* components and does not affect the execution of the components other than through adaptation. We verify that the adaptation pattern does not, unless adaptation is in progress, refuses any events from the

components eq. (9.1) or introduces events not in the interface of the components eq. (9.2),

$$RUN(E_{ac}, E_m) \quad \sqsubseteq_{F(ACSP)} \quad P_A \setminus \{|\star, ack|\} \tag{9.1}$$

$$P_A \setminus \{|\star, ack|\} \quad \sqsubseteq_{T(ACSP)} \quad RUN(E_{ac}, E_m) \tag{9.2}$$

On an $\star$-event, the adaptation function, encoded in the process F_A below, adapts the behaviour at location ac to the process $SOpen$,

$$F_A = \star \rightarrow ac!SOpen.ack \rightarrow F_A$$

The adaptation procedure Π_A is defined as the composition of the processes F_A and P_A synchronizing on the first-order $\star$- and ack events,

$$\Pi_A = (\nu \star, ack) \left(F_A \underset{\{\star, ack\}}{\parallel} P_A \right)$$

The adaptation procedure also monitors the *match status* component. This component is not adaptable and is encoded as,

$$status = before \rightarrow during \rightarrow after \rightarrow status$$

We encapsulate the process *status* in location ms,

$$MatchStatus = ms\langle status \rangle$$

The *unary cluster* for a section s is encoding as,

$$S_A = (\nu\, ac, ms) \left((AccessController \,|||\, MatchStatus) \underset{E_{ac}, E_m}{\parallel} \Pi_A \right)$$

We now discuss the specification for Req. A. On the *before* event, the behaviour of s is described by the process *BefEvt* that precludes non-ticket holders from entering s or the section from being used as a backup (note we preclude the *open* events). During or after a match, the behaviour is described by process R which encodes the most permissive behaviour, i.e., accepts all events from the components in the unary cluster.

$$Spec_A = \text{let}$$

$$R = \Box \begin{cases} goto & \rightarrow R \\ before & \rightarrow BefEvt \\ during & \rightarrow R \\ after & \rightarrow R \\ open & \rightarrow R \\ empty & \rightarrow R \end{cases} \qquad BefEvt = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases} goto_{s,a,s} & \rightarrow BefEvt \\ goto_{v,s,a} & \rightarrow BefEvt \\ during & \rightarrow R \\ after & \rightarrow R \\ before & \rightarrow BefEvt \\ empty & \rightarrow BefEvt \end{cases}$$

$$\text{within } R$$

We verify using FDR that for a section s, the cluster S_A trace refines the specification $Spec_A$,

$$Spec_A \sqsubseteq_{T(ACSP)} S_A$$

Requirement B: ***After a match, visitors should be directed to the closest exit to avoid congestion*** We satisfy this requirement by adapting the *access controller* to preclude visitors from entering a section that is being cleared after a match and monitoring the *match status* component to determine when a match has ended. To reduce congestion, the *access controller* may also request *empty* adjacent sections to be reopened so visitors can also exit through them. If the adjacent section is not empty, visitors are precluded from moving between sections as this tends to increase congestion.

The *access controller* can be in one of two functionalities

1. Visitors can only move from a section to the corridor and the controller also polls adjacent sections to see if they are empty, through the events $open_{s,a}$. The process $SExit$ has been defined in Req. A,

$$AC_1 = SExit \;|||\; \Box_{a \in adj \setminus corr} \Big\{ open_{s,a} \rightarrow SKIP$$

2. Once s is empty, visitors from (non-empty) adjacent sections can exit through it to further reduce congestion if needed. The process broadcasts the event $open_{a,s}$ to signal to adjacent sections that it is empty and can thus be used as a backup

$$AC_2 = SExit \;|||\; \Box_{a \in adj \setminus corr} \Big\{ open_{a,s} \rightarrow SKIP$$

We utilize the process *Beh* from Req. A to track the number of visitors entering and leaving s to ensure that the number of visitors leaving is less than or equal to the number of visitors that entered a section. The initial process for the *access controller* system is the ACSP process *AccessController* defined in the previous requirement, where an adaptation procedure can communicate through higher-order outputs to *ac* new behaviour to the component.

We now define an adaptation procedure Π_B that guarantees the satisfaction of Req. B for a section s. The adaptation procedure comprises an adaptation pattern P_B and the encoding for an adaptation function F_B that determines when and how to adapt respectively.

The adaptation pattern monitors the *access controller* and *match status* components. We trigger adaptation on two events: on the *after* event and when the section is empty after a match. We communicate a boolean variable with the $\star$-event to specify if the section is empty.

$$\begin{aligned} P_B = {} & \text{let} \\ & B(b) = \Box \begin{cases} after \rightarrow \star\langle F \rangle \rightarrow ack \rightarrow B(T) \\ goto \rightarrow B(b) \\ before \rightarrow B(F) \\ during \rightarrow B(F) \\ open \rightarrow B(b) \\ empty \rightarrow \text{if } a \text{ then } \star\langle T \rangle \rightarrow ack \rightarrow B(b) \text{ else } B(b) \end{cases} \\ & \text{within } B(F) \end{aligned}$$

We check using FDR that the adaptation pattern P_B only monitors the behaviour of the *access controller* and *match status* and only affects the execution of components through adaptation. We

verify that the adaptation pattern does not, unless adapting, refuse any events from the components eq. (9.3) or introduce events not in the interface of the components eq. (9.4),

$$RUN(E_{ac}, E_m) \quad \sqsubseteq_{F(ACSP)} \quad P_B \setminus \{|\star, ack|\} \tag{9.3}$$

$$P_B \setminus \{|\star, ack|\} \quad \sqsubseteq_{T(ACSP)} \quad RUN(E_{ac}, E_m) \tag{9.4}$$

The adaptation function adapts to AC_1 where visitors are directed to the corridor, when the section is not empty and to AC_2 once emptied.

$$F_B = \star\langle b\rangle \rightarrow \text{if } b \text{ then } ac!AC_2.ack \rightarrow F_B \text{ else } ac!AC_1.ack \rightarrow F_B$$

The adaptation procedure aimed at guaranteeing the satisfaction of Req. B is defined as the composition of the processes P_B and F_B

$$\Pi_B = (\nu\ \{|\star|\}, ack)\left(P_B \underset{\{|\star, ack|\}}{\parallel} F_B\right)$$

The *unary cluster* for Req. B is defined as the composition of the *access controller, match status* and the adaptation procedure Π_B

$$S_B = (\nu\, ac, ms)\left((AccessController \,|||\, MatchStatus) \underset{E_{ac}, E_m}{\parallel} \Pi_B\right)$$

We now discuss the specification for Req. B, $Spec_B$. The process R below encodes the most permissive behaviour by allowing all events from the *access controller* and *match status*. We use this process to describe the behaviour of the section before and during a match, when the requirement does not apply. On the *after* event, the process *AftEvt* describes the behaviour enforced by the requirement, where visitors are only allowed to leave s. Once the section is empty, visitors from adjacent sections can exit the stadium through s.

$$Spec_B = \text{let}$$

$$R = \Box \begin{cases} goto & \rightarrow R \\ after & \rightarrow AftEvt(F) \\ before & \rightarrow R \\ during & \rightarrow R \\ open & \rightarrow R \\ empty & \rightarrow R \end{cases} \qquad AftEvt(emp) = \underset{\substack{v \in Visitors \\ a \in adj \setminus Corr}}{\Box} \begin{cases} during & \rightarrow R \\ before & \rightarrow R \\ after & \rightarrow AftEvt(F) \\ goto_{v,s,Corr} & \rightarrow AftEvt(emp) \\ empty & \rightarrow AftEvt(T) \\ emp\ \&\ goto_{v,a,s} & \rightarrow AftEvt(emp) \\ emp\ \&\ open_{a,s} & \rightarrow AftEvt(emp) \\ \neg emp\ \&\ open_{s,a} & \rightarrow AftEvt(emp) \end{cases}$$

$$\text{within } R$$

We verify using FDR that for a section s, the cluster S_B trace refines the specification $Spec_B$,

$$Spec_B \sqsubseteq_{T(ACSP)} S_B$$

Requirement C: ***In the case of a fire alarm in a section, the section should be evacuated***
For this requirement, we preclude visitors from entering a section that is being evacuated. For the safety of visitors, we also allow visitors to exit to *empty* adjacent sections. Allowing visitors to exit through non-empty sections may lead to further congestion. We satisfy this requirement by

adapting the behaviour of the *access controller* and monitoring the *fire alarm*.

On a fire alarm, the section is closed to visitors and system polls adjacent sections to check if they are empty so visitors can also exit through them,

$$AC_1 = SExit \,|||\, \Box_{a \in adj} \Big\{ open_{s,a} \rightarrow SKIP$$

We now define an adaptation procedure Π_C that comprises an adaptation pattern P_C that triggers adaptation on the $alarm\langle T \rangle$ event and an adaptation function F_C that adapts the *access controller* behaviour to facilitate evacuation.

The process P_C monitors the *fire alarm* and *access controller* and triggers adaptation on the *fire alarm* event,

$$P_C = \Box \begin{cases} alarm\langle F \rangle & \rightarrow P_C \\ alarm\langle T \rangle & \rightarrow \star \rightarrow ack \rightarrow P_C \\ goto & \rightarrow P_C \\ open & \rightarrow P_C \\ empty & \rightarrow P_C \end{cases}$$

Akin to previous requirements, we can check using FDR that an adaptation pattern only monitors the behaviour of the components without influencing it. We verify that the adaptation pattern does not, unless adapting, refuses any events from the components eq. (9.5) or introduces events not in the interface of the components eq. (9.6),

$$RUN(Ev) \quad \sqsubseteq_{F(ACSP)} \quad P_C \setminus \{|\star, ack|\} \tag{9.5}$$

$$P_C \setminus \{|\star, ack|\} \quad \sqsubseteq_{T(ACSP)} \quad RUN(E_{ac}, E_f) \tag{9.6}$$

On an $\star$-event, the adaptation procedure adapts the behaviour of the *access controller* to the process AC_1 through a higher-order output on ac

$$F_C = \star \rightarrow ac!AC_1.ack \rightarrow F_C$$

The adaptation procedure is defined as the composition of the processes F_C and P_C

$$\Pi_C = (\nu \star, ack) \left(F_C \underset{\{\star, ack\}}{\parallel} P_C \right)$$

The *fire alarm* component is modelled as a stream of alternating *alarm* events. We encapsulate the process in location f

$$Alrm(b) = alarm\langle b \rangle \rightarrow Alrm(\neg b)$$

$$AlarmPanel = f\langle Alrm(T) \rangle$$

The *unary cluster* for Req. C for a section s is defined as

$$S_C = (\nu\, ac, f) \left((AccessController \,|||\, AlarmPanel) \underset{E_{ac}, E_f}{\parallel} \Pi_C \right)$$

We now discuss the specification for Req. C. We verify that on the $alarm\langle T \rangle$ event, visitors are

Unary cluster	Access Control	Floodlight	Noise Detector	Fire Alarm	Wind Monitor	Match Status
S_A	Δ					M
S_B	Δ					M
S_C	Δ			M		
S_D	Δ					M
S_E	Δ				M	M
S_F	Δ		M			M
S_G	Δ		M	M	M	M
S_H	M		Δ			M
S_I	M	Δ				M

Δ: Adapted Component, M: Monitored Component

Table 9.1: Overlaps in a section

precluded from entering the section. This behaviour is described in process *Evac*. The process R applies when the fire alarm is off and thus encodes the most permissive behaviour.

$$
\begin{aligned}
Spec_C = & \text{let} \\
& R = \Box \begin{cases} goto & \rightarrow R \\ alarm\langle T\rangle & \rightarrow Evac \\ alarm\langle F\rangle & \rightarrow R \\ open & \rightarrow R \\ empty & \rightarrow R \end{cases} \qquad Evac = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases} alarm\langle T\rangle & \rightarrow Evac \\ goto_{v,s,a} & \rightarrow Evac \\ open_{s,a} & \rightarrow Evac \\ empty & \rightarrow Evac \\ alarm\langle F\rangle & \rightarrow R \end{cases} \\
& \text{within } R
\end{aligned}
$$

With our verification approach, we verify that for a section s, the process S_C trace refines the process $Spec_C$

$$Spec_C \sqsubseteq_{T(ACSP)} S_C$$

9.5 Steps 5-6: Composing and Reverification of Overlapping Adaptation Procedures

We now proceed to Step 5 from our methodology presented in section 4.1, where we verify that the composition of adaptation procedures preserves the satisfaction of requirements.

In table 9.1, we summarize the adapted and monitored components for each unary cluster. Requirements A to G all adapt the behaviour of the *access controller* and thus their composition matches *case 4* in table 7.2. We need to reverify, using FDR, the satisfaction of the requirements by the composition of adaptation procedures $\Pi_A - \Pi_G$. The cluster S_H overlaps over the *noise detector* and *access controller* components. Its overlap with the clusters in $S_A - S_G$ matches *case 3* in table 7.2 which also require us to reverify all requirements. Since the overlaps all require us to reverify all the assertions, their order does not impact the number of reverification we need to perform. However, the adaptation procedures $\Pi_A - \Pi_G$ are very close to each other and thus it make sense to compose them together first to identify and resolve more likely conflicts early. The cluster S_I overlaps but does not adapt any components monitored in $S_A - S_H$. This means that

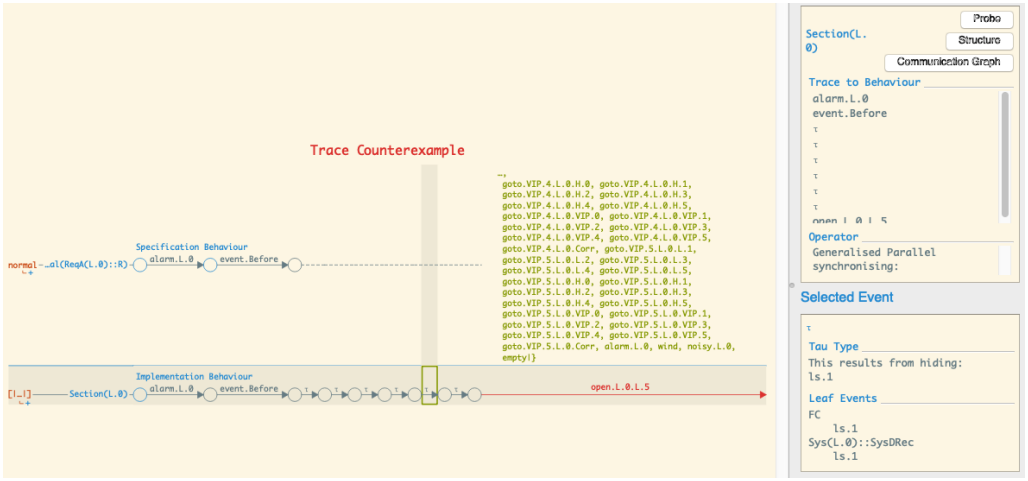


Figure 9.3: Violations introduced in $Spec_A$ from the Composition of S_A and S_B

for the composition of S_I with the other clusters, we only need to reverify S_I, because it matches *case 2* in table 7.2. We compose clusters in a section s alphabetically.

Composition of S_A and S_B

We compose the unary clusters S_A and S_B. We use the *merge* operation defined in Def. 7.2.1,

$$S_A \otimes S_B = (\nu\, ac, ms)\left((AccessController \;|||\; MatchStatus) \underset{E_{ac},E_m}{\parallel} \left(\Pi_A \underset{E_{ac},E_m}{\parallel} \Pi_B\right)\right)$$

We translate the process above to CSP_M using the tool presented in Chapter 8 and using FDR we verify that the specifications $Spec_A$ and $Spec_B$ are both trace-refined by $S_A \otimes S_B$, i.e.,

$$Spec_A \sqsubseteq_{T(ACSP)} S_A \otimes S_B \qquad (\texttt{REVERIFY})$$
$$Spec_B \sqsubseteq_{T(ACSP)} S_A \otimes S_B \qquad (\texttt{REVERIFY})$$

Composition of $S_A \otimes S_B$ and S_C

We now compose the clusters $S_A \otimes S_B$ with S_C. Using the *merge* operation, we know that

$$S_A \otimes S_B \otimes S_C =$$
$$(\nu\, ac, ms, f)\left((AccessController \;|||\; MatchStatus \;|||\; AlarmPanel) \underset{E_{ac},E_m,E_f}{\parallel} \left(\left(\Pi_A \underset{E_{ac},E_m}{\parallel} \Pi_B\right) \underset{E_{ac}}{\parallel} \Pi_C\right)\right)$$

We need to update all three specifications $Spec_A, Spec_B$ and $Spec_C$ because the scope of the merged adaptation procedure above contains the *fire alarm, access controller* and *match status* components and thus we extend the specification processes with first-order events from the missing components, i.e.,

$$
\begin{aligned}
&Spec'_A = \text{let} \\
&\quad R = \Box \begin{cases} goto & \to R \\ before & \to BefEvt \\ during & \to R \\ after & \to R \\ open & \to R \\ empty & \to R \\ alarm\langle b\rangle & \to R \end{cases} \qquad BefEvt = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases} goto_{s,a,s} & \to BefEvt \\ goto_{v,s,a} & \to BefEvt \\ during & \to R \\ after & \to R \\ before & \to BefEvt \\ empty & \to BefEvt \\ alarm\langle b\rangle & \to BefEvt \end{cases} \\
&\quad \text{within } R
\end{aligned}
$$

$$
\begin{aligned}
&Spec'_B = \text{let} \\
&\quad R = \Box \begin{cases} goto & \to R \\ after & \to AftEvt(F) \\ before & \to R \\ during & \to R \\ open & \to R \\ empty & \to R \\ alarm\langle b\rangle & \to R \end{cases} \qquad AftEvt(emp) = \Box_{\substack{v \in Visitors \\ a \in adj \setminus Corr}} \begin{cases} during & \to R \\ before & \to R \\ after & \to AftEvt(F) \\ goto_{v,s,Corr} & \to AftEvt(emp) \\ empty & \to AftEvt(T) \\ emp \ \& \ goto_{v,a,s} & \to AftEvt(emp) \\ emp \ \& \ open_{a,s} & \to AftEvt(emp) \\ \neg emp \ \& \ open_{s,a} & \to AftEvt(emp) \\ alarm\langle b\rangle & \to AftEvt(emp) \end{cases} \\
&\quad \text{within } R
\end{aligned}
$$

$$
\begin{aligned}
&Spec'_C = \text{let} \\
&\quad R = \Box \begin{cases} goto & \to R \\ alarm\langle T\rangle & \to Evac \\ alarm\langle F\rangle & \to R \\ open & \to R \\ empty & \to R \\ before & \to R \\ during & \to R \\ after & \to R \end{cases} \qquad Evac = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases} alarm\langle T\rangle & \to Evac \\ goto_{v,s,a} & \to Evac \\ open_{s,a} & \to Evac \\ empty & \to Evac \\ alarm\langle F\rangle & \to R \\ before & \to Evac \\ during & \to Evac \\ after & \to Evac \end{cases} \\
&\quad \text{within } R
\end{aligned}
$$

We verify the assertions below

$$Spec'_A \sqsubseteq_{T(ACSP)} (S_A \otimes S_B) \otimes S_C \qquad \texttt{(REVERIFY)}$$

$$Spec'_B \sqsubseteq_{T(ACSP)} (S_A \otimes S_B) \otimes S_C \qquad \texttt{(REVERIFY)}$$

$$Spec'_C \sqsubseteq_{T(ACSP)} (S_A \otimes S_B) \otimes S_C \qquad \texttt{(REVERIFY)}$$

The specifications $Spec'_A$ and $Spec'_B$ above are both violated.

The screen-shots in fig. 9.3 depicts the information we get from FDR for the violation in $Spec'_A$. On the left side, FDR shows a violating trace—the specification at the top and the implementation at the bottom, whereas on the right side, FDR provides information about selected events or states in the trace. In fig. 9.3, we see that the translated CSP$_M$ process for $(S_A \otimes S_B) \otimes S_C$ (the trace at the bottom) can perform the trace $alarm\langle L_0\rangle, before, open_{L_0,L_5}$ but the specification (the trace at the top) cannot. On clicking on the internal τ events, we see which adaptation events were performed and hidden to view the scheduling of adaptation commands that led to

Adaptation Procedure	Adaptation Trigger	Before	During	After
Π_C	*alarm*	AC_1	AC_1	AC_1
Π_E	*wind*	–	AC_4 or $SExit$	–
Π_F	*noisy*	–	AC_5	–
Π_G	*empty*	–	AC_3	$SExitAdj$
Π_A, Π_B, Π_D		$SOpen$	$SRoam$	AC_1

Priority (restricting access) ↑ high … low

Table 9.2: Expected behaviour of the access controller of a section when all adaptation procedures are composed together.

the violation. In our encoding, adaptations comprises the $\star$-event between the adaptation pattern and adaptation function, potential higher-order outputs from the adaptation function and the *ack*-event acknowledging the end of adaptation between the pattern and function. In this example, by clicking on the fifth τ event, FDR shows in the "selected event" panel at the bottom right of the window details about the τ event. The event $ac.1$ originating from the process FC has been hidden. We cross-reference this event with the translated CSP_M code to infer that the event $ac.1$ encodes the higher-order output $ac!AC_1$ from the adaptation procedure Π_C. By probing the other τ events, we know that an adaptation from Π_A (the first three τ events) was overwritten by an adaptation from Π_C.

This trace highlights two problems with our encoding that lead to a violation when the adaptation procedures are composed.

Precedence of Adaptations: In this example, the access to s is influenced by both Requirements A and C as it is before a match starts and also the fire alarm is on. The adaptation communicated to location *ac* by the adaptation procedures Π_A and Π_C conflicts. The adaptation procedure Π_A allows ticket holders to enter s, whereas Π_C looks to evacuate s. The safety of visitors is of utmost importance and therefore in this case access to s should follow Req. C, where a section should be evacuated. This also applies for the other components in s, if the fire alarm is on and a *wind* event or $noisy\langle T\rangle$ are observed in s, the access should not be changed and the system should proceed to evacuate s.

In table 9.2, we outline all the adaptations on the *access controller* before, during and after a match. The rows are ordered by priority, the rows at the top take precedence over lower rows. For instance, if the alarm in section U_0 is on, and thus the access controller is adapted to process AC_1 and then the wind became too unpleasant for visitors to be in the upper area; the section should not adapt and should stay with AC_1 because the *alarm* takes precedence over the *wind* event.

We thus update adaptation procedures $\Pi_A - \Pi_G$ so that adaptation patterns suppress adaptations if a higher precedence event has already been observed. For instance, we change the adaptation pattern P_E (reacts to the *wind* event) to monitor as well for the *alarm* event and delay adaptation if the *fire alarm* is on. In this case, the pattern P_E delays adaptation until the event $alarm\langle F\rangle$ is observed, i.e., the section is no longer in an emergency state. Similarly, the adaptation pattern P_D, which determines the default behaviour of the access controller during a match, must track the *noise detector, wind monitor* and *fire alarm* as they all take precedence over the default behaviour. From the table, we know that Π_C does not need to change as it is the highest priority adaptation procedure. Moreover, Π_A is not affected by the other adaptation procedures as the $before$ event marks the start of a match and thus happens before all other events.

Higher priority access controls are always stricter. For instance, when there is a strong wind, the access to upper sections is more restricting than if the section is noisy. Both only allow ticket holders to enter the section and both attempt to open an alternative section, but the wind adaptation

procedure closes the section once an alternative section is identified. This means that, we do not need to change specifications or requirements definitions to include precedence information.

Adaptation is not step-wise: The second problem highlighted in fig. 9.3 is that adaptation does not happen immediately after an important event. In the trace, we see that after the $alarm_{L_0}$ event, which should trigger adaptation from Π_C, the system continues to listen to first-order events and adaptation is delayed until after the $before$ event. This happens because the cluster S_C does not monitor for events in the *match status* component and thus adaptation from Π_C is interleaved with the first-order events from the *match status* component. We require that all adaptation procedures adapting the *access controller* to monitor all sensors in s, i.e., *wind monitor, noisy detector, fire alarm* and *match status* components. This ensures that adaptation in a section is step-wise with the execution of components and the section stops accepting first-order events until adaptation is complete. This also requires us to update the specifications to include the behaviour of any of the missing components.

To summarize, we must update the adaptation procedures $\Pi_A - \Pi_G$ to monitor all first-order events from the *fire alarm, wind monitor, noise detector* and *match status* components to ensure that adaptation happens step-wise with the execution. We also need to update the adaptation patterns to delay adaptations if a higher priority events has already been observed. Finally, we need to update the specifications to include the first-order events for the extended adaptation procedures scopes.

Recall that the adaptation procedures Π_A and Π_C do not need to be updated. We show how the encoding for Requirements D and E are updated. The others are similar.

Requirement D: ***During a match, visitors are allowed to roam in non-empty sections*** Here, we update the adaptation pattern P_D to monitor first-order events from the other controllers and delay adaptation if the access control is determined by a higher priority adaptation procedure, in this case Π_C or Π_E, but then trigger adaptation once all preceding events are switched off, e.g., adaptation should be triggered once the events $noisy\langle F\rangle$ or $alarm\langle F\rangle$ are observed. We define below a function *applies*, that returns true if the existing access control in the section is stricter than what is being proposed by the adaptation procedure We also update the pattern to also track the state of $noisy, alarm, wind$ levels in the section.

$$
\begin{array}{l}
P'_D = \text{let} \\
\quad applies(w,n,a,ev) = (s \notin \{|H|\} \vee \neg w) \wedge \neg n \wedge \neg a \wedge ev = during \\
\quad B(w,n,a,emp,ev) = \Box_{\substack{v \in Visitors \\ a \in adj}} \begin{cases}
goto_{v,a,s} & \rightarrow \text{if } applies(w,n,a,ev) \\
 & \quad \text{then } \star\langle F\rangle \rightarrow ack \rightarrow B(w,n,a,F,ev) \\
 & \quad \text{else } B(w,n,a,F,ev) \\
goto_{v,s,a} & \rightarrow B(others,emp,dur) \\
before & \rightarrow B(F,F,F,T,before) \\
after & \rightarrow B(w,n,a,emp,after) \\
open_{s,s'} & \rightarrow B(w,n,a,emp,ev) \\
open_{s',s} & \rightarrow B(w,n,a,F,ev) \\
during & \rightarrow \text{if } applies(w,n,a,ev) \text{ then } \star\langle emp\rangle \rightarrow ack \rightarrow B(w,n,a,emp,during) \\
 & \quad \text{else } B(w,n,a,emp,during) \\
empty & \rightarrow B(w,n,a,emp,ev) \\
noisy\langle b\rangle & \rightarrow \text{if } applies(w,b,a,ev) \text{ then } \star\langle emp\rangle \rightarrow ack \rightarrow B(w,b,a,emp,ev) \\
 & \quad \text{else } B(w,b,a,emp,ev) \\
alarm\langle b\rangle & \rightarrow \text{if } applies(w,n,b,ev) \text{ then } \star\langle emp\rangle \rightarrow ack \rightarrow B(w,n,b,emp,ev) \\
 & \quad \text{else } B(w,n,b,emp,ev) \\
wind & \rightarrow B(T,n,a,emp,ev)
\end{cases} \\
\quad \text{within } B(F,F,F,T,F)
\end{array}
$$

The adaptation function remains unaltered. We assume Π'_D replaces the adaptation pattern in Π_D with P'_D, The cluster for Req. D is

$$
S'_D = (\nu\, ac, np, ms, f)\left((AccessController \;|||\; MatchStatus \;|||\; AlarmPanel \;|||\; NoisePanel) \underset{E_{ac},E_m,E_n,E_f}{\|} \Pi'_D\right)
$$

We update the specification $Spec_D$ to include the missing first-order events from the *noise detector, wind monitor* and *alarm* controllers.

$$
\begin{array}{l}
Spec'_D = \text{let} \\
\quad R(emp) = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases}
before & \rightarrow R(T) \\
during & \rightarrow D(emp) \\
after & \rightarrow R(T) \\
goto_{v,s,a} & \rightarrow R(emp) \\
goto_{v,a,s} & \rightarrow R(F) \\
open & \rightarrow R(emp) \\
empty & \rightarrow R(emp) \\
noisy & \rightarrow R(emp) \\
wind & \rightarrow R(emp) \\
alarm & \rightarrow R(emp)
\end{cases} \quad
D(emp) = \Box_{\substack{a \in adj \\ v \in Visitors \\ s' \in SectionID \setminus s}} \begin{cases}
goto_{v,s,a} & \rightarrow D(emp) \\
emp \,\&goto_{s,a,s} & \rightarrow D(emp) \\
\neg emp \,\&goto_{v,a,s} & \rightarrow D(emp) \\
before & \rightarrow R(T) \\
after & \rightarrow R(T) \\
during & \rightarrow D(emp) \\
open_{s,s'} & \rightarrow D(emp) \\
emp \;\&\; open_{s',s} & \rightarrow D(emp) \\
empty & \rightarrow D(emp) \\
noisy & \rightarrow D(emp) \\
wind & \rightarrow D(emp) \\
alarm & \rightarrow D(emp)
\end{cases} \\
\quad \text{within } R(T)
\end{array}
$$

Requirement E: ***On a windy day, the system should attempt to empty the upper sections*** Similar to the previous requirement, we update the adaptation pattern P_E to monitor events from the other controllers and delay adaptation if the access control is determined by a more preceding adaptation procedure, in this case Π_C.

The adaptation pattern P_E also tracks the state of the *fire alarm*. Adaptation is suppressed on the *wind* event if the section is already being evacuated due to a fire alarm. Once the alarm is switched off encoded by the $alarm\langle F\rangle$ event, the pattern (if there are strong wind) triggers adaptation as it is the second preceding access control after the alarm. We define a function *applies* that returns true if the *alarm* is off and a match is ongoing. We also include in the adaptation pattern, the first-order events from the *noise detector* component to ensure adaptation occurs step-wise with the execution.

$$
\begin{aligned}
&P'_E = \text{let}\\
&\quad applies(a, ev) = \neg a \wedge ev = during\\
&\quad B(a, w, ev, bu) = \Box_{\substack{a \in adj \\ s' \in SectionID \setminus s}} \begin{cases}
goto & \rightarrow B(a, w, ev, bu)\\
before & \rightarrow B(F, F, before, F)\\
during & \rightarrow \text{if } w \wedge \neg others\\
& \qquad \text{then } \star\langle F\rangle \rightarrow ack \rightarrow B(a, w, during, bu)\\
& \qquad \text{else } B(a, w, during, bu)\\
after & \rightarrow B(a, w, F, bu)\\
open_{s,s'} & \rightarrow \text{if } applies(a, ev) \wedge w \wedge s' \notin \{\!|H, VIP|\!\}\\
& \qquad \text{then } \star\langle T\rangle \rightarrow ack \rightarrow B(a, w, ev, T)\\
& \qquad \text{else } B(a, w, ev, T)\\
open_{s',s} & \rightarrow B(a, w, ev, bu)\\
empty & \rightarrow B(a, w, ev, bu)\\
wind & \rightarrow \text{if } applies(a, ev)\\
& \qquad \text{then } \star\langle F\rangle \rightarrow ack \rightarrow B(a, T, ev, F)\\
& \qquad \text{else } B(a, T, ev, F)\\
noisy & \rightarrow B(a, w, ev, , bu)\\
alarm\langle b\rangle & \rightarrow \text{if } applies(b, ev) \wedge w\\
& \qquad \text{then } \star\langle F\rangle \rightarrow ack \rightarrow B(b, w, ev, bu)\\
& \qquad \text{else } B(b, w, ev, bu)
\end{cases}\\
&\quad \text{within } B(F, F, F, before)
\end{aligned}
$$

The adaptation function remains unaltered. We assume Π'_E replaces in Π_E the adaptation pattern with P'_E. The cluster for Req. E is

$$S'_E = (\nu\, ac, np, ms, f)\left((AccessController \;|||\; MatchStatus \;|||\; AlarmPanel \;|||\; NoisePanel) \underset{E_{ac}, E_m, E_n, E_f}{\|} \Pi'_E\right)$$

We update the specification $Spec_E$ to include the missing events from the *noise* and *alarm* components,

$$
\begin{aligned}
&Spec'_E = \text{let} \\
&\quad R(w) = \Box \begin{cases} goto & \to R(w) \\ wind & \to R(s \in \{|H|\}) \\ open & \to R(w) \\ before & \to R(F) \\ during & \to W(w,F) \\ after & \to R(w) \\ empty & \to R(w) \\ noisy & \to R(w) \\ alarm & \to R(w) \end{cases} \qquad W(w, backup) = \Box_{\substack{v \in Visitors \\ a \in adj \\ s' \in SectionID}} \begin{cases} w \wedge \ \& \ goto_{s,a,s} & \to W(w, backup) \\ \neg w \ \& goto_{v,a,s} & \to W(w, backup) \\ goto_{v,s,a} & \to W(w, backup) \\ empty & \to W(w, backup) \\ after & \to R(w) \\ before & \to R(F) \\ during & \to W(w,F) \\ wind & \to W(s \in \{|H|\}, F) \\ open_{s,s'} & \to W(w, w \wedge s' \notin \{|H|\}) \\ \neg w \ \& \ open_{s',s} & \to W(w, backup) \\ noisy & \to R(w) \\ alarm & \to R(w) \end{cases} \\
&\quad \text{within } R(F)
\end{aligned}
$$

We once again, using FDR, verify that the composition of the updated adaptation procedures satisfies the updated specifications $Spec'_A, Spec'_B, Spec'_C$

$$Spec'_A \sqsubseteq_{T(ACSP)} (S'_A \otimes S'_B) \otimes S'_C \qquad \texttt{(REVERIFY)}$$
$$Spec'_B \sqsubseteq_{T(ACSP)} (S'_A \otimes S'_B) \otimes S'_C \qquad \texttt{(REVERIFY)}$$
$$Spec'_C \sqsubseteq_{T(ACSP)} (S'_A \otimes S'_B) \otimes S'_C \qquad \texttt{(REVERIFY)}$$

9.5.1 Composition of S_A to S_G

After implementing the changes discussed in the previous section in the other adaptation procedures. We verify using FDR that the composition of adaptation procedures $\Pi_A - \Pi_G$ satisfies Requirements A to G. The overlap matches *case 4* in table 7.2, because all adaptation procedures adapt the behaviour of the access controller. We reverify the following assertions,

$$Spec'_A \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_G \qquad \texttt{(REVERIFY)}$$
$$Spec'_B \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_G \qquad \texttt{(REVERIFY)}$$
$$Spec'_C \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_G \qquad \texttt{(REVERIFY)}$$
$$Spec'_D \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_G \qquad \texttt{(REVERIFY)}$$
$$Spec'_E \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_G \qquad \texttt{(REVERIFY)}$$
$$Spec'_F \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_G \qquad \texttt{(REVERIFY)}$$
$$Spec'_G \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_G \qquad \texttt{(REVERIFY)}$$

Composition of $S_A \otimes \ldots \otimes S_G$ and S_H

The cluster S_H overlaps with the cluster $S_A \otimes \ldots \otimes S_G$ over the *match status*, *access controller* and *noise detector* component. We know that S_H monitors the *access controller* that is adapted by $S_A \otimes \ldots \otimes S_G$ and in turns the cluster $S_A \otimes \ldots \otimes S_G$ monitors the *noise detector* component that is adapted by S_H. This matches *case 3* in table 7.2, where we need to reverify all the specifications $Spec'_A - Spec_H$,

$$Spec'_A \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_H \qquad \texttt{(REVERIFY)}$$
$$Spec'_B \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_H \qquad \texttt{(REVERIFY)}$$
$$Spec'_C \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_H \qquad \texttt{(REVERIFY)}$$
$$Spec'_D \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_H \qquad \texttt{(REVERIFY)}$$
$$Spec'_E \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_H \qquad \texttt{(REVERIFY)}$$
$$Spec'_F \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_H \qquad \texttt{(REVERIFY)}$$
$$Spec'_G \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_H \qquad \texttt{(REVERIFY)}$$

The specification $Spec_H$ need to include the behaviour of the *fire alarm* and *wind monitor* components as these are now included in composed cluster,

$$Spec'_H = \text{let}$$
$$B(n, emp) = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases} before & \rightarrow B(n, T) \\ during & \rightarrow B(n, emp) \\ after & \rightarrow B(n, T) \\ \neg emp \ \&\, noisy\langle \neg n \rangle & \rightarrow B(\neg n, emp) \\ emp \wedge n \ \&\, noisy\langle F \rangle & \rightarrow B(F, emp) \\ goto_{v,a,s} & \rightarrow B(n, F) \\ goto_{v,s,a} & \rightarrow B(n, emp) \\ open & \rightarrow B(n, emp) \\ empty & \rightarrow B(n, T) \\ alarm\langle b \rangle & \rightarrow B(n, T) \\ wind & \rightarrow B(n, T) \end{cases}$$
$$\text{within } B(F, T)$$

We verify using FDR that the assertion below holds,

$$Spec'_H \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_H \qquad \texttt{(REVERIFY)}$$

Composition of $S_A \otimes \ldots \otimes S_H$ and S_I

The last cluster in section s to compose is S_I that adapts the *floodlights* component and monitors the *match status* and *access controller* components. This overlaps with $S_A \otimes \ldots \otimes S_G$ because it monitors the *access controller*, whose behaviour is changed by the other adaptation procedures. The overlap matches *case 2* in table 7.2, where only the satisfaction of $Spec_I$ may be affected by the composition, the satisfaction of $Spec'_A - Spec'_G$ follows from the compositionality theorem.

$$Spec''_A = Spec'_A \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) \sqsubseteq_{T(ACSP)} (S_A \otimes \ldots \otimes S_H) \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) = S_A \otimes \ldots \otimes S_I \quad \texttt{(FOLLOWS)}$$

$$Spec''_B = Spec'_B \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) \sqsubseteq_{T(ACSP)} (S_A \otimes \ldots \otimes S_H) \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) = S_A \otimes \ldots \otimes S_I \quad \texttt{(FOLLOWS)}$$

$$Spec''_C = Spec'_C \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) \sqsubseteq_{T(ACSP)} (S_A \otimes \ldots \otimes S_H) \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) = S_A \otimes \ldots \otimes S_I \quad \texttt{(FOLLOWS)}$$

$$Spec''_D = Spec'_D \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) \sqsubseteq_{T(ACSP)} (S_A \otimes \ldots \otimes S_H) \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) = S_A \otimes \ldots \otimes S_I \quad \texttt{(FOLLOWS)}$$

$$Spec''_E = Spec'_E \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) \sqsubseteq_{T(ACSP)} (S_A \otimes \ldots \otimes S_H) \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) = S_A \otimes \ldots \otimes S_I \quad \texttt{(FOLLOWS)}$$

$$Spec''_F = Spec'_F \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) \sqsubseteq_{T(ACSP)} (S_A \otimes \ldots \otimes S_H) \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) = S_A \otimes \ldots \otimes S_I \quad \texttt{(FOLLOWS)}$$

$$Spec''_G = Spec'_G \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) \sqsubseteq_{T(ACSP)} (S_A \otimes \ldots \otimes S_H) \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) = S_A \otimes \ldots \otimes S_I \quad \texttt{(FOLLOWS)}$$

$$Spec''_H = Spec'_H \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) \sqsubseteq_{T(ACSP)} (S_A \otimes \ldots \otimes S_H) \,|||\, (\nu\, lp) \left(LightPanel \underset{E_l}{\parallel} \Pi_I \right) = S_A \otimes \ldots \otimes S_I \quad \texttt{(FOLLOWS)}$$

We update the specification $Spec_I$ to include the behaviour of the *noise detector, wind monitor* and *fire alarm* because these are now in the scope of the composed cluster,

$$\begin{aligned}
Spec'_I = \text{let}\ & \\
& B(l, emp, ev) = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases}
before & \rightarrow B(l, T, before) \\
during & \rightarrow B(l, emp, during) \\
after & \rightarrow B(l, emp, after) \\
\neg emp \wedge \neg l \ \&\ floodlights\langle \neg l \rangle & \rightarrow B(\neg l, emp, ev) \\
\neg l \wedge e = before \ \&\ floodlights\langle T \rangle & \rightarrow B(T, emp, ev) \\
emp \wedge l \wedge ev \neq before \ \&\ floodlights\langle F \rangle & \rightarrow B(F, emp, ev) \\
goto_{v,a,s} & \rightarrow B(l, F, ev) \\
goto_{v,s,a} & \rightarrow B(l, emp, ev) \\
open_{s',s} & \rightarrow B(l, F, ev) \\
open_{s,s'} & \rightarrow B(l, emp, ev) \\
empty & \rightarrow B(l, emp \vee ev = after, ev) \\
noisy\langle b \rangle & \rightarrow B(l, emp, ev) \\
alarm\langle b \rangle & \rightarrow B(l, emp, ev) \\
wind & \rightarrow B(l, emp, ev)
\end{cases} \\
& \text{within } B(F, T)
\end{aligned}$$

We verify, using FDR, that the specification $Spec'_I$ is satisfied by the composition of adaptation procedures in a section,

$$Spec'_I \sqsubseteq_{T(ACSP)} S_A \otimes \ldots \otimes S_I \qquad \texttt{(REVERIFY)}$$

Here, we verified that the composition of all adaptation procedures in a section s satisfies all the requirements.

Composition of Sections in the Stadium

The satisfaction of the requirements for s also hold for the whole stadium. This follows from the compositionality theorem. From the component model in fig. 9.2, we know that sections overlap over the *wind monitor* and *match status* components. Because none of the adaptation

procedures adapt their behaviour, the overlap between sections matches *case 1* in table 7.2. As proven in Pro. 7.2.6, the satisfaction of requirements by the composition of this type of overlap is preserved. The stadium comprises 972 adaptation procedures, 9 adaptation procedures in each of the 108 sections, the composition of all the adaptation procedures would lead to computationally infeasible verification tasks. Even though the sections overlap, by making a distinction between the adapted and monitored components, we infer that interference is not possible, allowing us to derive the satisfaction from the compositionality theorem.

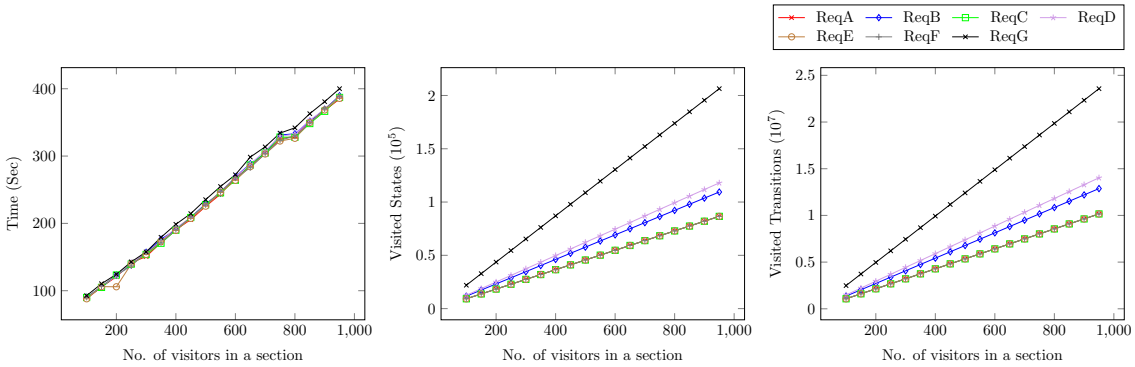
9.6 Scalability of Verification

We validate the tool presented in Chapter 8 by translating the $ACSP_M$ encoding for the stadium case-study to CSP$_M$. The tool produces a CSP$_M$ file with 229 lines of code, from the original 203 lines with $ACSP_M$ encoding.

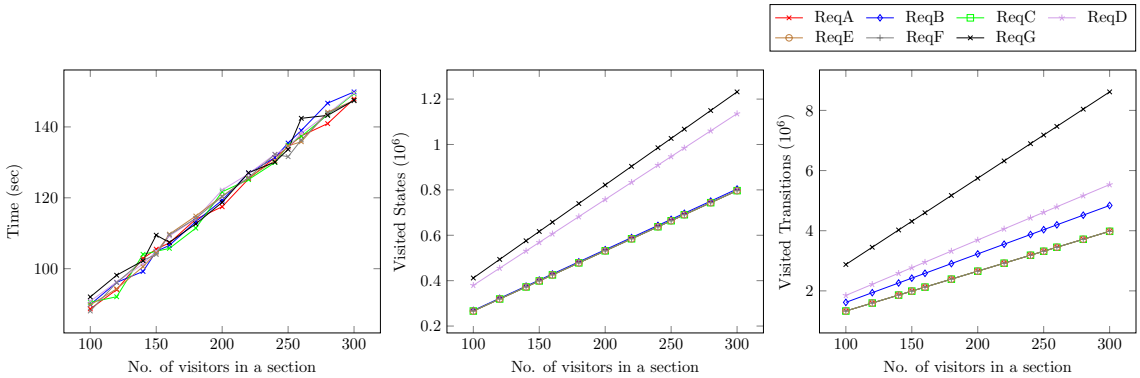
We also investigate the scalability of our verification approach for the stadium example. Here, we consider the verification of requirements by the composition of all adaptation procedures. We verify each requirement presented in section 9.2 with an increasing number of section sizes. For each verification, we measure the running time (in seconds) using Unix *time* command and the number of visited states and transitions from FDR output. This allows us to measure the order of growth of our verification approach. We run the experiment on a server that has 56 cores running at 2.2 GHz with 256GB of RAM. Here, we only show the results derived from the execution performed on the server machine.

In fig. 9.4a, we summarize the results when we verify each requirement independently after composition. We see that the model size grows linearly with the number of visitors, so verification time does the same. This is because adaptation procedures are all defined at the same level of granularity comprising all controllers and sensors found in a section. We verify the order of growth is not affected by FDR minimization by verifying the same assertions again with FDR minimization turned off. In fig. 9.4b, we attain the same order of growth, however the verification does not scale as well. In particular, for the assertion `ReqA [T= Section` with 200 visitors, with minimization FDR takes 120.73 seconds after visiting 18,309 states and 2,140,891 transitions and with no minimization the same assertion takes 117.48 seconds after visiting 531,416 states and 2,655,360. Note that the running time with no minimization is faster, however the process verified is much bigger in size and it because of the process size and the memory required to store such processes that the verification with no minimization does not scale. The minimization, in the stadium example, takes a considerable amount of the running time but reduces the size of the processes considerably, which allows us to verify bigger examples before memory runs out. The minimization removes internal transition that in our encoding represent adaptation commands. In this case-study, adaptation over a single section happens more frequent due to more requirements overlapping on the *access controller* in a section and thus the minimization is more effective.

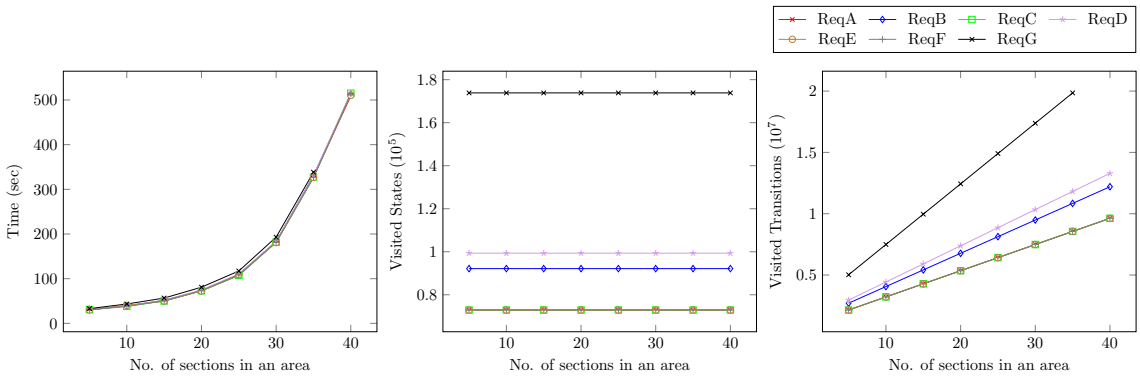
In fig. 9.4c, we investigate how the number of sections in the stadium affect the order of growth, despite the verification performed against a single section. Recall, the *open* event is a global event between pairs of sections. We see that the number of states is independent from the number of sections in the stadium, but the number of transitions is related to the number of sections and thus we observe a growth in the number of transitions that also results in an increase in running time for verifying the requirements in a single section.



(a) Experimental results for verifying the stadium requirements against a single section containing the composition of all adaptation procedures. Here, FDR minimization is enabled and the number of sections in a stadium area is set to 35.



(b) Experimental results for verifying the stadium requirements in a singe sections against the composition of all adaptation procedures. Here, FDR minimization is disabled and the number of sections in an area is set to 35. Note the varying scale on the x-axis, when compared with the graphs in fig. 9.4a.



(c) Experimental results for verifying the stadium requirements against a single section with the composition of all adaptation procedures for an increasing number of sections in an area. The section size for this experiment is set to 800 and FDR minimization is enabled.

Figure 9.4: Experimental results of verifying the stadium requirements against a single section, containing the composition of all adaptation procedures

9.7 Discussion

The purpose of this evaluation is to investigate the applicability of our *modelling methodology* and *verification process* for SA CPSs through a second case-study—a smart stadium inspired from [92]. In this chapter we follow the methodology presented in section 4.1 to model and verify the satisfaction of requirements defined in section 9.2.

In this case-study, we show how topology-driven modelling, discussed in Chapter 5, guides us to exploit the regularity in the physical layout of a stadium to reduce the state space for verification tasks. In total, there are 434 controllers installed in the stadium and including all controllers in verification tasks is not computationally feasible. Topology and topological relations guides the system designer to systematically explore different grouping of controllers that affect the satisfaction of a requirement.

In our framework, we also propose requirement-driven adaptation. We model an adaptation procedure for each requirement spanning over a subset of the components in a section. An adaptation procedure comprises an adaptation pattern that determines when a system *must*-adapt and adaptation function that plan and execute adaptations. This modularization is helpful when debugging violations, as the system designer can check whether a violation occurs due to the absence of adaptation or an ineffective adaptation. Given the nature of the process language, an adaptation procedure may potentially block or interfere with the execution of components by refusing to accept first-order events. The encoding for adaptation procedures enables us to check (using FDR) that this is not the case. First-order events are monitored by the adaptation pattern and since this process does not contain higher-order communication, we can using FDR, check that the adaptation pattern does not refuse or introduce first-order events.

Our methodology breaks down the verification process into two steps. We first verify that adaptation procedures do indeed satisfy system requirements. This is done over a small subset of the components that a system designer identify to affect the satisfaction of the requirement, independent from other adaptation procedures and the rest of the CPS. Once we determine that all adaptation procedures are correct (in isolation), we verify the composition of all adaptation procedures leads to a correct system. In this case-study, in comparison with the art gallery, we have more adaptation procedures overlapping on the same components—the *access controller* of a section in the stadium is adapted by 7 different adaptation procedures and monitored by 2 other adaptation procedures whereas *room A* in the *exhibition area*, which is the most adapted component in the art gallery is adapted by 3 adaptation procedures. As discussed in this chapter, this concentration of adaptation procedures over the *access controller* introduces a series of violations, e.g., unsynchronized and conflicting adaptations. This lead us to extend the scope of adaptation procedures to include all components in s and all the assertions over a section s needed to be reverified for each composition. However, the violations in the adaptation procedures were similar reducing the efforts required to resolve conflicts. Two observations are that requirement-driven adaptation helps us understand a problem better as we abstract away a lot of unnecessary details and the two step verification process force us to identify and resolve conflicts early. Moreover, through compositionality, we are able to replicate verification results for all sections and verify the satisfaction of requirements for the whole stadium with minimal verification efforts, even though the scope of adaptation procedures in different sections overlap over the *wind monitor* and *match status* components.

As discussed previously, the order of composition affects the verification efforts needed in the second step of the verification process. The order has to be decided by the system designer as there are different criteria that affect the efforts required for reverification. A systematic way to identify

the order requires further exploration of case-studies and maybe also new theoretical results.

We verify the satisfaction of requirements by leveraging FDR, an existing refinement checker. We present an adequate translation from a subset of the ACSP processes to CSP. This approach allows us to leverage the theory established for CSP in the ACSP process language, e.g., compositionality. This also includes the tool FDR that provides a GUI to improve the usability of the verification approach, algebraic laws proven in CSP and semantic models for ACSP processes. To further improve the usability of our framework, we also develop a concrete syntax for our process language and a tool that automatically translates the concrete syntax into CSP_M, the input language of FDR. The constructs introduced in the concrete syntax has undoubtedly improved the usability of the process language e.g., the enumerated sets and processes definition make the code more understandable. As future work, we intend to implement the functionality for the inclusion of parameters in communicated processes (over higher-order outputs) to further improve the framework application in realistic systems. In this case-study, the section identifier was set as a global variable. An alternative would be to have the section identifier passed as a parameter.

A limitation of this verification approach, and formal verification in general, is the mapping between requirements and specifications. The requirement is the English description of the intended behaviour, whereas the specification is the formal process encoding the requirement. With any formal verification approach, there is always a risk that the specification does not faithfully encode the requirement. It is the responsibility of the system designer to ensure that the specifications and requirements are close to each other as much as possible such that the mapping from the requirements to the specifications is obvious. This led us to require a single specification for each requirement that encodes at a very high-level of abstraction the requirement.

Another limitation is the encoding of the sensors. Once again this is not specific to our verification approach but to formal verification in general. We encode the detection of strong winds, fire alarm and high noise level as non-deterministic events that can occur intermittently. Verification is usually performed at a high-level of abstraction and thus cannot capture this low-level detail. We verify models of the system and if the model does not reflect faithfully the implementation then violations may still occur at runtime.

A threat to the validity of the evaluation, is the similarity to the art gallery example. However, as highlighted in section 9.2.1, this example is not only much larger in scale but also the overlapping of adaptation procedures is more dense e.g., *access controller* and in some cases global e.g., *match status*. In the art gallery, the overlaps involved mostly monitored components, whereas in this case the overlap involved the adapted components. In this example, we were able to leverage the regular structure that is typically found in large buildings to reuse most of our verification efforts.

Finally, topology-based modelling can potentially allow us to infer the correctness of even bigger case-studies. Consider a smart city, which contains an art gallery, two smart stadiums and autonomous vehicles that transport people between the buildings. From the topological layout of the city, we can potentially infer that the elements do not overlap and can thus be verified independently. With the overall correctness following from the compositionality theorem, it suffices to verify the art gallery, a stadium (assuming the two stadiums are homogeneous) and an autonomous vehicle (also assuming homogeneity) independently. In this dissertation, we already discuss the verification of an art gallery and a stadium and therefore show that we can leverage existing verification techniques to prove the correctness of complex realistic SA CPSs.

Chapter 10

Conclusions

We conclude the dissertation by summarizing the main contributions and outlining future research directions.

10.1 Summary

Cyber-Physical Systems (CPSs) must often self-adapt to respond to changes in their operating environment. However, providing assurances of critical requirements through formal verification techniques can be computationally intractable due to the large state space of self-adaptive (SA) CPSs. In this thesis, we tackle the complexity of modelling and verifying the satisfaction of security requirements in SA CPSs through *compositionality*.

We first propose a novel modelling language, which we call Adaptive CSP, to model and support compositional verification of SA CPSs. The process language extends Communicating Sequential Processes (CSP), with constructs that support the modelling of self-adaptation at a high level. Being process-based (as e.g., [89, 70]) and able to directly express self-adaptation, ACSP can readily support the definition of decentralized adaptation procedures at different levels of granularity in a system, as well as compositional reasoning for the system.

We also provide a *requirement-driven* methodology to model and verify such systems, and explore in ACSP alternative adaptation procedures for each requirement. This methodology allows the designer to leverage the *topological structure* [102] of CPSs and topological relations e.g., containment and connectivity, to explore different levels of granularity (i.e., grouping of CPS components) to encode adaptation procedures. The methodology—guided by the apparent topological structure and the requirements—offers a a systematic approach of how to effectively model such complex CPS behaviour.

We present a verification technique for SA CPSs encoded in ACSP that supports compositional reasoning and leverages existing verification tools. Here, we use the refinement-checker FDR [59], however the underlying principles of our technique is general enough to allow the use of other verification tools for process calculi. We highlight how our approach reduces the state space in verification tasks and thus provides a computationally tractable verification solution for realistic SA CPSs.

When different requirements involve disjoint sets of components, verification of each requirement can be performed against only its relevant components, independently from the rest of the system. A *compositionality theorem* guarantees that successful verification of a requirement over such a set of components implies the satisfaction of the requirement for the entire system. When components are relevant for multiple requirements, we need to explicitly address the potential of

interference created when the overlapping adaptation procedures are composed together and may require additional verification tasks. Due to compositionality results in ACSP, these verification tasks are needed only in certain types of overlaps (e.g., when two adaptation procedures modify the same components) and unnecessary in others (e.g., when the two adaptation procedures only monitor the same components), thus reducing the needed verification effort.

We showcase and evaluate our methodology using a running example of a smart art gallery. Moreover, we evaluate it by a case study of a modern sports stadium, where we use a prototype tool to translate the ACSP model to FDR and perform compositional verification. Our results show that our technique reduces the computational complexity of verifying self-adaptive CPSs.

10.2 Future Work

Our verification approach can be extended to incorporate known probability of certain events occurring. For example, an SA security system might have established statistics about the possibility of hacking or burglar attempts or the probability of bypassing specific security measures such as breaking a password. The probability of all events may not be known, e.g., human behaviour is considered to be non-deterministic, however utilizing knowledge of known probabilities in the verification process can improve its effectiveness [125]. The proposed extension can thus incorporate both probabilistic and non-deterministic behaviour. Once again, being entirely transition-based, a potential probabilistic extension can leverage established trace and testing equivalence theories and tools. Behavioural equivalence, in particular trace equivalence, is an effective technique for verifying safety properties of non-deterministic and probabilistic processes. Different definitions of trace equivalence for non-deterministic and probabilistic processes modelled by means of an extended LTS have been proposed [13].

Alternatives to CSP refinement can be explored as verification techniques, and potentially new ones may be developed for Adaptive CSP directly. Here our verification approach is applied only in parts of the system where all higher-order communications are internalized and hidden. The benefit of this was the ability to translate such system parts to FDR (a purely first-order process language) and reuse its advanced verification techniques. However, verification of adaptation procedures that span over many system components may still prove computationally hard. Recall how we change Req. 5 in the art gallery example, requiring visitors to have a clear exit from the building in the case of an emergency, because the original requirement spanned over the whole building and could not take advantage of our compositionality infrastructure. This may be overcome if reasoning techniques for higher-order events are developed, similar to those in [80, 81], which would allow us to define compositional reasoning for more refined parts of the system. Of course such techniques would require novel verification tools.

FDR allows us to reason about timing-constraints [59]. In our case-studies, we abstract over the notion of time. It would be an interesting direction of future work to extend the ACSP process language to express time and reason, using FDR, about time-sensitive requirements and timed events.

Our verification approach relies on the premise that requirements can be expressed easily with closely related CSP specifications. Any discrepancy between the two may potentially introduce errors and can negatively affect the effectiveness of verification. In Chapter 7, we discuss how the satisfaction of requirements may be affected when adaptation procedures are composed together. There, some specifications needed to be updated to incorporate behaviour of other requirements (see Ex. 7.1.3). This task is manually performed by the system designer and may introduce unintentional discrepancy between requirements and specifications when performed repeatedly. Systematic

software-engineering techniques guiding system designers to derive specifications from requirements are needed and would provide stronger assurances that requirements are indeed satisfied.

Appendix A

Proofs for Self-Adaptive Automata

Lemma about SAA Composition

Lemma A.0.1 (Intersection Determinism)**.** *For the SAAs* $\mathcal{M}_1 = \langle Q_1, \Sigma, \Delta_1, q_1, \delta_1, \Pi_1 \rangle$ *and* $\mathcal{M}_2 = \langle Q_2, \Sigma, \Delta_2, q_2, \delta_2, \Pi_2 \rangle$

$$M_1 \cap M_2 \text{ is deterministic}$$

Proof. Proven by contradiction. Assume there is t such that

$$\langle (q_1, q_2), \delta_1 \cap \delta_2 \rangle \xrightarrow{t} \langle (q_1', q_2'), \delta' \rangle \tag{IA.1}$$

$$\langle (q_1, q_2), \delta_1 \cap \delta_2 \rangle \xrightarrow{t} \langle (q_1'', q_2''), \delta'' \rangle \tag{IA.2}$$

such that $(q_1', q_2') \neq (q_1'', q_2'')$ or $\delta' \neq \delta''$. By Lem. A.0.7

$$\langle q_1, \delta_1 \rangle \xrightarrow{t} \langle q_1', \delta_1' \rangle \text{ and } \langle q_2, \delta_2 \rangle \xrightarrow{t} \langle q_2', \delta_2' \rangle$$
$$\langle q_1, \delta_1 \rangle \xrightarrow{t} \langle q_1'', \delta_1'' \rangle \text{ and } \langle q_2, \delta_2 \rangle \xrightarrow{t} \langle q_2'', \delta_2'' \rangle$$

such that $\delta' = \delta_1' \cap \delta_2'$ and $\delta'' = \delta_1'' \cap \delta_2''$ for some transition functions $\delta_1', \delta_2', \delta_1''$ and δ_2''. Since $\mathcal{M}_1$ and $\mathcal{M}_2$, it can never be the case that $q_1' \neq q_1''$ and $\delta_1' \neq \delta_1''$ (similarly for $\mathcal{M}_2$).

□

Lemma A.0.2 (Intersection Commutative)**.** *For SAAs* $\mathcal{M}_1 = \langle Q_1, \Sigma, \Delta_1, q_1, \delta_1, \Pi_1 \rangle$ *and* $\mathcal{M}_2 = \langle Q_2, \Sigma, \Delta_2, q_2, \delta_2, \Pi_2 \rangle$ *then*

$$M_1 \cap M_2 =_\alpha M_2 \cap M_1$$

Proof. Proven by structural induction on traces derived by each automata. □

Lemma A.0.3 (Intersection Associative)**.** *For* $i \in \{1, 2, 3\}$ *and SAAs* $\mathcal{M}_i = \langle Q_i, \Sigma, \Delta_i, q_i, \delta_i, \Pi_i \rangle$ *then*

$$(M_1 \cap M_2) \cap M_3 =_\alpha M_1 \cap (M_2 \cap M_3)$$

Proof. Proven by structural induction on traces derived by each automata. □

Lemma A.0.4 (Intersection Idempotent)**.** *Let* $\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta, \Pi \rangle$ *be an SAA then*

$$M \cap M =_\alpha M$$

Proof. Proven by structural induction on traces derived by each automata. □

Lemma A.0.5 (Intersection Id)**.** *Let* $\mathcal{M}_\top = \langle\{\bot\}, \Sigma, \{\delta\}, \bot, \delta, \Pi_{id}\rangle$ *where* $\delta = \mathsf{fn}(\bot, a) \Rightarrow \bot$ *for any* $a \in \Sigma$ *and* $\Pi_{id} = \mathsf{fn}(\bot) \Rightarrow \langle \bot, \delta\rangle$*, then for all SAA* $\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$ *implies*

$$\mathcal{M}_\top \cap \mathcal{M} =_\alpha \mathcal{M}$$

Lemma A.0.6 (Intersection Id 2)**.** *Let* $\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$ *be an SAA then*

$$M \cap \emptyset =_\alpha \emptyset$$

Proof. The empty automaton is defined as $\langle Q_2, \Sigma, q_2, \emptyset, \emptyset\rangle$ where the transition function and adaptation function are undefined for all $q \in Q_2$.

We define $M \cap \emptyset = \langle Q \times Q_2, \Sigma, (q_0, q_2), \delta_1 \cap \emptyset, \Pi\rangle$. Note that $\delta_0 \cap \emptyset = \emptyset$ as a transition has to be in both. Similarly $\Pi = \emptyset$ because the adaptation function has to be defined in both automata. □

Lemma A.0.7. *Assume* $\mathcal{M}_1 = \langle Q_1, \Sigma, \Delta_1, q_1, \delta_1, \Pi_1\rangle$ *and* $\mathcal{M}_2 = \langle Q_2, \Sigma, \Delta_2, q_2, \delta_2, \Pi_2\rangle$ *and* $\mathcal{M}_1 \cap \mathcal{M}_2 = \langle Q_1 \times Q_2, \Sigma, \Delta, (q_1, q_2), \delta_1 \cap \delta_2, \Pi\rangle$ *be SAAs then*

$$\langle (q_1, q_2), \delta_1 \cap \delta_2\rangle \xrightarrow{t} \langle (q_1', q_2'), \delta\rangle \text{ implies}$$
$$\langle q_1, \delta_1\rangle \xrightarrow{t} \langle q_1', \delta_1'\rangle \text{ and } \langle q_2, \delta_2\rangle \xrightarrow{t} \langle q_2', \delta_2'\rangle$$

such that $\delta = \delta_1' \cap \delta_2'$ *for some transition functions* δ_1' *and* δ_2'

Proof. Proven by structural induction on t

case $t = \epsilon$ *This means that* $(q_1, q_2) = (q_1', q_2')$*. Result follows by reflexivity of the reduction semantics.*

case $t = t'.a_\star$ *By transitivity,* $\langle (q_1, q_2), \delta_1 \cap \delta_2\rangle \xrightarrow{t} \langle (q_1', q_2'), \delta'\rangle$ *can be decomposed into*

$$\langle (q_1, q_2), \delta_1 \cap \delta_2\rangle \xrightarrow{t'} \langle (q_1'', q_2''), \delta''\rangle \tag{IA.3}$$
$$\langle (q_1'', q_2''), \delta''\rangle \xrightarrow{a_\star} \langle (q_1', q_2'), \delta'\rangle \tag{IA.4}$$

By IH, eq. (IA.3) *implies*

$$\langle q_1, \delta_1\rangle \xrightarrow{t'} \langle q_1'', \delta_1''\rangle \tag{IA.5}$$
$$\langle q_2, \delta_2\rangle \xrightarrow{t'} \langle q_2'', \delta_2''\rangle \tag{IA.6}$$
$$\delta'' = \delta_1'' \cap \delta_2'' \tag{IA.7}$$

By case-analysis on the structure of a

- $a_\star = \star$ *This means that*

$$\Pi(q_1'', q_2'') = \langle (q_1', q_2'), \delta'\rangle \tag{IA.8}$$

From the definition of Π*,*

$$\Pi(q_1'', q_2'') = \langle (q_1', q_2'), \delta_1' \cap \delta_2'\rangle \text{ where} \tag{IA.9}$$
$$\Pi(q_1'') = \langle q_1', \delta_1'\rangle \tag{IA.10}$$
$$\Pi(q_2'') = \langle q_2', \delta_2'\rangle \tag{IA.11}$$

This allows us to extend the reductions eq. (IA.5) *and eq.* (IA.6)

$$\langle q_1, \delta_1 \rangle \xrightarrow{t} \langle q_1'', \delta_1' \rangle \tag{IA.12}$$

$$\langle q_2, \delta_2 \rangle \xrightarrow{t} \langle q_2'', \delta_2' \rangle \tag{IA.13}$$

where $\delta' = \delta_1' \cap \delta_2'$.

- $a_\star \neq \star$ *Note that eq.* (IA.4) *is define as* $\delta''(\langle q_1'', q_2'' \rangle, a) = \langle q_1', q_2' \rangle$ *where* $\delta_1''(q_1'', a) = q_1'$ *and* $\delta_2''(q_2'', a) = q_2'$. *This allow us to construct the next reduction steps in eq.* (IA.5) *and eq.* (IA.6)

$$\langle q_1, \delta_1 \rangle \xrightarrow{t} \langle q_1'', \delta_1'' \rangle \tag{IA.14}$$

$$\langle q_2, \delta_2 \rangle \xrightarrow{t} \langle q_2'', \delta_2'' \rangle \tag{IA.15}$$

□

Lemma A.0.8. *Assume* $\mathcal{M}_1 = \langle Q_1, \Sigma, \Delta_1, q_1, \delta_1, \Pi_1 \rangle$ *and* $\mathcal{M}_2 = \langle Q_2, \Sigma, \Delta_2, q_2, \delta_2, \Pi_2 \rangle$ *and* $\mathcal{M}_1 \cap \mathcal{M}_2 = \langle Q_1 \times Q_2, \Sigma, \Delta, (q_1, q_2), \delta_1 \cap \delta_2, \Pi \rangle$ *be SAAs then*
$\langle q_1, \delta_1 \rangle \xrightarrow{t} \langle q_1', \delta_1' \rangle$ *and* $\langle q_2, \delta_2 \rangle \xrightarrow{t} \langle q_2', \delta_2' \rangle$ *implies* $\langle (q_1, q_2), \delta_1 \cap \delta_2 \rangle \xrightarrow{t} \langle (q_1', q_2'), \delta_1' \cap \delta_2' \rangle$

Proof. Proven by structural induction on t

case $t = \epsilon$ *immediate.*

case $t = t'.a_\star$ *By transitivity, we know that the reductions can be decomposed*

$$\langle q_1, \delta_1 \rangle \xrightarrow{t'} \langle q_1'', \delta_1'' \rangle \tag{IA.16}$$

$$\langle q_1'', \delta_1'' \rangle \xrightarrow{a_\star} \langle q_1', \delta_1' \rangle \tag{IA.17}$$

$$\langle q_2, \delta_2 \rangle \xrightarrow{t'} \langle q_2'', \delta_2'' \rangle \tag{IA.18}$$

$$\langle q_2'', \delta_2'' \rangle \xrightarrow{a_\star} \langle q_2', \delta_2' \rangle \tag{IA.19}$$

By IH with eq. (IA.16) *and eq.* (IA.18)

$$\langle (q_1, q_2), \delta_1 \cap \delta_2 \rangle \xrightarrow{t'} \langle (q_1'', q_2''), \delta_1'' \cap \delta_2'' \rangle \tag{IA.20}$$

By case-analysis on the structure of $a_\star$

- $a_\star = \star$ *This means that the reductions eq.* (IA.17) *and eq.* (IA.17) *happen because*

$$\Pi_1(q_1'') = \langle q_1', \delta_1' \rangle \tag{IA.21}$$

$$\Pi_2(q_2'') = \langle q_2', \delta_2' \rangle \tag{IA.22}$$

This allows us to extends the reduction in eq. (IA.20)

$$\langle (q_1, q_2), \delta_1 \cap \delta_2 \rangle \xrightarrow{t'.\star} \langle (q_1', q_2'), \delta_1' \cap \delta_2' \rangle \tag{IA.23}$$

- $a_\star \neq \star$ *This means that*

$$\delta_1''(q_1'', a_\star) = q_1' \text{ and } \delta_1' = \delta_1'' \tag{IA.24}$$

$$\delta_2''(q_2'', a_\star) = q_2' \text{ and } \delta_2' = \delta_2'' \tag{IA.25}$$

This allows us to extends the reduction in eq. (IA.20)

$$\langle (q_1, q_2), \delta_1 \cap \delta_2 \rangle \xrightarrow{t'.a_\star} \langle (q_1', q_2'), \delta_1'' \cap \delta_2'' \rangle \tag{IA.26}$$

□

Expressiveness of Self-Adaptive Automata

Here, we prove that SAA have the same power as EM by providing a bidirectional translation between the two models. This means that adding self-adaptation to automata, as we do in SAA, does not change the computational model. As shown in previous work for self-modifiable automata [106], this is not always the case.

Execution Monitors is a specific sub-class of Büchi automata. An EM $\mathcal{M}$ is formally defined as a tuple $\mathcal{M} = \langle Q, \Sigma, q_0, \delta \rangle$ where

- Q is a potentially infinite set of states
- Σ is a potentially infinite set of actions
- $q_0 \in Q$ is the initial state
- $\delta : Q \times \Sigma \rightharpoonup Q$ is a (partial) transition function.

We write $q \xrightarrow[\delta]{a} q'$ to represent a single transition $\delta(q, a) = q'$, and $q_0 \xrightarrow[\delta]{t}$ to denote the transitive application of δ, where t ranges over a potentially infinite sequence of symbols $a_0, a_1, \ldots$; i.e., $q_0 \xrightarrow[\delta]{a_0} q_1 \xrightarrow[\delta]{a_1} \ldots$.

Translating Self-Adaptive Automata into Execution Monitors

Intuitively, the transformation is a union of all instances of the SAA obtained by adaptations where the $\star$-transitions are forced whenever they are enabled.

The resulting automaton is the combination of all transition functions $\delta \in \Delta$ linked by Π. The transformed states $q_0(q_1)$ give us enough information to map back to the original state in SAA ; in this case $q_0(q_1)$ asserts that we are in state q_0 where the last adaptation was on state q_1.

In fig. A.1 we present two examples of this translation. The SAA at the top left is translated to the EM at the bottom left of the figure. Note that in this EM we force and hide the $\star$-transition, when enabled (grey state). The SAA at the top right of fig. A.1 is not an adaptation automaton because q_1 has both a $\star$- and an a-outgoing transition. Through the translation, the resulting automaton is not deterministic (state $q_1(q_1)$ has two outgoing transitions), and thus it is not an EM.

We show that for an adaptation automaton the translation accepts the same set of traces, modulo the $\star$-events. For a trace t, we let t^- be the trace t stripped of the $\star$-events.

Definition A.0.9. For an SAA $\mathcal{M} = \langle Q_\mathcal{A}, \Sigma_\star, \Delta, q_0, \delta_0, \prod \rangle$, we write $[\![\mathcal{M}]\!]$ for the EM $\langle Q, \Sigma, q_0, \delta \rangle$

for which

$$Q = Q_{\mathcal{A}} \uplus \{q(p) \mid q, p \in Q_{\mathcal{A}}\}$$

$$\delta(q,a) = \begin{cases} y'(y) & \text{if } q = x(p),\ \Pi(x) = \langle y, \delta' \rangle \text{ and } y' = \delta'(y,a) \\ y'(p) & \text{else if } q = x(p),\ \Pi(p) = \langle y, \delta' \rangle \text{ and } y' = \delta'(x,a) \\ y'(q) & \text{else if } q \in Q_{\mathcal{A}},\ \ \Pi(q) = \langle y, \delta' \rangle \text{ and } \delta'(y,a) = y' \\ y' & \text{else if } q \in Q_{\mathcal{A}},\ \delta_0(q,a) = y' \end{cases}$$

$\Diamond$

The following theorem shows that $[\![\mathcal{M}]\!]$ and $\mathcal{M}$ accept the same set of traces, and that $[\![\mathcal{M}]\!]$ is an EM, provided that $\mathcal{M}$ is deterministic.

Theorem A.0.10. *Suppose a SAA $\mathcal{M} = \langle Q_{\mathcal{A}}, \Sigma_\star, q_0, \delta_0, \Pi \rangle$ has adaptation automaton and $[\![M]\!] = \langle Q, \Sigma_\star, q_0, \delta \rangle$; then*

1. $\langle q_0, \delta_0 \rangle \xrightarrow{t} \ $ *iff* $q_0 \xrightarrow[\delta]{t^-}$
2. $[\![\mathcal{M}]\!]$ *is deterministic*
3. $\mathcal{M}$ *is finite implies* $[\![\mathcal{M}]\!]$ *is finite*

Proof. It follows directly as a corollary from Lem. A.0.11—Lem. A.0.13 below. □

Lemma A.0.11. *For an SAA $\mathcal{M} = \langle Q_{\mathcal{A}}, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$ and EM $[\![\mathcal{M}]\!] = \langle Q, \Sigma, q_0, \delta \rangle$ such that for all $\sigma \in \Sigma^*$*

$$\langle q_0, \delta_0 \rangle \xrightarrow{t} \langle q', \delta' \rangle \text{ implies } \exists\, q'' \in Q.q_0 \xrightarrow[\delta]{t^-} p'$$

where the structure of p' can be one of the following

1. $\big(p' = q', \delta_0 = \delta'\big)$ *or*
2. $\big(\exists\, x \in Q_{\mathcal{A}}.p' = q'(x) \text{ and } \langle \_, \delta' \rangle = \Pi(x)\big)$ *or*
3. $\big(\exists\, x \in Q_{\mathcal{A}}.p' = x(\_) \text{ and } \langle q', \delta' \rangle = \Pi(x)\big)$ *or*
4. $\big(\exists\, p' \in Q_{\mathcal{A}}.\langle q', \delta' \rangle = \Pi(p')\big)$.

Proof. Proven by structural induction on t

case $t = \epsilon$ *follows by reflexivity such that $q_0 = q' = q''$ and $\delta_0 = \delta'$.*

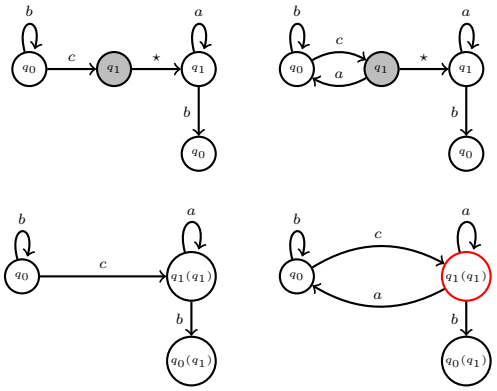


Figure A.1: The translation of an SAA to deterministic EM

case $t = t'.a_\star$

By transitivity, $\langle q_0, \delta_0\rangle \xrightarrow{t} \langle q', \delta'\rangle$ *can be broken down into*

$$\langle q_0, \delta_0\rangle \xrightarrow{t'} \langle q'', \delta''\rangle \tag{IA.27}$$

$$\langle q'', \delta''\rangle \xrightarrow{a_\star} \langle q', \delta'\rangle \tag{IA.28}$$

By IH and eq. (IA.27)

$$q_0 \xrightarrow{t'^-} p'' \tag{IA.29}$$

By case-analysis on the structure of $a_\star$ *and* q''

- $a_\star = \star$ *and* $\big(p'' = q'', \delta_0 = \delta'\big)$. *eq.* (IA.28) *could only happen if* $\Pi(q'') = \langle q', \delta'\rangle$. *Result follows from IH, such that* $p'' = q''(\_)$ *and* $\Pi(q'') = \langle q', \delta'\rangle$ *as required.*

- $a_\star = \star$ *and* $\big(\exists x \in Q_{\mathcal{A}}.p'' = q''(x)$ *and* $\langle \_, \delta''\rangle = \Pi(x)\big)$ *eq.* (IA.28) *could only happen if* $\Pi(q'') = \langle q', \delta'\rangle$. *This contradicts the initial assumption that* $\mathcal{M}$ *is a adaptation automaton as from* x *we can derive the transition* $\langle x, \_\rangle \xrightarrow{\star} \langle q'', \delta''\rangle \xrightarrow{\star} \langle q', \delta'\rangle$

- $a_\star = \star$ *and* $\big(\exists x \in Q_{\mathcal{A}}.p'' = x(\_)$ *and* $\langle q'', \delta''\rangle = \Pi(x)\big)$ *—analogous to the prev case.*

- $a_\star = \star$ *and* $\big(\exists p'' \in Q_{\mathcal{A}}.\langle q'', \delta''\rangle = \Pi(p'')\big)$ *eq.* (IA.28) *could only happen if* $\Pi(q'') = \langle q', \delta'\rangle$. *This contradicts the initial assumption that* $\mathcal{M}$ *is adaptation automaton as from* p'' *we can derive the transitions* $\langle p'', \_\rangle \xrightarrow{\star} \langle q'', \delta''\rangle \xrightarrow{\star} \langle q', \delta'\rangle$

- $a_\star \neq \star$ *and* $\big(p'' = q'', \delta_0 = \delta'\big)$ *eq.* (IA.28) *could only happen if* $\delta_0(q'', a) = q'$. *Result follows from translation* $\delta(p'', a) = \delta_0(q'', a) = q'$

- $a_\star \neq \star$ *and* $\big(\exists x \in Q_{\mathcal{A}}.p'' = q''(x)$ *and* $\langle \_, \delta''\rangle = \Pi(x)\big)$*eq.* (IA.28) *could only happen if* $\delta''(q'', a) = q'$ *and* $\delta' = \delta''$. *Result follows from translation* $\delta(p'', a) = q'(x)$

- $a_\star \neq \star$ *and* $\big(\exists x \in Q_{\mathcal{A}}.p'' = x(\_)$ *and* $\langle q'', \delta''\rangle = \Pi(x)\big)$ *eq.* (IA.28) *could only happen if* $\delta''(q'', a) = q'$ *and* $\delta' = \delta''$. *Result follows from translation* $\delta(p'', a) = q'(x)$

- $a_\star \neq \star$ *and* $\big(\exists p'' \in Q_{\mathcal{A}}.\langle q'', \delta''\rangle = \Pi(p'')\big)$ *eq.* (IA.28) *could only happen if* $\delta''(q'', a) = q'$ *and* $\delta' = \delta''$. *Result follows from translation* $\delta(p'', a) = q'(p'')$

□

Lemma A.0.12. *For anadaptation automaton* $\mathcal{M} = \langle Q_{\mathcal{A}}, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$ *and EM* $[\![\mathcal{M}]\!] = \langle Q, \Sigma, q_0, \delta\rangle$ *such that for all* $t \in A^*$

$$q_0 \xrightarrow[\delta]{t} p' \text{ implies } \exists\, q' \in Q_{\mathcal{A}}, \delta' \in \Delta.\langle q_0, \delta_0\rangle \xrightarrow{\sigma} \langle q', \delta'\rangle$$

for some $\sigma \in \Sigma^*$ *such that* $\sigma = t^-$. *The structure of* p' *can be either* $\big(p' = q', \delta_0 = \delta'\big)$ *or* $\big(\exists x \in Q_{\mathcal{A}}.p' = q'(x)$ *and* $\langle \_, \delta'\rangle = \Pi(x)\big)$

Proof. Proven by structural induction on t

case $t = \epsilon$ *follows by reflexivity such that* $q_0 = q' = q''$ *and* $\delta_0 = \delta'$.

case $t = t'.a$

By transitivity, $q_0 \xrightarrow[\delta]{t} p'$ *can be broken down into*

$$q_0 \xrightarrow[\delta]{t'} p'' \tag{IA.30}$$

$$p'' \xrightarrow[\delta]{a} p' \tag{IA.31}$$

By IH and eq. (IA.30)

$$\langle q_0, \delta_0 \rangle \xrightarrow{\sigma'} \langle q'', \delta'' \rangle \tag{IA.32}$$

for some $\sigma' \in \Sigma^*$ *such that* $\sigma'^{-} = t'$*. By case-analysis on the structure of* p'' *and* $\delta(p'', a)$

- $\big(p'' = q'', \delta_0 = \delta'\big)$*,* $\delta(p'', a) = y'(y)$ *such that* $\Pi(p'') = \langle y, \delta' \rangle$ *and* $\delta'(y, a) = y'$ *We can construct the derivation*

$$\langle q_0, \delta_0 \rangle \xrightarrow{\sigma'} \langle p'', \delta_0 \rangle \xrightarrow{\star} \langle y, \delta' \rangle \xrightarrow{a} \langle y', \delta' \rangle$$

- $\big(p'' = q'', \delta_0 = \delta'\big)$*,* $\delta(p'', a) = \delta_0(p'', a) = p'$ *such that* $\Pi(p'')$ *is undefined. We can construct the derivation*

$$\langle q_0, \delta_0 \rangle \xrightarrow{\sigma'} \langle p'', \delta_0 \rangle \xrightarrow{a} \langle p', \delta_0 \rangle$$

- $\big(\exists\, x \in Q_{\mathcal{A}}.p'' = q''(x)$ *and* $\langle \_, \delta'' \rangle = \Pi(x)\big)$*,* $\delta(p'', a) = y'(y)$ *such that* $\Pi(q'') = \langle y, \delta' \rangle$ *and* $\delta'(y, a) = y'$ *We can construct the derivation*

$$\langle q_0, \delta_0 \rangle \xrightarrow{\sigma'} \langle q'', \delta'' \rangle \xrightarrow{\star} \langle y, \delta' \rangle \xrightarrow{a} \langle y', \delta' \rangle$$

- $\big(\exists\, x \in Q_{\mathcal{A}}.p'' = q''(x)$ *and* $\langle \_, \delta'' \rangle = \Pi(x)\big)$*,* $\delta(p'', a) = y'(x)$ *such that* $\Pi(q'')$ *is undefined and* $\delta''(q'', a) = y'$ *We can construct the derivation*

$$\langle q_0, \delta_0 \rangle \xrightarrow{\sigma'} \langle q'', \delta'' \rangle \xrightarrow{a} \langle y', \delta'' \rangle$$

□

Lemma A.0.13. *For a deterministic SAA* $\mathcal{M} = \langle Q_{\mathcal{A}}, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$ *and EM* $[\![\mathcal{M}]\!] = \langle Q, \Sigma, q_0, \delta \rangle$

$$\delta(q, a) = q' \text{ and } \delta(q, a) = q'' \text{ implies } q' = q''$$

Proof. By case-analysis on $\delta(q, a)$

case $\delta(q, a) = y'(y)$ *such that* $q = x(p)$*,* $\Pi(x) = \langle y, \delta \rangle$ *and* $y' = \delta(y, a)$ *Result follows from determinism of* Π *and* δ

case $\delta(q, a) = y'(p)$ *such that* $q = x(p)$*,* $\Pi(x)$ *is undefined,* $\Pi(p) = \langle y, \delta \rangle$ *and* $y' = \delta(x, a)$ *Result follows from determinism of* Π *and* δ

case $\delta(q, a) = y(q)$ *such that* $q \in Q_{\mathcal{A}}$*,* $\Pi(q) = \langle y, \delta \rangle$ *and* $\delta(y, a) = y'$ *Result follows from determinism of* Π *and* δ

case $\delta(q, a) = y'$ *such that* $q \in Q_{\mathcal{A}}$*,* $\Pi(q)$ *is undefined,* $\delta_0(q, a) = y'$ *Result follows from determinism of* Π *and* δ_0

□

FDR Code for Vehicles Example

```
-- Configurations
size = 7

Pos = {(a,b) | a <- {0..size+1}, b<-{0..size+1}}
GoodPos = {(a,b) | a <- {1..size}, b<-{1..size}}

loc1 = (2,2)
loc2 = (4,5)

channel g,s : Pos.Pos
channel adapt

Loc = {(a,b) | a <- Pos, b<- Pos}

-----------------------------------------
--              Wiring
-----------------------------------------

Sys(strategy,buff) =
let
 adaptation = s?x: GoodPos ?y :GoodPos ->  Adapt(x,y,buff)
 Mach = (V(loc1,loc2) /+ {|s|} +\ adaptation)
 abstractMach = Mach [[s.x.y <- adapt | x <- GoodPos, y <- GoodPos]]
within
 abstractMach  [| {|g,adapt|} |] strategy


-----------------------------------------
--              Adaptation Function
-----------------------------------------

Adapt(x,y,t) =
let
 (a,b) = fix(x,y,t)
 continuation = V(a,b) /+ {|s|} +\ s?a:GoodPos?b:GoodPos -> Adapt(a,b,t)
within
 g!a!b -> continuation

-----------------------------------------
--              Transition Functions
-----------------------------------------

V(p1,p2) = let
 correctMoves = \p @ {q1 | q1 <- Pos, dis(q1,p) <= 1}
within
 (g?q1 : correctMoves(p1) ?q2 :correctMoves(p2) -> V(q1,q2))
 [] (s!p1!p2 -> STOP)

-----------------------------------------
```

```
--              Adaptation Strategies
-----------------------------------------------

-- adapts when the vehicles are b units apart
close(b) = let
 correctMoves = \x @ {y | y<-GoodPos,  goodLoc(x,y,b)}
 wrongMoves = \x @ diff(GoodPos, correctMoves(x))
within
   (g?x : GoodPos ?y : correctMoves(x) -> close(b))
 [] (g?x : GoodPos ?y : wrongMoves(x) -> adapt -> close(b))

-- adapts every 2 steps
--(we skip the first g after adapt as that is from the co-ordinatoor)
every2Steps = let
 correctMoves = \p @ {q1 | q1 <- Pos, dis(q1,p) <= 1}
within
    g?x : GoodPos ?y : GoodPos
  -> g?x2 : correctMoves(x) ?y2 : correctMoves(y)
  -> adapt
  -> g?x : GoodPos ?y : GoodPos
  -> every2Steps

-- adapts every other step
--(we skip the first g after adapt as that is from the co-ordinatoor)
everyStep = g?_ : GoodPos ?_ : GoodPos -> adapt
 -> g?x : GoodPos ?y : GoodPos -> everyStep

transparent sbisim, wbisim, diamond,normal
-----------------------------------------------
--              Helper Functions
-----------------------------------------------

fix(p1,p2,buffer) =
 if (not goodLoc(p1,p2,buffer) )
 then getLoc(p1,p2,buffer)
 else (p1,p2)

goodLoc(q1,q2,buffer) = dis(q1,q2) > buffer and not outOfBound(q1,buffer) and not
    outOfBound(q2,buffer)

max(x,y) = if(x > y) then x else y
min(x,y) = if (x > y) then y  else x
minPos(x,y) = if x < y then True else False
disA(x,y) = max(x,y) - min(x,y)

getLoc2(p1,p2,d,buf) =
let
 q1 = min(p1,p2)
 q2 = max(p1,p2)
 d1 = disA(q1,q2)
 t1 = moveLeft(q1,buf -d1 -d + 1,buf+1)
```

```
 d2 = disA(t1,q2)
 t2 = moveRight(q2,buf -d -d2+1,size-buf)
within
 if(p1 <= p2) then (t1,t2) else (t2,t1)

moveToBuffer(p1,buf) = min(max(p1,buf+1),size-buf)

moveLeft(x,dist,min) =
if(x == min or dist <= 0) then x else moveLeft(x-1,dist-1,min)


moveRight(x,dist,mx) =
if(x == mx or dist <= 0) then x else moveRight(x+1,dist-1,mx)

collide((a,b),(c,d), (x,y),(w,z)) =
 minPos(a,c) != minPos(x,w) or minPos(b,d) != minPos(y,z)

outOfBound((x,y),buff) = (x + buff > size)
 or (x - buff <= 0) or (y + buff > size) or (y - buff <= 0)

getLoc((x1,y1),(x2,y2),buf) =
let
 x3 = moveToBuffer(x1,buf)
 x4 = moveToBuffer(x2,buf)
 y3 = moveToBuffer(y1,buf)
 y4 = moveToBuffer(y2,buf)
 d1 = max(disA(y3,y4)-1,0)
 (p1,p2) = getLoc2(x3,x4,d1,buf)
 d2 = max(disA(p1,p2)-1,0)
 (q1,q2) = getLoc2(y3,y4,d2,buf)
within
 ((p1,q1),(p2,q2))


-----------------------------------------
--              Assertions
-----------------------------------------

SEveryStep = Sys(everyStep,2)

-- Safety Properties
NoCollision = let
 correctMoves = \x @ { y | y <- Pos, dis(y,x)>0}
within (g?x : Pos ?y : correctMoves(x) -> NoCollision)
 [] (adapt -> g?x : Pos ?y : correctMoves(x) -> NoCollision)

WithinBounds =
let correctMoves = \x @ { y | y <- GoodPos}
within (g?x : GoodPos ?y : correctMoves(x) -> WithinBounds)
  [] (adapt -> g?x : GoodPos ?y : correctMoves(x) -> WithinBounds)

assert NoCollision [T= SEveryStep
```

```
assert WithinBounds [T= SEveryStep

-- Adaptation strategy and composed automata are strongly adaptable
StronglyAdaptable =
 (g?_ : GoodPos ?_: GoodPos -> StronglyAdaptable)
  |~| (adapt -> g?x :GoodPos ?y:GoodPos -> g?_: GoodPos  ?_:GoodPos  ->StronglyAdaptable)
assert StronglyAdaptable [T= everyStep
assert StronglyAdaptable [T= SEveryStep

-- Composition follows the adaptation strategy
assert everyStep [T= SEveryStep

-- Composition yields a deterministic, deadlock and livelock free system
assert SEveryStep :[deterministic [F]]
assert SEveryStep :[deadlock free [F]]
assert SEveryStep :[divergence-free [FD]]

-- We show that changing the strategy is still safe
assert Sys(close(2),2)  \ {|adapt|} [T=  SEveryStep \ {|adapt|}

-- Eventually reaches every location
channel success,ok

Eventually =
let
 correctMoves = {(g.a.b) | a<- GoodPos,b <- GoodPos,dis(a,b) > 0}
 testCases = ([| correctMoves |]  p : GoodPos @ T(p,correctMoves))
 SuccessProc = (SEveryStep [| correctMoves |] testCases) |\ {success}
 OkProc = (; i : seq(GoodPos) @ success -> SKIP); (ok -> STOP)
within
 (SuccessProc [| {success} |] OkProc) |\ {ok}

T(p,correctMoves) = g?x : GoodPos ?y : { y | y<- GoodPos, member(g.x.y,correctMoves)}
 -> if(x==p or y == p) then success-> RUN(correctMoves)  else T(p,correctMoves)

assert ok -> STOP [F= Eventually
assert ok -> STOP [T= Eventually

-- Because the vehicles choose the next goto position at random, the implementation
    contains divergences
-- These assertions rightfully fail
assert Eventually :[divergence-free]
assert ok -> STOP [FD= Eventually
```

Appendix B

Proofs for Verification Technique

Bisimulation and Progress Theorems

Lemma B.0.1 (Well-Formed Progress)**.** $\Gamma \vdash P$ *then*

- $P \xrightarrow{e} P'$ *implies* $\Gamma \vdash P'$
- $P \xrightarrow{l!R} P'$ *implies* $\Gamma \vdash P'$ *and* $\emptyset \vdash R$ *and* $l \in \Gamma$
- $P \xrightarrow{l?R} P'$ *and* $\emptyset \vdash R$ *implies* $\Gamma \vdash P'$ *and* $l \in \Gamma$

Proof. By rule induction on $\Gamma \vdash P$

wPar

$$\textbf{case}\ \frac{\Gamma \uplus L \vdash Q_1 \quad \Gamma \uplus L \vdash Q_2 \qquad in(Q_2) \cap in(Q_1) = \emptyset \quad out(Q_1) \cap out(Q_2) \subseteq \Gamma \quad in(Q_2) \cap out(Q_1) \subseteq L \quad in(Q_1) \cap out(Q_2) \subseteq L}{\Gamma \vdash (\nu\, L)Q_1 \underset{E}{\parallel} Q_2}$$

We know that

$$\Gamma \uplus L \vdash M \tag{IB.1}$$
$$\Gamma \uplus L \vdash N \tag{IB.2}$$
$$in(M) \cap in(N) = \emptyset \tag{IB.3}$$
$$out(M) \cap out(N) \subseteq \Gamma \tag{IB.4}$$
$$in(N) \cap out(M) \subseteq L \tag{IB.5}$$
$$in(M) \cap out(N) \subseteq L \tag{IB.6}$$

By case-analysis on $(\nu\, L)M \underset{E}{\parallel} N \xrightarrow{\alpha} T$,

- *evEsc which implies*

$$M \underset{E}{\parallel} N \xrightarrow{e} T' \tag{IB.7}$$
$$T = (\nu\, L)T' \tag{IB.8}$$
$$e \notin L \tag{IB.9}$$

By case-analysis again on $M \underset{E}{\parallel} N \xrightarrow{e} T'$*, there are three sub-cases (omitting their symmetric cases)*

– *evParL*: $M \xrightarrow{e} M'$ *such that* $e \notin E$. *By IH with eq.* (IB.1) *we infer*

$$\Gamma \uplus L \vdash M' \tag{IB.10}$$

By Lemma B.0.5 and B.0.6 and the transitivity with eqs. (IB.3) *to* (IB.6) *implies*

$$in(M') \cap in(N) = \emptyset \tag{IB.11}$$
$$out(M') \cap out(N) \subseteq \Gamma \tag{IB.12}$$
$$in(N) \cap out(M') \subseteq L \tag{IB.13}$$
$$in(M') \cap out(N) \subseteq L \tag{IB.14}$$

By wPar and eqs. (IB.2) *and* (IB.11) *to* (IB.14), *we infer*

$$\Gamma \vdash (\nu\, L)M' \underset{E}{\parallel} N \tag{IB.15}$$

– *evSync*: $M \xrightarrow{e} M'$ *and* $N \xrightarrow{e} N'$ *such that* $e \in E$—*analogous to the previous case.*

– *adSyncL*: $M \xrightarrow{l!R} M'$ *and* $N \xrightarrow{l?R} N'$ *such that* $e = \tau$

By IH with eq. (IB.1)

$$\Gamma \vdash M' \tag{IB.16}$$
$$l \in \Gamma \uplus L \tag{IB.17}$$
$$\emptyset \vdash R \tag{IB.18}$$

By IH with eqs. (IB.2) *and* (IB.18)

$$\Gamma \vdash N' \tag{IB.19}$$
$$l \in \Gamma \uplus L \tag{IB.20}$$

By Lemma B.0.5 and B.0.6 and the transitivity with eqs. (IB.3) *to* (IB.6) *implies*

$$in(M') \cap in(N') = \emptyset \tag{IB.21}$$
$$out(M') \cap out(N') \subseteq \Gamma \tag{IB.22}$$
$$in(N') \cap out(M') \subseteq L \tag{IB.23}$$
$$in(M') \cap out(N') \subseteq L \tag{IB.24}$$

By wPar eqs. (IB.21) *to* (IB.24), *we infer*

$$\Gamma \vdash (\nu\, L)M' \underset{E}{\parallel} N \tag{IB.25}$$

• *adEsc (this is analogous to the previous sub-case) which implies*

$$M \underset{E}{\parallel} N \xrightarrow{h} T' \tag{IB.26}$$
$$T = (\nu\, L)T' \tag{IB.27}$$
$$h \notin L \tag{IB.28}$$

By case-analysis on $M \underset{E}{\parallel} N \xrightarrow{h} T'$*: only* AdParL *(and its symmetric rule) is applicable where*

$$M \xrightarrow{h} M' \tag{IB.29}$$

By IH with eqs. (IB.1) *and* (IB.29) *we infer*

$$\Gamma \uplus L \vdash M' \tag{IB.30}$$
$$h \in \Gamma \uplus L \tag{IB.31}$$

By Lemma B.0.5 and B.0.6 and the transitivity with eqs. (IB.3) *to* (IB.6) *implies*

$$in(M') \cap in(N) = \emptyset \tag{IB.32}$$
$$out(M') \cap out(N) \subseteq \Gamma \tag{IB.33}$$
$$in(N) \cap out(M') \subseteq L \tag{IB.34}$$
$$in(M') \cap out(N)) \subseteq L \tag{IB.35}$$

By wPar *eqs.* (IB.2) *and* (IB.32) *to* (IB.35)*, we infer*

$$\Gamma \vdash (\nu\, L)M' \underset{E}{\parallel} N \tag{IB.36}$$

By eqs. (IB.28) *and* (IB.31)*, we infer that* $h \in \Gamma$*.*

case $\dfrac{l \in \Gamma \quad \emptyset \vdash P}{\Gamma \vdash l\langle P\rangle}\ \textsc{wLoc}$ *We know*

$$l \in \Gamma \tag{IB.37}$$
$$\emptyset \vdash P \tag{IB.38}$$

By case-analysis on $l\langle P\rangle \xrightarrow{\alpha} T$*, there are three sub-cases*

- evLoc *where we also know*

$$P \xrightarrow{e} P' \tag{IB.39}$$
$$T = l\langle P'\rangle \tag{IB.40}$$

By IH with eqs. (IB.38) *and* (IB.39)

$$\emptyset \vdash P' \tag{IB.41}$$

result follows from wLoc *and eq.* (IB.37)*.*

- adLoc *Let's consider the case* $h = l?R$ *for some location* l *and process* R *such that* $\emptyset \vdash R$*, by IH and eq.* (IB.38) *and* $P \xrightarrow{h} P'$ *we know*

$$\emptyset \vdash P' \tag{IB.42}$$
$$l \in \emptyset \tag{IB.43}$$

contradiction as we do not allow nested locations or processes with higher-order communication to be adaptable—static adaptation.

- adLoc *Let's consider the case* $h = l!R$ *for some location* l *and process* R *- analogous to the previous case as locations can perform higher-order outputs as described in the constraint* static adaptation.
- adRcv *where* $l\langle P\rangle \xrightarrow{l?R} l\langle R\rangle$*, We assume that* $\emptyset \vdash R$ *and then result follows from* wLoc *and eq.* (IB.37).

case $\dfrac{l \in \Gamma \quad \emptyset \vdash P \quad \Gamma \vdash Q \quad in(Q) = \emptyset}{\Gamma \vdash l!P.Q}$ (wSnd) *We know that*

$$l \in \Gamma \tag{IB.44}$$
$$\emptyset \vdash P \tag{IB.45}$$
$$\Gamma \vdash Q \tag{IB.46}$$
$$in(Q) = \emptyset \tag{IB.47}$$

By case-analysis on the $l!P.Q \xrightarrow{l!P} Q$*. The only applicable rule in* AdSnd*. Result follows fromeqs.* (IB.44) *to* (IB.46)

case $\dfrac{i \in \mathcal{I} \text{ implies } \Gamma \vdash P_i \text{ and } in(P_i) = \emptyset}{\Gamma \vdash \square_{i\in\mathcal{I}} e_i \to P_i}$ (wChx), $\dfrac{\Gamma \vdash P \quad in(P) = \emptyset}{\Gamma \vdash recX(\vec{y} := \vec{e}).P}$ (wRec), $\dfrac{\Gamma \vdash P \quad \Gamma \vdash Q \quad in(P) = \emptyset \quad in(Q) = \emptyset}{\Gamma \vdash if e_1 \le e_2 \text{ then } P \text{ else } Q}$ (wIf)

These cases are analogous. Here we only show the case for wChx*. By case-analysis on* $\square_{i\in\mathcal{I}} e_i \to P_i \xrightarrow{e} P'$*, only* evCh *can be applied such that there exists a* $j \in \mathcal{I}$*, where*

$$\square_{i\in\mathcal{I}} e_i \to P_i \xrightarrow{e_j} P_j \tag{IB.48}$$

$\Gamma \vdash P_j$ *follows from the premises.*

case $\dfrac{\Gamma \vdash M}{\Gamma \vdash (\nu e)M}$ (wScp)

From the structure of $(\nu\, e)M$*, there are three sub-cases*

- EvHide *where* $M \xrightarrow{e} M'$*. By IH and the premise* $\Gamma \vdash M$*, we infer that*

$$\Gamma \vdash M' \tag{IB.49}$$

Result follows from wScp

- EvEsc *analogous to the previous case.*
- AdEsc $M \xrightarrow{h} M'$ *such that* $c \notin h$*. By IH, we know that depending on the structure of* h
  - $P \xrightarrow{l!R} P'$ *implies* $\Gamma \vdash P'$ *and* $\emptyset \vdash R$ *and* $l \in \Gamma$
  - $P \xrightarrow{l?R} P'$ *and* $\emptyset \vdash R$ *implies* $\Gamma \vdash P'$ *and* $l \in \Gamma$

Result follows by IH and wScp

case $\Gamma \vdash SKIP, \Gamma \vdash X(\vec{e})$ (wSkp, wApp) *are vacuously true because these processes do not reduce.*

□

Lemma B.0.2 (Bisimulation reduction)**.** *For a well-formed process* $\Gamma \vdash P$ *and CSP process* S *such that* $P \rhd S$ *and* $P \xrightarrow{\alpha} P'$ *implies* $P' \rhd S'$ *and*

- $\alpha = e$ *implies* $S \xrightarrow{e} S'$
- $\alpha = l!R$ *implies* $S \xrightarrow{e} S'$ *and* $e = m(h)$
- $\alpha = l?R$ *and* $\emptyset \vdash R$ *implies* $S \xrightarrow{e} S'$ *and* $e = m(h)$

Proof. Proven by rule induction on $M \rhd S$

case $\dfrac{M \rhd S \quad N \rhd T \quad A = \{m(l!R) \mid R \in Proc, l \in L\}}{(\nu\, L)M \underset{E}{\parallel} N \rhd (S \underset{E,A}{\parallel} T) \setminus A}$ tPar

From tPar*, we know that*

$$M \rhd S \tag{IB.1}$$
$$N \rhd T \tag{IB.2}$$
$$A = \{m(l!R) \mid l \in L, R \in Proc\} \tag{IB.3}$$

From $\Gamma \vdash (\nu\, L)M \underset{E}{\parallel} N$*, we also know*

$$\Gamma \uplus L \vdash M \tag{IB.4}$$
$$\Gamma \uplus L \vdash N \tag{IB.5}$$
$$in(M) \cap out(N) \subseteq L \tag{IB.6}$$
$$in(N) \cap out(M) \subseteq L \tag{IB.7}$$
$$out(M) \cap out(N) \subseteq \Gamma \tag{IB.8}$$
$$in(M) \cap in(N) = \emptyset \tag{IB.9}$$

By case-analysis on $(\nu\, L)M \underset{E}{\parallel} N \xrightarrow{\alpha} T$ *(omitting symmetric cases)*

- evEsc *where we also know*

$$M \underset{E}{\parallel} N \xrightarrow{\alpha} T' \tag{IB.10}$$
$$T = (\nu\, L)T' \tag{IB.11}$$
$$\alpha \notin L \tag{IB.12}$$

By case-analysis on $M \underset{E}{\parallel} N \xrightarrow{\alpha} T'$

  - AdSyncL *where we know that*

$$M \xrightarrow{l!R} M' \tag{IB.13}$$
$$N \xrightarrow{l?R} N' \tag{IB.14}$$
$$\alpha = \tau \tag{IB.15}$$

By IH with eqs. (IB.1), (IB.4) *and* (IB.13) *and eqs.* (IB.2), (IB.5) *and* (IB.14)

$$S \xrightarrow{e} S' \tag{IB.16}$$
$$M' \rhd S' \tag{IB.17}$$
$$e = m(l!R) \quad \text{and } e' = m(l?R) \tag{IB.18}$$
$$N' \rhd T' \tag{IB.19}$$
$$T \xrightarrow{e'} T' \tag{IB.20}$$

But from the definition of m*, we know that* $e = e'$ *and from Lemma B.0.7 and B.0.8*

$$l \in out(M) \quad \text{and} \quad l \in in(N) \tag{IB.21}$$

which means that $l \in L$ *and* $e \in A$*, which by* CSync

$$S \underset{E,A}{\parallel} T \xrightarrow{e} S' \underset{E,A}{\parallel} T' \tag{IB.22}$$
$$(S \underset{E,A}{\parallel} T) \setminus A \xrightarrow{\tau} (S' \underset{E,A}{\parallel} T') \setminus A \tag{IB.23}$$

Using tPar*, we can infer* $(\nu\, L)M' \underset{E}{\parallel} N' \rhd (S' \underset{E,A}{\parallel} T') \setminus A$

– EvParL *where we know that*

$$M \xrightarrow{\alpha} M' \tag{IB.24}$$
$$\alpha \notin E \tag{IB.25}$$
$$\alpha \in \Sigma \tag{IB.26}$$

By IH with eqs. (IB.1), (IB.4) *and* (IB.24)

$$S \xrightarrow{e} S' \tag{IB.27}$$
$$M' \rhd S' \tag{IB.28}$$

By evParL*, we infer*

$$S \underset{E}{\parallel} T \xrightarrow{e} S' \underset{E}{\parallel} T \tag{IB.29}$$

and by tPar *with eqs.* (IB.2), (IB.3) *and* (IB.28)

$$(\nu\, L)M' \underset{E}{\parallel} N \rhd \left(S' \underset{E,A}{\parallel} T \right) \setminus A \tag{IB.30}$$

– EvSync *analogous to the previous cases.*

- AdEsc*, which means*

$$M \underset{E}{\parallel} N \xrightarrow{h} T' \tag{IB.31}$$
$$T = (\nu\, L)T' \tag{IB.32}$$
$$h \notin L \tag{IB.33}$$

By case-analysis on $M \underset{E}{\parallel} N \xrightarrow{h} T'$ *only* AdParL *is applicable. where we know that*

$$M \xrightarrow{h} M' \tag{IB.34}$$

By IH with eqs. (IB.1), (IB.4) *and* (IB.34)

$$S \xrightarrow{e} S' \tag{IB.35}$$
$$e = m(h) \tag{IB.36}$$
$$M' \rhd S' \tag{IB.37}$$

By tPar *by eqs.* (IB.2), (IB.3) *and* (IB.37)

$$(\nu\, L)M' \underset{E}{\parallel} N \rhd S' \underset{E,A}{\parallel} T \setminus A \tag{IB.38}$$

We know that the co-domain of m *is disjoint from* Σ *and hence* $e \notin E$*. We also know that* $h \notin L$ *and this means that* $e \notin A$*. By* CPar

$$S \underset{E,A}{\parallel} T \xrightarrow{e} S' \underset{E,A}{\parallel} T' \tag{IB.39}$$
$$(S \underset{E,A}{\parallel} T) \setminus A \xrightarrow{e} (S' \underset{E,A}{\parallel} T') \setminus A \tag{IB.40}$$

case $\dfrac{P \rhd S}{recX(\vec{y} := \vec{e}).P \rhd let\, X(\vec{y}) = S\ within\, X(\vec{e})}$ tRec

We know that

$$P \rhd S \tag{IB.41}$$

We infer that the only reduction rule that applies from fig. 4.3 is Rec

$$recX(\vec{y} := \vec{e}).P \xrightarrow{\tau} P[\vec{e}, (recX(\vec{y} := \vec{e}).P)/\vec{y}, X] \tag{IB.42}$$

The translated process

$$let\, X(\vec{y}) = S\ within\, X(\vec{e}) \xrightarrow{\tau} S[\![\vec{e}/\vec{y}]\!][\![^{let\, X(\vec{y}) = S\ within\, X(\vec{e})}/_X]\!] \tag{IB.43}$$

Result follows by Lem. B.0.12.

case $\dfrac{i \in \mathcal{I}\ implies\ P_i \rhd S_i}{\square_{i\in\mathcal{I}}\, e_i \to P_i \rhd \square_{i\in\mathcal{I}}\, e_i \to S_i}$ tChx

From the rule, we know that

$$i \in \mathcal{I}\ implies\ P_i \rhd S_i \tag{IB.44}$$

From the structure, we know that the only reduction rule that applies is through EvCh*, i.e.,*

there is a $j \in \mathcal{I}$

$$\Box_{i\in\mathcal{I}}\, e_i \to P_i \xrightarrow{e_j} P_j \tag{IB.45}$$

From tChx*'s premises, we know that* $P_j \rhd S_j$*, we can also construct the reduction*

$$\Box_{i\in\mathcal{I}}\, e_i \to S_i \xrightarrow{e_j} S_j \tag{IB.46}$$

case $\dfrac{P \rhd S \quad Q \rhd T}{if e_1 \le e_2 \text{ then } P \text{ else } Q \rhd if e_1 \le e_2 \text{ then } S \text{ else } T}$ tIf

From the rule we know

$$P \rhd S \tag{IB.47}$$

$$Q \rhd T \tag{IB.48}$$

From the structure, we know the only reduction rule that applies is through IfTrue *or* IfFalse *depending on* b*. Here we show the case when* b *evaluates to true i.e., the reduction* IfTrue *is applied, the other case is similar. This allow us to construct the CSP reduction*

$$if e_1 \le e_2 \text{ then } S \text{ else } T \xrightarrow{\tau} S \tag{IB.49}$$

Result follows from eqs. (IB.47) *and* (IB.49)*.*

case $\dfrac{P \rhd S}{l\langle P\rangle \rhd S \triangle rec(l)}$ tLoc

From the rule we know

$$P \rhd S \tag{IB.50}$$

$$rec(l) = \Box_{R\in Proc}\; m(l!R) \to (T_R \triangle rec(l) \text{ where } R \rhd T_R) \tag{IB.51}$$

We also know that $\Gamma \vdash l\langle P\rangle$ *which means*

$$\emptyset \vdash P \tag{IB.52}$$

$$l \in \Gamma \tag{IB.53}$$

By case-analysis on $l\langle P\rangle \xrightarrow{\alpha} T$*, we know that there are three applicable rules*

- AdRcv *where we know*

$$l\langle P\rangle \xrightarrow{l?R} l\langle R\rangle \tag{IB.54}$$

$$T = l\langle R\rangle \quad \text{and } \alpha = l?R \tag{IB.55}$$

From the lemma definition, we assume that $\emptyset \vdash R$ *which by theorem 6.1.14, we know there exists a* T_R *such that* $R \rhd T_R$*. Hence, there is an event* $e = m(l!R)$*. This allows us to*

construct the reduction by CInt

$$S \triangle rec(l) \xrightarrow{e} T_R \triangle rec(l) \tag{IB.56}$$

Result follows from tLoc *with* $R \rhd T_R$

- AdLoc *where we know*

$$P \xrightarrow{h} P' \tag{IB.57}$$
$$T = l\langle P' \rangle \tag{IB.58}$$

By IH with eqs. (IB.50), (IB.52) *and* (IB.57), *we infer*

$$S \xrightarrow{e} S' \tag{IB.59}$$
$$P' \rhd S' \tag{IB.60}$$
$$e = m(h) \tag{IB.61}$$

Result follows follows from tLoc.

- EvLoc—*analogous to the previous case.*

case $\dfrac{m(l!P) = e \quad Q \rhd S}{l!P.Q \rhd e \to S}$ tSnd

From rule, we know that

$$m(l!P) = e \tag{IB.62}$$
$$Q \rhd S \tag{IB.63}$$

By case-analysis on $l!P.Q \xrightarrow{\alpha} T$, *we know only the reduction rule* AdSnd *is applicable,*

$$l!P.Q \xrightarrow{l!P} Q \tag{IB.64}$$

The translated process can only resolve the prefix operation,

$$e \to S \xrightarrow{e} S \tag{IB.65}$$

Result follows from eqs. (IB.63) *and* (IB.65).

case $\dfrac{M \rhd S}{(\nu e)M \rhd S \setminus \{e\}}$ tScp

From rule we know that

$$M \rhd S \tag{IB.66}$$

From $\Gamma \vdash (\nu\, e)M$ *and* wScp, *we know*

$$\Gamma \vdash M \tag{IB.67}$$

By case-analysis $(\nu\, e)M \xrightarrow{\alpha} M'$, *there are three sub-cases*

- *EvHide where we know*

$$M \xrightarrow{e} M' \tag{IB.68}$$

By IH with eqs. (IB.66) *to* (IB.68)*, there exists* S' *such that*

$$S \xrightarrow{e} S' \tag{IB.69}$$
$$M' \rhd S' \tag{IB.70}$$

From eq. (IB.69)*, we construct the reduction* $S \setminus \{e\} \xrightarrow{\tau} S' \setminus \{e\}$ *and from eq.* (IB.70) *with* tScp *we infer*

$$(\nu\, e)M' \rhd S' \setminus \{e\}$$

- *EvEsc where we know*

$$M \xrightarrow{e'} M' \tag{IB.71}$$
$$e \neq e' \tag{IB.72}$$

By IH with eqs. (IB.66)*,* (IB.67) *and* (IB.71)*, there exists S*"

$$S \xrightarrow{e'} S' \tag{IB.73}$$
$$M' \rhd S' \tag{IB.74}$$

We can use tScp*, to infer*

$$(\nu\, e)M' \rhd S' \setminus \{e\}$$

- *AdEsc analogous to the prevoius case.*

	tApp	tSkp
case	$X(\vec{e}) \rhd X(\vec{e})$	*SKIP* $\rhd$ *SKIP are vacuously true because these processes do not reduce.*

□

Lemma B.0.3 (Bisimulation reduction)**.** *For all* $\Gamma \vdash M$*, CSP processes* S, S' *and CSP events* e *such that* $S \xrightarrow{e} S'$ *and* $M \rhd S$ *implies there exists* M' *such that* $M' \rhd S'$ *and*

- $e \in \Sigma$ *or* $e = \tau$ *implies* $M \xrightarrow{e} M'$
- $e = m(h)$ *implies* $M \xrightarrow{h} M'$

Proof. Proven by rule induction on $M \rhd S$

case tChx
$$\frac{i \in \mathcal{I} \text{ implies } P_i \rhd S_i}{\Box_{i \in \mathcal{I}}\, e_i \to P_i \rhd \Box_{i \in \mathcal{I}}\, e_i \to S_i}$$

From tChx *we know*

$$M = \Box_{i \in \mathcal{I}}\, e_i \to P_i \tag{IB.75}$$

$$S = \Box_{i \in \mathcal{I}}\, e_i \to S_i \tag{IB.76}$$

$$i \in \mathcal{I} \text{ implies } P_i \rhd S_i \tag{IB.77}$$

From the structures of both S *and* M*, we know that the only reduction rule that applies is to resolve the external choice, i.e.,*

$$\Box_{i \in \mathcal{I}}\, e_i \to S_i \xrightarrow{e_j} S_j \tag{IB.78}$$

for a $j \in \mathcal{I}$*, which allow us to construct the reduction*

$$\Box_{i \in \mathcal{I}}\, e_i \to P_i \xrightarrow{e_j} P_j \tag{IB.79}$$

Result $P_j \rhd S_j$ *follows from eq.* (IB.77)*.*

case $\dfrac{M \rhd S}{(\nu e)M \rhd S \setminus \{e\}}$ TSCP

From tScp*, we know*

$$M \rhd S \tag{IB.80}$$

From $\Gamma \vdash (\nu\, e)M$ *and* wScp*, we also know*

$$\Gamma \vdash M \tag{IB.81}$$

By case-analysis on $S \setminus \{e\} \xrightarrow{e'} S' \setminus \{e\}$

- CHid*, where we know*

$$S \xrightarrow{e} S' \quad \text{and} \quad e' = \tau \tag{IB.82}$$

By IH with eqs. (IB.80) *to* (IB.82)

$$M \xrightarrow{e} M' \tag{IB.83}$$

$$M' \rhd S' \tag{IB.84}$$

By evHide*, we construct the reduction* $(\nu\, e)M \xrightarrow{\tau} (\nu\, e)M'$*. By* tScp *with eq.* (IB.84)*, we infer* $(\nu\, e)M' \rhd S' \setminus \{e\}$

- CEsc *where we know*

$$S \xrightarrow{e'} S' \quad \text{and } e' \neq e \tag{IB.85}$$

By IH with eqs. (IB.80) *to* (IB.82)

$$M \xrightarrow{\alpha} M' \tag{IB.86}$$
$$M' \rhd S' \tag{IB.87}$$
$$m(\alpha) = e' \tag{IB.88}$$

By evEsc *or* adEsc*, we construct the reduction* $(\nu\, e)M \xrightarrow{\alpha} (\nu\, e)M'$*. By* tScp *with eq.* (IB.87)*, we infer* $(\nu\, e)M' \rhd S' \setminus \{e\}$*.*

case $\dfrac{P \rhd S}{recX(\vec{y} := \vec{e}).P \rhd let\, X(\vec{y}) = S\, within\, X(\vec{e})}$ tRec

From $\Gamma \vdash recX(\vec{y} := \vec{e}).P$*,*

$$\tag{IB.89}$$
$$in(P) = \emptyset \tag{IB.90}$$
$$\Gamma \vdash P \tag{IB.91}$$

From $recX(\vec{y} := \vec{e}).P \rhd let\, X(\vec{y}) = S\, within\, X(\vec{e})$*, we know that*

$$let\, X(\vec{y}) = S\, within\, X(\vec{e}) \xrightarrow{\tau} S[\vec{e}/\vec{y}][^{let\, X(\vec{y}) = S\, within\, X(\vec{e})}/_X] \tag{IB.92}$$

By Rec*, we construct the reduction*

$$recX(\vec{y} := \vec{e}).P \xrightarrow{\tau} P\sigma \tag{IB.93}$$
$$\sigma = [\vec{e}/\vec{y}][^{(recX(\vec{y} := \vec{e}).P)}/_X] \tag{IB.94}$$

Result follows by eq. (IB.91) *and Pro. B.0.11.*

case $\dfrac{P \rhd S \quad Q \rhd T}{if e_1 \leq e_2\, then\, P\, else\, Q \rhd if e_1 \leq e_2\, then\, S\, else\, T}$ tIf

From tIf*, we know*

$$P \rhd S \tag{IB.95}$$
$$Q \rhd T \tag{IB.96}$$

From the structure of S *and* M*, we infer that the only applicable reduction rule is to resolve the if statement. Here we show the case where* $e_1 \leq e_2$ *where* $if e_1 \leq e_2\, then\, P\, else\, Q \xrightarrow{\tau} P$*, then by* ifTrue*, we can perform the reduction,*

$$if e_1 \leq e_2\, then\, S\, else\, T \xrightarrow{\tau} S \tag{IB.97}$$

Result follows from eq. (IB.95) *and* wIf*.*

case $\dfrac{m(l!P) = e \quad Q \rhd S}{l!P.Q \rhd e \to S}$ tSnd

From tSnd*, we know that*

$$m(l!P) = e \tag{IB.98}$$
$$Q \rhd S \tag{IB.99}$$

From the structure of both M *and* S *we know that the only reduction rule that applies is* CPrf *where* $e \to S \xrightarrow{e} S$*. With* AdSnd*, we can reduce to*

$$l!P.Q \xrightarrow{l!P} Q \tag{IB.100}$$

From wSnd*,* $l!P$ *implies*

$$l \in \Gamma \tag{IB.101}$$
$$\emptyset \vdash P \tag{IB.102}$$
$$\Gamma \vdash Q \tag{IB.103}$$
$$in(Q) = \emptyset \tag{IB.104}$$

Result follows from eqs. (IB.98) *to* (IB.100)*.*

case $\dfrac{\text{tLoc}\quad P \rhd S}{l\langle P\rangle \rhd S \triangle rec(l)}$

From tLoc*, we know that*

$$P \rhd S \tag{IB.105}$$
$$rec(l) = \square_{R \in Proc}\ m(l!R) \to (T_R \triangle rec(l) \text{ where } R \rhd T_R) \tag{IB.106}$$

From $\Gamma \vdash l\langle P\rangle$*, we know*

$$\emptyset \vdash P \tag{IB.107}$$

By case-analysis on $S \triangle rec(l) \xrightarrow{e} S'$*,*

- cInt *where* $rec(l) \xrightarrow{e} S''$*. From the structure of* $rec(l)$*, where* CChx *must have been applied. There is a process* $R \in Proc$ *such that*

$$e = m(l!R) = m(l?R) \tag{IB.108}$$
$$R \rhd T_R \tag{IB.109}$$
$$S'' = T_R \triangle rec(l) \tag{IB.110}$$

By tLoc *with eq.* (IB.109)

$$l\langle R\rangle \rhd T_R \triangle rec(l) \tag{IB.111}$$

By AdRcv*, we construct the reduction* $l\langle P\rangle \xrightarrow{l?R} l\langle R\rangle$ *as required.*

- *cRdc where* $S \xrightarrow{e} S'$. *By IH with eqs.* (IB.105) *and* (IB.107),

$$P \xrightarrow{\alpha} P' \tag{IB.112}$$
$$P' \rhd S' \tag{IB.113}$$
$$m(\alpha) = e \tag{IB.114}$$

Depending on the structure of α, *we use either* AdLoc *or* EvLoc

$$l\langle P\rangle \xrightarrow{\alpha} l\langle P'\rangle \tag{IB.115}$$

By tLoc *with eq.* (IB.113),

$$l\langle P'\rangle \rhd S' \triangle rec(l) \tag{IB.116}$$

Result follows from eqs. (IB.114) *to* (IB.116).

case $\dfrac{\text{tPar}\quad M \rhd S \quad N \rhd T \quad A = \{m(l!R) \mid R \in Proc, l \in L\}}{(\nu\, L)M \underset{E}{\parallel} N \rhd (S \underset{E,A}{\parallel} T) \setminus A}$

From tPar, *we know*

$$M \rhd S \tag{IB.117}$$
$$N \rhd T \tag{IB.118}$$
$$A = \{m(l!R) \mid R \in Proc, l \in L\} \tag{IB.119}$$

By wPar,

$$\Gamma \uplus L \vdash M \tag{IB.120}$$
$$\Gamma \uplus L \vdash N \tag{IB.121}$$
$$in(N) \cap in(M) = \emptyset \tag{IB.122}$$
$$out(M) \cap out(N) \subseteq \Gamma \tag{IB.123}$$
$$in(N) \cap out(M) \subseteq L \tag{IB.124}$$
$$in(M) \cap out(N) \subseteq L \tag{IB.125}$$

By case-analysis on $(S \underset{E,A}{\parallel} T) \setminus A \xrightarrow{e} P'$

- *cHid where*

$$S \underset{E,A}{\parallel} T \xrightarrow{e'} P' \tag{IB.126}$$
$$e' \in A \tag{IB.127}$$
$$e = \tau \tag{IB.128}$$

By case-analysis on $S \underset{E,A}{\parallel} T \xrightarrow{e'} P'$

Since we know that $e \in A$*, then only* cSync *applies*

$$S \xrightarrow{e'} S' \tag{IB.129}$$

$$T \xrightarrow{e'} T' \tag{IB.130}$$

$$e' \in A \tag{IB.131}$$

By IH with eqs. (IB.117), (IB.120) *and* (IB.129) *and eqs.* (IB.118), (IB.121) *and* (IB.130)

$$M \xrightarrow{\alpha} M' \tag{IB.132}$$

$$N \xrightarrow{\alpha'} N' \tag{IB.133}$$

$$M' \rhd S' \tag{IB.134}$$

$$N' \rhd T' \tag{IB.135}$$

By tPar *with eqs.* (IB.119), (IB.134) *and* (IB.135)*, we infer*

$$(\nu\, L)M' \underset{E}{\parallel} N' \rhd S' \underset{E,A}{\parallel} T' \setminus A \tag{IB.136}$$

By case-analysis on α

1. $\alpha = h$ *and* $\alpha' = h'$*. From the definition of* m*, we know that an unique location* l *and process* R *are mapped to* e*. However the events might be encoding higher-order prefixes or named processes, i.e.,* $m(l!R) = m(l?R) = e$*. By case-analysis*

   - $h = l!R$ *and* $h' = l!R$*: by Lem. B.0.7, we know that*

     $$l \in out(M) \tag{IB.137}$$

     From eq. (IB.123) *and* $l \in L$*, we know*

     $$l \notin out(N) \tag{IB.138}$$

   - $h = l?R$ *and* $h' = l?R$*: analogous to the previous case.*
   - $h = l!R$ *and* $h' = l?R$*: We use* AdSyncL *to infer*

     $$M \underset{E}{\parallel} N \xrightarrow{\tau} M' \underset{E}{\parallel} N' \tag{IB.139}$$

   - $h = l?R$ *and* $h' = l!R$*: analogous to the previous case.*

- cEsc *where*

$$S \underset{E,A}{\parallel} T \xrightarrow{e} P' \tag{IB.140}$$

$$e \notin A \tag{IB.141}$$

By case-analysis on $S \underset{E,A}{\parallel} T \xrightarrow{e} P'$

– *cPar where*

$$S \xrightarrow{e} S' \tag{IB.142}$$

$$e \notin A, E \tag{IB.143}$$

By IH with eqs. (IB.117), (IB.120) *and* (IB.142)

$$M \xrightarrow{\alpha} M' \tag{IB.144}$$

$$M' \rhd S' \tag{IB.145}$$

such that $\alpha \in \Sigma$ *implies* $\alpha = e$ *otherwise* $m(\alpha) = e$. *Depending on the structure of* α *we use* adParL *or* evParL *to infer*

$$M \underset{E}{\parallel} N \xrightarrow{\alpha} M' \underset{E}{\parallel} N \tag{IB.146}$$

By tPar *with eqs.* (IB.118), (IB.119) *and* (IB.145), *we infer*

$$(\nu\, L)M' \underset{E}{\parallel} N \rhd S' \underset{E,A}{\parallel} T \setminus A \tag{IB.147}$$

From eqs. (IB.119) *and* (IB.143), *we infer that* $\alpha \notin L$ *and hence by* adEsc *or* evEsc *depending on the structure of* α

$$(\nu\, L)M \underset{E}{\parallel} N \xrightarrow{\alpha} (\nu\, L)M' \underset{E}{\parallel} N \tag{IB.148}$$

– *cSync—analogous to the previous sub-case but since we know that* $e \notin A$, α *must be in* Σ *and* $e \in E$.

	tApp	tSkp
case	$X(\vec{e}) \rhd X(\vec{e})$	$SKIP \rhd SKIP$ *are vacuously true because these processes do not reduce.*

□

Lemma B.0.4 (Translation Deterministic)**.** *For a well-formed ACSP process* P, $P \rhd S_1$ *and* $P \rhd S_2$ *implies* $S_1 = S_2$.

Proof. Proven by rule induction on $P \rhd S_1$.

case $\dfrac{M \rhd S \quad N \rhd T \quad A = \{m(l!R) \mid R \in Proc, l \in L\}}{(\nu\, L)M \underset{E}{\parallel} N \rhd (S \underset{E,A}{\parallel} T) \setminus A}$ tPar *In this case, we know that*

$$P = (\nu\, L)M \underset{E}{\parallel} N \tag{IB.149}$$

$$S_1 = (S \underset{E,A}{\parallel} T) \setminus A \tag{IB.150}$$

$$M \rhd S \tag{IB.151}$$

$$N \rhd T \tag{IB.152}$$

$$A = \{m(l!R) \mid R \in Proc, l \in L\} \tag{IB.153}$$

From the structure of P, we know that tPar *is the only rule that can be applied and thus,*

$$S_2 = (S' \underset{E,A_2}{\parallel} T') \setminus A_2 \tag{IB.154}$$

$$M \rhd S' \tag{IB.155}$$

$$N \rhd T' \tag{IB.156}$$

$$A_2 = \{m(l!R) \mid R \in Proc, l \in L\} \tag{IB.157}$$

By IH the translations eqs. (IB.151), (IB.152), (IB.155) *and* (IB.156) *imply that*

$$S = S' \tag{IB.158}$$

$$T = T' \tag{IB.159}$$

From the set definition, we can assume that $A = A_2$. This means that $S_1 = S_2$

tLoc
$$\frac{P \rhd S}{l\langle P\rangle \rhd S \triangle rec(l)}$$
case *We know that*

$$P = l\langle Q\rangle \tag{IB.160}$$

$$S_1 = S \triangle rec(l) \tag{IB.161}$$

$$Q \rhd S \tag{IB.162}$$

From the structure of P, we know that tLoc *is the only applicable rule and thus*

$$S_2 = S' \triangle rec(l) \tag{IB.163}$$

$$Q \rhd S' \tag{IB.164}$$

By IH, we know that $S = S'$ and similarly the process $rec(l)$ is deterministic by IH. These allow us to infer that $S_1 = S_2$.

tSnd
$$\frac{m(l!P) = e \quad Q \rhd S}{l!P.Q \rhd e \to S}$$
case *We know that*

$$P = l!P.Q \tag{IB.165}$$

$$S_1 = e \to S \tag{IB.166}$$

$$m(l!P) = e \tag{IB.167}$$

$$Q \rhd S \tag{IB.168}$$

From the structure of P we infer that the only rule applicable is tSnd *and thus,*

$$S_2 = e' \to S' \tag{IB.169}$$

$$m(l!P) = e \tag{IB.170}$$

$$Q \rhd S' \tag{IB.171}$$

By IH and eqs. (IB.168) *and* (IB.171)*, we know that* $S = S'$*. We also know that the mapping* m *is deterministic and hence* $e = e'$*. This allows us to infer that* $S_1 = S_2$ *as required.*

tApp tSkp

case $X(\vec{e}) \rhd X(\vec{e}), SKIP \rhd SKIP$ *vacuously true.*

tRec tChx tScp tIf

case $\dfrac{P \rhd S}{recX(\vec{y} := \vec{e}).P \rhd let\, X(\vec{y}) = S\, within\, X(\vec{e})}, \dfrac{i \in \mathcal{I}\ implies\ P_i \rhd S_i}{\Box_{i \in \mathcal{I}} e_i \to P_i \rhd \Box_{i \in \mathcal{I}} e_i \to S_i}, \dfrac{M \rhd S}{(\nu e)M \rhd S \setminus \{e\}}, \dfrac{P \rhd S \quad Q \rhd T}{if e_1 \le e_2\, then\, P\, else\, Q \rhd if e_1 \le e_2\, then\, S\, else\, T}$

Here, we only show the case for tScp*. We know that*

$$P = (\nu\, e)M \tag{IB.172}$$

$$S_1 = S \setminus \{e\} \tag{IB.173}$$

$$M \rhd S \tag{IB.174}$$

From the structure of P*, we know that the only rule applicable is* tScp *and hence*

$$S_2 = S' \setminus \{e\} \tag{IB.175}$$

By IH we know that $S = S'$ *and hence we infer that* $S_1 = S_2$*.*

□

Properties of the Process Language

Lemma B.0.5. $\Gamma \vdash M$ *and* $M \xrightarrow{\alpha} M'$ *implies* $in(M) \subseteq in(M')$

Proof. Proven by rule induction on $M \xrightarrow{\alpha} M'$

□

Lemma B.0.6. $\Gamma \vdash M$ *and* $M \xrightarrow{\alpha} M'$ *then* $out(M') \subseteq out(M)$

Proof. Proven by rule induction on $M \xrightarrow{e} M'$

□

Lemma B.0.7. $\Gamma \vdash M$ *and* $M \xrightarrow{l!R} M'$ *implies* $l \in out(M)$

Proof. Proven by rule induction on $M \xrightarrow{l!R} M'$

□

Lemma B.0.8. $\Gamma \vdash M$ *and* $M \xrightarrow{l?R} M'$ *implies* $l \in in(M)$

Proof. Proven by rule induction on $M \xrightarrow{l!R} M'$

□

Proposition B.0.9. *For an ACSP process* P *and environment* Γ *such that* $\Gamma \vdash P$ *then*

$$l \in in(P)\ implies\ l \in \Gamma$$
$$l \in out(P)\ implies\ l \in \Gamma$$

Proof. Proven by rule induction on $\Gamma \vdash P$. □

Proposition B.0.10. *For an environment* $L \cap loc(P) = \emptyset$ *then* $\Gamma \vdash P$ *iff* $\Gamma \uplus L \vdash P$

Proposition B.0.11. *For all P, S and $\sigma : Var \rightharpoonup \bigcup\{Event, Var, Proc\}$ such that $\Gamma \vdash P$ implies $\Gamma \vdash P\sigma$*

Proof. Proven by rule induction on well-formed derivation. □

Lemma B.0.12. *For all P, S and $\sigma : Var \rightharpoonup \bigcup\{Event, Var, Proc\}$ such that $P \rhd S$ implies $P\sigma \rhd S\sigma$*

Proof. Proven by rule induction on $P \rhd S$,

case $\dfrac{\textsc{tPar}\quad M \rhd S \quad N \rhd T \quad A = \{m(l!R) \mid R \in Proc, l \in L\}}{(\nu\, L)M \underset{E}{\parallel} N \rhd (S \underset{E,A}{\parallel} T) \setminus A}$

We know that

$$M \rhd S \tag{IB.176}$$

$$N \rhd T \tag{IB.177}$$

$$A = \{m(l!R) \mid l \in L, R \in Proc\} \tag{IB.178}$$

By IH and eqs. (IB.176) *and* (IB.177)

$$M\sigma \rhd S\sigma \tag{IB.179}$$

$$N\sigma \rhd T\sigma \tag{IB.180}$$

We may infer that $(\nu\, L)M \parallel N\sigma = (\nu\, L)M\sigma \parallel N\sigma$. This means that from the structure only tPar *can be applied, where*

$$(\nu\, L)M\sigma \underset{E}{\parallel} N\sigma \rhd (S\sigma \underset{E,A}{\parallel} T\sigma) \setminus A \tag{IB.181}$$

Note that the set of events E, A are unaffected by the substitution as they do not contain free variables.

case $\dfrac{\textsc{tRec}\quad P \rhd S}{recX(\vec{y} := \vec{e}).P \rhd let\, X(\vec{y}) = S\ within\, X(\vec{e})}$

From IH we know that $P \rhd S$ implies that

$$P\sigma \rhd S\sigma \tag{IB.182}$$

We know that the substitution maps free variables and so $(recX(\vec{y} := \vec{e}).P)\sigma = recX(\vec{y} := \vec{e}).P\sigma$. Result follows from tRec.

case $\dfrac{\textsc{tLoc}\quad P \rhd S}{l\langle P\rangle \rhd S \triangle rec(l)}$

By IH, we know that $P\sigma \rhd S\sigma$. We also know that $l\langle P\rangle\sigma = l\langle P\sigma\rangle$. We also know that

$$rec(l) = \underset{e \in ch(l)}{\Box}\ e \rightarrow (T_e \triangle rec(l)) \tag{IB.183}$$

where any $e \in ch(l)$ translates to CSP process T_e by $p(e) \rhd T_e$.

case $\dfrac{m(l!P) = e \quad Q \rhd S}{l!P.Q \rhd e \to S}$ tSnd

case $\dfrac{i \in \mathcal{I} \text{ implies } P_i \rhd S_i}{\Box_{i\in\mathcal{I}} e_i \to P_i \rhd \Box_{i\in\mathcal{I}} e_i \to S_i}$ tChx, $\dfrac{P \rhd S \quad Q \rhd T}{if e_1 \le e_2 \text{ then } P \text{ else } Q \rhd if e_1 \le e_2 \text{ then } S \text{ else } T}$ tIf, $\dfrac{M \rhd S}{(\nu e)M \rhd S \setminus \{e\}}$ tScp

case $X(\vec{e}) \rhd X(\vec{e})$ tApp $SKIP \rhd SKIP$ tSkp *are vacuously true because these processes do not have free variables.*

□

Merge Operation

Lemma B.0.13. *For well-formed processes $S^{M_1}_{A_1}$ and $S^{M_2}_{A_2}$ then $S^{M_1}_{A_1} \otimes S^{M_2}_{A_2}$ is well-formed.*

Proof. We know that

$$S^{M_1}_{A_1} \otimes S^{M_2}_{A_2} = (\nu\, A_\otimes, M_\otimes)\left(\left(\Big\|_{l\in A_\otimes, M_\otimes} l\langle P_l\rangle\right) \|_{E_{A_\otimes}, E_{M_\otimes}} \Pi^{M_\otimes}_{A_\otimes}\right)$$

where

$$A_\otimes = (A_1, A_2) \qquad \text{and } M_\otimes = (M_1, M_2) \setminus A_\otimes \tag{IB.184}$$

$$\Pi^{M_\otimes}_{A_\otimes} = \Pi^{M_1}_{A_1} \; {}_{E_{A_1}, E_{M_1}}\|_{E_{A_2}, E_{M_2}} \; \Pi^{M_2}_{A_2} \tag{IB.185}$$

From the structure of $S^{M_1}_{A_1}$, we know that `wPar` must have been applied to infer well-formed, i.e.,

$$A_1, M_1 \vdash \Big\|_{l\in A_1, M_1} l\langle P_l\rangle \tag{IB.186}$$

$$A_1, M_1 \vdash \Pi^{M_1}_{A_1} \tag{IB.187}$$

$$in(\Pi^{M_1}_{A_1}) \cap in(\Big\|_{l\in A_1, M_1} l\langle P_l\rangle) = \emptyset \tag{IB.188}$$

$$out(\Pi^{M_1}_{A_1}) \cap out(\Big\|_{l\in A_1, M_1} l\langle P_l\rangle) \subseteq \emptyset \tag{IB.189}$$

$$out(\Pi^{M_1}_{A_1}) \cap in(\Big\|_{l\in A_1, M_1} l\langle P_l\rangle) \subseteq A_1, M_1 \tag{IB.190}$$

$$in(\Pi^{M_1}_{A_1}) \cap out(\Big\|_{l\in A_1, M_1} l\langle P_l\rangle) \subseteq A_1, M_1 \tag{IB.191}$$

Similarly from the structure of $S_{A_2}^{M_2}$,

$$A_2, M_2 \vdash \Big\|_{l \in A_2, M_2} l\langle P_l \rangle \tag{IB.192}$$

$$A_2, M_2 \vdash \Pi_{A_2}^{M_2} \tag{IB.193}$$

$$in(\Pi_{A_2}^{M_2}) \cap in(\Big\|_{l \in A_2, M_2} l\langle P_l \rangle) = \emptyset \tag{IB.194}$$

$$out(\Pi_{A_2}^{M_2}) \cap out(\Big\|_{l \in A_2, M_2} l\langle P_l \rangle) \subseteq \emptyset \tag{IB.195}$$

$$out(\Pi_{A_2}^{M_2}) \cap in(\Big\|_{l \in A_2, M_2} l\langle P_l \rangle) \subseteq A_2, M_2 \tag{IB.196}$$

$$in(\Pi_{A_2}^{M_2}) \cap out(\Big\|_{l \in A_2, M_2} l\langle P_l \rangle) \subseteq A_2, M_2 \tag{IB.197}$$

From the structure of processes in eqs. (IB.186) and (IB.192), we can infer that

$$out(\Big\|_{l \in A_1, M_1} l\langle P_l \rangle) = \emptyset \tag{IB.198}$$

$$out(\Big\|_{l \in A_2, M_2} l\langle P_l \rangle) = \emptyset \tag{IB.199}$$

$$in(\Big\|_{l \in A_1, M_1} l\langle P_l \rangle) = A_1, M_1 \tag{IB.200}$$

$$in(\Big\|_{l \in A_2, M_2} l\langle P_l \rangle) = A_2, M_2 \tag{IB.201}$$

From Pro. 5.1.5, we know that

$$in(\Pi_{A_1}^{M_1}) = \emptyset \tag{IB.202}$$
$$in(\Pi_{A_2}^{M_2}) = \emptyset \tag{IB.203}$$
$$out(\Pi_{A_1}^{M_1}) = \emptyset \tag{IB.204}$$
$$out(\Pi_{A_2}^{M_2}) = \emptyset \tag{IB.205}$$
$$\tag{IB.206}$$

From Pro. B.0.10, the well-formed is preserved if we include the locations not mentioned in each process such that

$$A_\otimes, M_\otimes \vdash \Big\|_{l \in A_1, M_1} l\langle P_l \rangle \tag{IB.207}$$

$$A_\otimes, M_\otimes \vdash \Big\|_{l \in A_2, M_2} l\langle P_l \rangle \tag{IB.208}$$

We split eq. (IB.208) into two parallel processes, those that are contained in (A_1, M_1)-$A_2'' = A_2 \cap (A_1, M_1)$, $M_2'' = M_2 \cap (A_1, M_1)$ and those that are not in (A_1, M_1)-$A_2' = A_2 \setminus (A_1, M_1)$ and

$M_2' = M_2 \setminus (A_1, M_1)$. This means that eq. (IB.184)

$$A_\otimes = A_1 \uplus A_2' \qquad \text{and} \qquad M_\otimes = M_1 \uplus M_2' \tag{IB.209}$$

Through wPar,

$$A_\otimes, M_\otimes \vdash \Big\|_{l \in A_2', M_2'} l\langle P_l \rangle \tag{IB.210}$$

$$A_\otimes, M_\otimes \vdash \Big\|_{l \in A_2'', M_2''} l\langle P_l \rangle \tag{IB.211}$$

$$in(\Big\|_{l \in A_2', M_2'} l\langle P_l \rangle) \cap in(\Big\|_{l \in A_2'', M_2''} l\langle P_l \rangle) = \emptyset \tag{IB.212}$$

$$out(\Big\|_{l \in A_2', M_2'} l\langle P_l \rangle) \cap out(\Big\|_{l \in A_2'', M_2''} l\langle P_l \rangle) \subseteq A_\otimes, M_\otimes \tag{IB.213}$$

$$out(\Big\|_{l \in A_2', M_2'} l\langle P_l \rangle) \cap in(\Big\|_{l \in A_2'', M_2''} l\langle P_l \rangle) \subseteq \emptyset \tag{IB.214}$$

$$in(\Big\|_{l \in A_2', M_2'} l\langle P_l \rangle) \cap out(\Big\|_{l \in A_2'', M_2''} l\langle P_l \rangle) \subseteq \emptyset \tag{IB.215}$$

From $A_2' \uplus A_1$, we know that

$$in(\Big\|_{l \in A_1, M_1} l\langle P_l \rangle) \cap in(\Big\|_{l \in A_2', M_2'} l\langle P_l \rangle) = \emptyset \tag{IB.216}$$

$$out(\Big\|_{l \in A_1, M_1} l\langle P_l \rangle) \cap out(\Big\|_{l \in A_2', M_2'} l\langle P_l \rangle) = \emptyset \tag{IB.217}$$

$$out(\Big\|_{l \in A_1, M_1} l\langle P_l \rangle) \cap in(\Big\|_{l \in A_2', M_2'} l\langle P_l \rangle) = \emptyset \tag{IB.218}$$

$$in(\Big\|_{l \in A_1, M_1} l\langle P_l \rangle) \cap out(\Big\|_{l \in A_2', M_2'} l\langle P_l \rangle) = \emptyset \tag{IB.219}$$

Then with `wPar` with eqs. (IB.207) and (IB.210)

$$A_\otimes, M_\otimes \vdash \Big\|_{l \in A_\otimes, M_\otimes} l\langle P_l \rangle \tag{IB.220}$$

From Pro. B.0.10 and eqs. (IB.187) and (IB.193), we add the missing (disjoint) names to the environment,

$$A_\otimes, M_\otimes \vdash \Pi_{A_1}^{M_1} \tag{IB.221}$$

$$A_\otimes, M_\otimes \vdash \Pi_{A_2}^{M_2} \tag{IB.222}$$

From eqs. (IB.202) to (IB.205), (IB.221) and (IB.222) and `wPar`

$$A_\otimes, M_\otimes \vdash \Pi_{A_1}^{M_1} \; {}_{E_{A_1}, E_{M_1}} \|_{E_{A_2}, E_{M_2}} \; \Pi_{A_2}^{M_2} \tag{IB.223}$$

From `wPar` and eqs. (IB.198), (IB.199), (IB.220) and (IB.223)

$$\emptyset \vdash (\nu A_{\otimes}, M_{\otimes}) \left(\Big\| _{l \in A_{\otimes}, M_{\otimes}} l\langle P_l \rangle \right) \|_{E_{A_{\otimes}}, E_{M_{\otimes}}} \left(\Pi^{M_1}_{A_1 \ E_{A_1}, E_{M_1}} \|_{E_{A_2}, E_{M_2}} \Pi^{M_2}_{A_2} \right) \tag{IB.224}$$

□

Lemma B.0.14. *For well-formed processes $S^{M_1}_{A_1}$ and $S^{M_2}_{A_2}$ such that $A_2 \cap (A_1, M_1) = \emptyset$*

$$S^{M_1}_{A_1} \otimes S^{M_2}_{A_2} = S^{M_1}_{A_1} \|_{(E_{A_1}, E_{M_1}) \cap (E_{A_2}, E_{M_2})} S^{M_2 \setminus (A_1, M_1)}_{A_2}$$

Proof. From Lem. B.0.13, we know that $S^{M_1}_{A_1} \otimes S^{M_2}_{A_2}$ is also well-formed, therefore by theorem 6.1.14 there exists CSP processes P_1, P_2 and P, such that

$$S^{M_1}_{A_1} \rhd P_1 \tag{IB.225}$$
$$S^{M_2}_{A_2} \rhd P_2 \tag{IB.226}$$
$$S^{M_1}_{A_1} \otimes S^{M_2}_{A_2} \rhd P \tag{IB.227}$$

From the structure of $S^{M_1}_{A_1}$, we know that `tPar` was used to derive eq. (IB.225)

$$E'_1 = \{e \mid m(l!P), P \in Proc, l \in A_1, M_1\} \tag{IB.228}$$
$$\left(\Big\| _{l \in A_1, M_1} l\langle P_l \rangle \right) \rhd [\![BS_1]\!] \tag{IB.229}$$
$$\Pi^{M_1}_{A_1} \rhd [\![\Pi_1]\!] \tag{IB.230}$$
$$P_1 = \left([\![BS_1]\!] \|_{E_{A_1}, E_{M_1}, E'_1} [\![\Pi_1]\!] \right) \setminus E'_1 \tag{IB.231}$$

From the structure of $S^{M_2}_{A_2}$, we know that `tPar` was used to derive eq. (IB.226)

$$E'_2 = \{e \mid m(l!P), P \in Proc, l \in A_2, M_2\} \tag{IB.232}$$
$$\left(\Big\| _{l \in A_2, M_2} l\langle P_l \rangle \right) \rhd [\![BS_2]\!] \tag{IB.233}$$
$$\Pi^{M_2}_{A_2} \rhd [\![\Pi_2]\!] \tag{IB.234}$$
$$P_2 = \left([\![BS_2]\!] \|_{E_{A_2}, E_{M_2}, E'_2} [\![\Pi_2]\!] \right) \setminus E'_2 \tag{IB.235}$$

From the structure of $S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$, we know that `tPar` to derive eq. (IB.227)

$$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} = (\nu\, A_\otimes, M_\otimes)\left(\left(\Big\|_{l \in A_\otimes, M_\otimes} l\langle P_l\rangle\right) \|_{E_{A_\otimes}, E_{M_\otimes}} \left(\Pi_{A_1}^{M_1} \|_{E_{A_1}, E_{M_1} \cap E_{A_2}, E_{M_2}} \Pi_{A_2}^{M_2}\right)\right) \tag{IB.236}$$

$$A_\otimes = A_1 \uplus A_2 \qquad \text{and} \qquad M_\otimes = (M_1, M_2) \setminus (A_1, A_2) \tag{IB.237}$$

$$E' = \{e \mid m(l!P), P \in Proc, l \in A_\otimes, M_\otimes\} \tag{IB.238}$$

$$\left(\Big\|_{l \in A_\otimes, M_\otimes} l\langle P_l\rangle\right) \rhd [\![BS]\!] \tag{IB.239}$$

$$\left(\Pi_{A_1}^{M_1} \|_{E_{A_1}, E_{M_1} \cap E_{A_2}, E_{M_2}} \Pi_{A_2}^{M_2}\right) \rhd [\![\Pi]\!] \tag{IB.240}$$

$$P = \left([\![BS]\!] \|_{E_{A_\otimes}, E_{M_\otimes}, E'} [\![\Pi]\!]\right) \setminus E' \tag{IB.241}$$

We split the process BS_2 into two processes-the locations that intersect with (A_1, M_1)-$M_2'' = M_2 \cap (A_1, M_1)$ and the locations that do not: A_2 and $M_2' = M_2 \setminus (A_1, M_1)$. From eq. (IB.233),

$$\left(\Big\|_{l \in A_2, M_2'} l\langle P_l\rangle\right) \rhd [\![BS_2']\!] \tag{IB.242}$$

$$\left(\Big\|_{l \in M_2''} l\langle P_l\rangle\right) \rhd [\![BS_2'']\!] \tag{IB.243}$$

$$[\![BS_2]\!] = [\![BS_2']\!] \|_{E_{A_2}, E_{M_2} \cap E_{M_2''}} [\![BS_2'']\!] \tag{IB.244}$$

eq. (IB.237) can be restated as $M_\otimes = M_1 \uplus M_2 \setminus (A_1, M_1) = M_1 \uplus M_2'$. By eq. (IB.239) and Pro. 6.1.13

$$BS = \left(\Big\|_{l \in A_1, M_1} l\langle P_l\rangle\right)\left(\Big\|_{l \in A_2, M_2'} l\langle P_l\rangle\right) \tag{IB.245}$$

$$[\![BS]\!] = [\![BS_1]\!] \|_{E_{A_1}, E_{M_1} \cap EA_2, E_{M_2'}} [\![BS_2']\!] \tag{IB.246}$$

From eqs. (IB.240), (IB.241) and (IB.246), we infer the structure of P,

$$P = \left(\left([\![BS_1]\!]_{\,E_{A_1}, E_{M_1}} \|_{E_{A_2}, E_{M_2'}} [\![BS_2']\!]\right) \|_{E_{A_\otimes}, E_{M_\otimes}, E'} \left([\![\Pi_1]\!]_{\,E_{A_1}, E_{M_1}} \|_{E_{A_2}, E_{M_2}} [\![\Pi_2]\!]\right)\right) \setminus E'$$

From the definition of ${}_A\|_B$,

$$P = \left(\left([\![BS_1]\!]_{\,E_{A_1}, E_{M_1}, E_1'} \|_{E_2', E_{A_2}, E_{M_2'}} [\![BS_2']\!]\right)_{\,E_{A_\otimes}, E_{M_\otimes}, E'} \|_{E_{A_\otimes}, E_{M_\otimes}, E'} \left([\![\Pi_1]\!]_{\,E_{A_1}, E_{M_1}, E_1'} \|_{E_2', E_{A_2}, E_{M_2'}} [\![\Pi_2]\!]\right)\right) \setminus E'$$

where $E' = E_1' \uplus E_{A_2}, E_{M_2}'$

From $\langle {}_A\|_B - \texttt{assoc} \rangle$

$$P = \left(\left(\left([\![BS_1]\!]_{\ E_{A_1}, E_{M_1}, E_1' \| E_{A_2}, E_{M_2}, E_2''} \ [\![BS_2']\!] \right) \ {\scriptstyle E_{A_2}, E_{M_2} \atop E_{A_1}, E_{M_1}}_{,E'} \| _{E_{A_1}, E_{M_1}, E_1'} \ [\![\Pi_1]\!] \right) \ {\scriptstyle E_{A_2}, E_{M_2} \atop E_{A_1}, E_{M_1}}_{,E'} \| _{E_{A_2}, E_{M_2}, E_2''} \ [\![\Pi_2]\!] \right) \setminus E'$$

From $\langle {}_A\|_B - \texttt{symm} \rangle$

$$P = \left(\left(\left([\![BS_2']\!]_{\ E_{A_2}, E_{M_2}, E_2''} \| _{E_{A_1}, E_{M_1}, E_1'} \ [\![BS_1]\!] \right)_{\ E_{A_\otimes}, E_{M_\otimes}, E'} \| _{E_{A_1}, E_{M_1}, E_1'} \ [\![\Pi_1]\!] \right)_{\ E_{A_\otimes}, E_{M_\otimes}, E'} \| _{E_{A_2}, E_{M_2}, E_2''} \ [\![\Pi_2]\!] \right) \setminus E'$$

From $\langle {}_A\|_B - \texttt{assoc} \rangle$

$$P = \left(\left([\![BS_2']\!]_{\ E_{A_2}, E_{M_2}, E_2''} \| _{E_{A_1}, E_{M_1}, E_1'} \left([\![BS_1]\!]_{\ E_{A_1}, E_{M_1}, E_1'} \| _{E_{A_1}, E_{M_1}, E_1'} \ [\![\Pi_1]\!] \right) \right)_{\ E_{A_\otimes}, E_{M_\otimes}, E'} \| _{E_{A_2}, E_{M_2}, E_2''} \ [\![\Pi_2]\!] \right) \setminus E'$$

From $\langle {}_A\|_B - \texttt{symm} \rangle$

$$P = \left([\![\Pi_2]\!]_{\ E_{A_2}, E_{M_2}, E_2''} \| _{E_{A_\otimes}, E_{M_\otimes}, E'} \left([\![BS_2']\!]_{\ E_{A_2}, E_{M_2}, E_2''} \| _{E_{A_1}, E_{M_1}, E_1'} \left([\![BS_1]\!]_{\ E_{A_1}, E_{M_1}, E_1'} \| _{E_{A_1}, E_{M_1}, E_1'} \ [\![\Pi_1]\!] \right) \right) \right) \setminus E'$$

From $\langle {}_A\|_B - \texttt{assoc} \rangle$

$$P = \left(\left([\![\Pi_2]\!]_{\ E_{A_2}, E_{M_2}, E_2''} \| _{E_{A_2}, E_{M_2}, E_2''} \ [\![BS_2']\!] \right)_{\ E_{A_2}, E_{M_2}, E_2''} \| _{E_{A_1}, E_{M_1}, E_1'} \left([\![BS_1]\!]_{\ E_{A_1}, E_{M_1}, E_1'} \| _{E_{A_1}, E_{M_1}, E_1'} \ [\![\Pi_1]\!] \right) \right) \setminus E'$$

From $\langle {}_A\|_B - \texttt{symm} \rangle$

$$P = \left(\left([\![BS_1]\!]_{\ E_{A_1}, E_{M_1}, E_1'} \| _{E_{A_1}, E_{M_1}, E_1'} \ [\![\Pi_1]\!] \right)_{\ E_{A_1}, E_{M_1}, E_1'} \| _{E_{A_2}, E_{M_2}, E_2''} \left([\![\Pi_2]\!]_{\ E_{A_2}, E_{M_2}, E_2''} \| _{E_{A_2}, E_{M_2}, E_2''} \ [\![BS_2']\!] \right) \right) \setminus E'$$

From $\langle \texttt{hide} - {}_A\|_B - \texttt{dist} \rangle$

$$P = \left([\![BS_1]\!]_{\ E_{A_1}, E_{M_1}, E_1'} \| _{E_{A_1}, E_{M_1}, E_1'} \ [\![\Pi_1]\!] \right) \setminus (E_1', E_{A_1}, E_{M_1} \cap E')_{\ E_{A_1}, E_{M_1}, E_1'} \| _{E_{A_2}, E_{M_2}, E_2''} \left([\![\Pi_2]\!]_{\ E_{A_2}, E_{M_2}, A_2'} \| _{E_{A_2}, E_{M_2}, E_2''} \ [\![BS_2']\!] \right) \setminus (E' \cap E_2'', E_i' \dots E_j')$$

From the definition of ${}_A\|_B$

$$P = \left([\![BS_1]\!]_{\ E_{A_1}, E_{M_1}, E_1'} \| \ [\![\Pi_1]\!] \right) \setminus (E_1', E_{A_1}, E_{M_1} \cap E') \ \| _{E_{A_1}, E_{M_1} \cap E_{A_2}, E_{M_2}} \left([\![\Pi_2]\!]_{\ E_{A_2}, E_{M_2}, E_2''} \| \ [\![BS_2']\!] \right) \setminus (E' \cap E_2'', E_i' \dots E_j')$$

Simplifying the set intersection (knowing that the first-order events are disjoint from E')

$$P = \left([\![BS_1]\!]_{\ E_{A_1}, E_{M_1}, E_1'} \| \ [\![\Pi_1]\!] \right) \setminus E_1' \ \| _{E_{A_1}, E_{M_1}, E_1'} \left([\![\Pi_2]\!]_{\ E_{A_2}, E_{M_2}, E_2''} \| \ [\![BS_2']\!] \right) \setminus E_2''$$

This translate to

$$P \lhd (\nu\, A_1, M_1) \left(\left(\|_{\ l \in A_1, M_1} \ l_l \langle P_l \rangle \right) \|_{\ E_{A_1}, E_{M_1}} \ \Pi_{A_1}^{M_1} \right) \ \| _{E_{A_1}, E_{M_1} \cap E_{A_2}, E_{M_2}} \ (\nu\, A_2, M_2'') \left(\left(\|_{\ l \in A_2, M_2''} \ l_l \langle P_l \rangle \right) \|_{\ E_{A_2}, E_{M_2''}} \ \Pi_{A_2}^{M_2} \right)$$

as required □

Lemma B.0.15. *From Lem. B.0.14 and the compositionality theorem, we prove that the specifications labelled as follows in table 7.2 are not affected by the composition of the cluster.*

Compositionality case 0 $R_1 \sqsubseteq S_{A_1}^{M_1}$ *and* $R_2 \sqsubseteq S_{A_2}^{M_2}$ *such that* $(A_1, M_1) \cap (A_2, M_2) = \emptyset$ *implies*

$$R_1 \underset{E}{\|} S_{A_2}^{M_2} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} \quad \textit{and} \quad R_2 \underset{E}{\|} S_{A_2}^{M_2} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$$

Compositionality case 1 *$R_1 \sqsubseteq S_{A_1}^{M_1}$ and $R_2 \sqsubseteq S_{A_2}^{M_2}$ such that $A_1 \cap (A_2, M_2) = \emptyset$, $A_2 \cap (A_1, M_1) = \emptyset$ implies*

$$R_1 \underset{E}{\parallel} S_{A_2}^{M_2 \setminus M_1} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} \quad \text{and} \quad R_2 \underset{E}{\parallel} S_{A_1}^{M_1 \setminus M_2} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$$

Compositionality case 2 *$R_1 \sqsubseteq S_{A_1}^{M_1}$ and $R_2 \sqsubseteq S_{A_2}^{M_2}$ such that $A_1 \cap (A_2, M_2) = \emptyset$ implies*

$$R_2 \underset{E}{\parallel} S_{A_1}^{M_1 \setminus (M_2, A_2)} \sqsubseteq S_{A_1}^{M_1} \otimes S_{A_2}^{M_2}$$

where $E = (E_{A_2}, E_{M_2}) \cap (E_{A_1}, E_{M_1})$ is the set of events that connect the components in $S_{A_1}^{M_1}$ and $S_{A_2}^{M_2}$

Proof. **case** *Compositionality 0 We know that*

$$R_1 \sqsubseteq S_{A_1}^{M_1} \tag{IB.247}$$
$$R_2 \sqsubseteq S_{A_2}^{M_2} \tag{IB.248}$$
$$(A_1, M_1) \cap (A_2, M_2) = \emptyset \tag{IB.249}$$

From the compositionality theorem theorem 6.1.18, we infer

$$R_1 \underset{E}{\parallel} S_{A_2}^{M_2} \sqsubseteq S_{A_1}^{M_1} \underset{E}{\parallel} S_{A_2}^{M_2} \tag{IB.250}$$
$$R_2 \underset{E}{\parallel} S_{A_1}^{M_1} \sqsubseteq S_{A_1}^{M_1} \underset{E}{\parallel} S_{A_2}^{M_2} \tag{IB.251}$$
$$E = (E_{A_2}, E_{M_2}) \cap (E_{A_1}, E_{M_1}) \tag{IB.252}$$

From Lem. B.0.14, we know that

$$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} = S_{A_1}^{M_1} \underset{E}{\parallel} S_{A_2}^{M_2 \setminus (A_1, M_1)} \tag{IB.253}$$

However, we know that $(A_1, M_1) \cap (A_2, M_2) = \emptyset$ and thus

$$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} = S_{A_1}^{M_1} \underset{E}{\parallel} S_{A_2}^{M_2} \tag{IB.254}$$

case *Compositionality 1 We know that*

$$R_1 \sqsubseteq S_{A_1}^{M_1} \tag{IB.255}$$
$$R_2 \sqsubseteq S_{A_2}^{M_2} \tag{IB.256}$$
$$A_1 \cap (A_2, M_2) = \emptyset \tag{IB.257}$$
$$A_2 \cap (A_1, M_1) = \emptyset \tag{IB.258}$$

In this case, the components are not disjoint, they overlap over the monitored components. From Pro. B.0.16, we know that for the well-formed clusters, removing a monitored component preserves the well-formed property

$$S_{A_2}^{M_2 \setminus (M_1, A_1)} \text{ is well-formed} \tag{IB.259}$$
$$S_{A_1}^{M_1 \setminus (M_2, A_2)} \text{ is well-formed} \tag{IB.260}$$

From the compositionality theorem theorem 6.1.18, we infer

$$R_1 \underset{E}{\parallel} S_{A_2}^{M_2 \setminus (M_1, A_1)} \sqsubseteq S_{A_1}^{M_1} \underset{E}{\parallel} S_{A_2}^{M_2 \setminus (M_1, A_1)} \tag{IB.261}$$

$$R_2 \underset{E}{\parallel} S_{A_1}^{M_1 \setminus (M_2, A_2)} \sqsubseteq S_{A_1}^{M_1 \setminus (M_2, A_2)} \underset{E}{\parallel} S_{A_2}^{M_2} \tag{IB.262}$$

$$E = (E_{A_2}, E_{M_2}) \cap (E_{A_1}, E_{M_1}) \tag{IB.263}$$

We know that $A_1 \cap (A_2, M_2) = \emptyset$, therefore from Lem. B.0.14, we know that

$$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} = S_{A_1}^{M_1} \underset{E}{\parallel} S_{A_2}^{M_2 \setminus (M_1, A_1)} \tag{IB.264}$$

Similarly, we know that $A_2 \cap (A_1, M_1) = \emptyset$, therefore from Lem. B.0.14, we know that

$$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} = S_{A21}^{M_2} \underset{E}{\parallel} S_{A_1}^{M_1 \setminus (M_2, A_2)} \tag{IB.265}$$

case *Compositionality 2 We know that*

$$R_1 \sqsubseteq S_{A_1}^{M_1} \tag{IB.266}$$

$$R_2 \sqsubseteq S_{A_2}^{M_2} \tag{IB.267}$$

$$A_1 \cap (A_2, M_2) = \emptyset \tag{IB.268}$$

We know that A_2 overlaps with M_1. Because of this overlap, the satisfaction of R_1 is affected by the composition but the satisfaction of R_2 is not. From Pro. B.0.16, we know that for the well-formed clusters, removing a monitored component preserves the well-formed property.

$$S_{A_1}^{M_1 \setminus (M_2, A_2)} \textit{ is well-formed} \tag{IB.269}$$

From the compositionality theorem theorem 6.1.18, we infer

$$R_2 \underset{E}{\parallel} S_{A_1}^{M_1 \setminus (M_2, A_2)} \sqsubseteq S_{A_2}^{M_2} \underset{E}{\parallel} S_{A_1}^{M_1 \setminus (M_2, A_2)} \tag{IB.270}$$

$$E = (E_{A_2}, E_{M_2}) \cap (E_{A_1}, E_{M_1}) \tag{IB.271}$$

We know that $A_1 \cap (A_2, M_2) = \emptyset$, therefore from Lem. B.0.14, we know that

$$S_{A_1}^{M_1} \otimes S_{A_2}^{M_2} = S_{A_2}^{M_2} \underset{E}{\parallel} S_{A_1}^{M_1 \setminus (M_2, A_2)} \tag{IB.272}$$

□

Proposition B.0.16. *For a well-formed cluster S_A^M, we know that $S_A^{M \setminus l}$ is also well-formed*

Proof. We know that

$$S_A^M = (\nu\, A, M) \left(\left(\underset{l \in (A,M)}{\Big\|} l\langle P_l \rangle \right) \underset{E_A, E_M}{\|} \Pi_A^M \right)$$

From the well-formed rules in fig. 6.1 and the structure of the process, we infer that only wPar

must have been applied. This means that

$$A, M \vdash \left(\Big\| _{l \in (A,M)} \; l\langle P_l \rangle \right) \tag{IB.273}$$

$$A, M \vdash \Pi_A^M \tag{IB.274}$$

$$in\left(\Big\| _{l \in (A,M)} \; l\langle P_l \rangle \right) \cap in(\Pi_A^M) = \emptyset \tag{IB.275}$$

$$out\left(\Big\| _{l \in (A,M)} \; l\langle P_l \rangle \right) \cap out(\Pi_A^M) = \emptyset \tag{IB.276}$$

$$out\left(\Big\| _{l \in (A,M)} \; l\langle P_l \rangle \right) \cap in(\Pi_A^M) \subseteq A, M \tag{IB.277}$$

$$in\left(\Big\| _{l \in (A,M)} \; l\langle P_l \rangle \right) \cap out(\Pi_A^M) \subseteq A, M \tag{IB.278}$$

From Pro. 5.1.5 and the definition of a monitored component, we know that

$$l \notin loc(\Pi_A^M) \tag{IB.279}$$

$$out(\Pi_A^M) \subseteq A \tag{IB.280}$$

$$in(\Pi_A^M) = \emptyset \tag{IB.281}$$

which from Pro. B.0.10, we know that eq. (IB.274) implies

$$A, M - \{l\} \vdash \Pi_A^M \tag{IB.282}$$

We know that $\left(\Big\| _{l \in (A,M)} \; l\langle P_l \rangle \right)$ is equivalent $\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \Big\| _{E_l \cap (E_A, E_M)} \; l\langle P_l \rangle$ which by

eq. (IB.273), we know that tPar must have been applied and therefore

$$A, M \vdash \left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \tag{IB.283}$$

$$A, M \vdash l_l\langle P_l \rangle \tag{IB.284}$$

$$in\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \cap in(l_l\langle P_l \rangle) = \emptyset \tag{IB.285}$$

$$out\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \cap out(l_l\langle P_l \rangle) \subseteq A, M \tag{IB.286}$$

$$in\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \cap out(l_l\langle P_l \rangle) \subseteq \emptyset \tag{IB.287}$$

$$out\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \cap in(l_l\langle P_l \rangle) \subseteq \emptyset \tag{IB.288}$$

$$\tag{IB.289}$$

From the structure of the process in eq. (IB.283), we know that $l \notin \left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right)$ and thus by Pro. B.0.10,

$$A, M - \{l\} \vdash \left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \tag{IB.290}$$

From eqs. (IB.280) and (IB.281) We know that eqs. (IB.275) to (IB.278) are all preserved in the assertions below

$$in\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \cap in(\Pi_A^M) = \emptyset \tag{IB.291}$$

$$out\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \cap out(\Pi_A^M) = \emptyset \tag{IB.292}$$

$$out\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \cap in(\Pi_A^M) \subseteq A \uplus (M - \{l\}) \tag{IB.293}$$

$$in\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \cap out(\Pi_A^M) \subseteq A \uplus (M - \{l\}) \tag{IB.294}$$

By wPar with eqs. (IB.282) and (IB.283), we infer

$$\emptyset \vdash (\nu\, A, M - \{l\}) \left(\left(\Big\| _{l \in (A,M)-\{l\}} \; l\langle P_l \rangle \right) \Big\|_{E_A, E_{M-\{l\}}} \Pi_A^M \right) \tag{IB.295}$$

$$S_A^{M\setminus\{l\}} \text{ is well-formed} \tag{IB.296}$$

□

Lemma B.0.17. *We show that the following property holds for well-formed clusters S_1, S_2 and S_3,*

$$S_1 \otimes S_2 = S_2 \otimes S_1 \qquad \langle \otimes - \mathit{sym} \rangle$$
$$S_1 \otimes (S_2 \otimes S_3) = (S_1 \otimes S_2) \otimes S_3 \qquad \langle \otimes - \mathit{assoc} \rangle$$

Proof. **case** $S_1 \otimes S_2 = S_2 \otimes S_1$

We know that S_1 and S_2 are well-formed then from Lem. B.0.13 $S_1 \otimes S_2$ is well-formed. Thus from theorem 6.1.14, we know that there is a CSP process such that $S_1 \otimes S_2 \rhd Q$. From the structure of clusters, we know that

$$S_1 \otimes S_2 = (\nu\, l_\otimes) \left(\left(\Big|\Big|_{l \in l_\otimes} l\langle P_l \rangle \right) \underset{E_\otimes}{\|} (\Pi_1 \;{}_{E_1}\|_{E_2}\; \Pi_2) \right)$$

where $l_\otimes = (loc(S_1) \cup loc(S_2))$, $E_1 = ev(\Pi_1)$ and $E_2 = ev(\Pi_2)$.

From the structure of the process, we know that wPar *must have been applied which by Pro. 6.1.13, we know that Q then must be of the form*

$$\rhd \left([\![BS]\!] \underset{E_\otimes, L_\otimes}{\|} ([\![\Pi_1]\!] \;{}_{E_1}\|_{E_2}\; [\![\Pi_2]\!]) \right) \setminus L_\otimes$$

From $\langle {}_A\|_B - \mathit{symm} \rangle$

$$= \left([\![BS]\!] \underset{E_\otimes, L_\otimes}{\|} ([\![\Pi_2]\!] \;{}_{E_2}\|_{E_1}\; [\![\Pi_1]\!]) \right) \setminus L_\otimes$$

which from the translation, this is equivalent to

$$\lhd (\nu\, l_\otimes) \left(\left(\Big|\Big|_{l \in l_\otimes} l\langle P_l \rangle \right) \underset{E_\otimes}{\|} (\Pi_2 \;{}_{E_2}\|_{E_1}\; \Pi_1) \right)$$

$$= S_2 \otimes S_1$$

case $S_1 \otimes (S_2 \otimes S_3) = (S_1 \otimes S_2) \otimes S_3$ *We know that S_1, S_2 and S_3 are well-formed then $S_1 \otimes (S_2 \otimes S_3)$ is well-formed. Thus from theorem 6.1.14, we know that there is a CSP process such that $S_1 \otimes (S_2 \otimes S_3) \rhd Q$. From the structure of clusters, we know that*

$$S_1 \otimes (S_2 \otimes S_3) = (\nu\, l_\otimes) \left(\left(\Big|\Big|_{l \in l_\otimes} l\langle P_l \rangle \right) \underset{E_\otimes}{\|} (\Pi_1 \;{}_{E_1}\|_{E_2, E_3}\; (\Pi_2 \;{}_{E_2}\|_{E_3}\; \Pi_3)) \right)$$

where $l_\otimes = \bigcup loc(S_i)$, $E_i = ev(\Pi_i)$ for $i \in \{1, 2, 3\}$. We know that Q then must be of the form

$$\rhd \left([\![BS]\!] \underset{E_\otimes, L_\otimes}{\|} ([\![\Pi_1]\!] \;{}_{E_1}\|_{E_2, E_3}\; ([\![\Pi_2]\!] \;{}_{E_2}\|_{E_3}\; [\![\Pi_3]\!])) \right) \setminus L_\otimes$$

From $\langle_A \| _B - \texttt{assoc}\rangle$

$$= \left([\![BS]\!] \underset{E_\otimes, L_\otimes}{\|} (([\![\Pi_1]\!] \ {}_{E_1}\|_{E_2} \ [\![\Pi_2]\!]) \ {}_{E_1,E_2}\|_{E_3} \ [\![\Pi_3]\!]) \right) \setminus L_\otimes$$

which from the translation, this is equivalent to

$$\triangleleft(\nu \, l_\otimes) \left(\left(\underset{l \in l_\otimes}{\Big\|} l\langle P_l \rangle \right) \underset{E_\otimes}{\|} ((\Pi_1 \ {}_{E_1}\|_{E_2} \ \Pi_2) \ {}_{E_1,E_2}\|_{E_3} \ \Pi_3) \right)$$

$$=(S_1 \otimes S_2) \otimes S_3$$

□

Appendix C

Encoding for the Art Gallery Case-Study

Requirement 2: ***At most 10 visitors in Room D*** In section 1.2, we outline two approaches to satisfy Req. 2. In Ad. Proc. 2.1, we track the number of visitors entering and leaving *room D* and we lock the entrance to *room D* once the number of visitors in the room is 10. In Ad. Proc. 2.2, we also want to prevent some degree of tailgating in adjacent rooms by restricting the number of visitors in *rooms B, D* to 10 visitors.

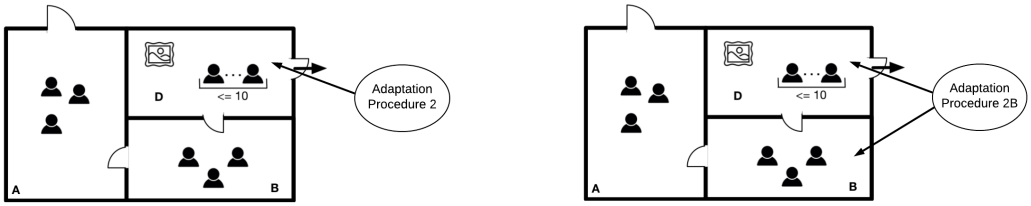
First Approach (Ad. Proc. 2.1) The components relevant to this approach is *room D*, which, as depicted in fig. 4.1, is physically connected to the *corridor* and *room B* through doors. We define a set of first-order events to encode these movement: $E_d = \{t_{(ED,c)}, t_{(EB,ED)}\}$. Here t can be vis, grd or emp to represent a visitor, guard or employees movements respectively. For simplicity we only consider vis in our presentation of the encoding. *Room D* can be in one of two functionalities:

1. The entrance to *room D* is open. From the topology, we know that there is no connectivity that allows visitors to move from *room D* to B and thus we do not include the event $vis_{(ED,EB)}$

$$R'_D = \square \begin{cases} vis_{(EB,ED)} \rightarrow R'_D \\ vis_{(ED,c)} \rightarrow R'_D \end{cases}$$

2. No more visitors are allowed to enter *room D* and thus visitors can only move from *room D* to the *corridor*,

$$R''_D = vis_{(ED,c)} \rightarrow R''_D$$



(a) First Approach described in (Ad. Proc. 2.1) (b) Second Approach described in (Ad. Proc. 2.2)

Figure C.1: Adaptation Procedures for enforcing Req. 2 at different levels of Granularities

The initial configuration of *room D* is encapsulated in location l_d, with the process R'_D—$l_d\langle R'_D\rangle$. An adaptation procedure can adapt the behaviour of *room D* through higher-order outputs to l_d.

The internal state of *Room D* tracks the number of visitors entering and leaving *room D* to make sure that the number of people leaving is less than or equal to the number of people entering *room D*,

$$C_D(n) = \Box \begin{cases} n > 0 \,\& & vis_{(ED,c)} \rightarrow C_D(n-1) \\ & vis_{(EB,ED)} \rightarrow C_D(n+1) \end{cases}$$

We now define an adaptation procedure Π_2 that aims to ensure the satisfaction of Req. 2. The adaptation procedure sends adaptation commands, in the form of higher-order communication to l_d to close or open the door to *room D*. In our encoding adaptation procedures comprise an adaptation pattern and an adaptation function. The pattern tracks the number of visitors in *room D* and triggers adaptation when the number reaches 9. To trigger adaptation, the process broadcasts an $\star$-event to the adaptation function. With an $\star$-event, we communicate the number of visitors in the rooms so the adaptation function can correctly determine the system behaviour that aims to ensure the satisfaction of Req. 2.

$$P_2(n) = \Box \begin{cases} vis_{(EB,ED)} \rightarrow & \text{if } n+1 \geq 10 \\ & \text{then } \star\langle n+1\rangle \rightarrow ack \rightarrow P_2(n+1) \\ & \text{else } P_2(n+1) \\ vis_{(ED,c)} \rightarrow & \text{if } n-1 \geq 9 \\ & \text{then } \star\langle n-1\rangle \rightarrow ack \rightarrow P_2(n-1) \\ & \text{else } P_2(n-1) \end{cases}$$

The encoding of the adaptation function F_2 is a process that listens to the $\star$-events and adapts the behaviour at location l_d through higher-order outputs depending on the number of visitors in the room. Irrespective of whether adaptation is needed, the process broadcasts the *ack* event back to P_2 to unblock it,

$$F_2 = \star\langle n\rangle \rightarrow \text{if } n \geq 10 \text{ then } l_d!R''_D.ack \rightarrow F_2 \text{ else } l_d!R'_D.ack \rightarrow F_2$$

The adaptation procedure is encoded as

$$\Pi_2 = (\nu \{\!|\star|\!\}, ack)\left(P_2(0) \underset{\{\!|\star|\!\},ack}{\parallel} F_2\right)$$

The *unary cluster* for Req. 2 is encoded as the composition of *room D* and the adaptation procedure Π_2,

$$S_2 = (\nu\, l_d)\left(\left(l_d\langle R'_D\rangle \underset{E_d}{\parallel} C_D(0)\right) \underset{E_d}{\parallel} \Pi_2\right)$$

We later verify that the *unary cluster* S_2 is a trace-refinement of the specification, $Spec_2$ where

only 10 visitors are in *room D* at any time,

$$Spec_2 = \text{let}$$
$$T(n) = \square \begin{cases} vis_{(EB,ED)} \to T(n+1) & \text{if } n < 10 \\ vis_{(ED,c)} \to T(n-1) & \text{if } n > 0 \end{cases}$$
$$\text{within } T(0)$$

The second approach (Ad. Proc. 2.2) To satisfy this approach, we extend the scope of our adaptation procedure. Through the *connectivity* relation, we know that *room D* is connected through a door with *room B*. In this approach, we limit the number of visitors in *rooms B, D* to 10, to reduce tail-gating to *room D*.

To satisfy this requirement, we define an adaptation procedure Π_{2B} that controls the entrance to *rooms B*. From fig. 4.1, we know that *Room B* is connected through a door with *room A* and a one-way door to *room D*. The door between *rooms A* and *B* is the only entrance to *rooms B, D*. We define a family of first-order events for the movement from and to *room B*—$E_B = \{t_{(EB,ED)}, t_{(EA,EB)}, t_{(EB,EA)}\}$. The movement from and to *room D* is the set of events E_D from the first approach. Again, for simplicity, we only consider vis in our presentation.

1. The entrance to the *rooms* is open to visitors, by allowing the event $vis_{(EA,EB)}$ that represents a visitor movement from *room A* to *room B*

$$R'_B = \square \begin{cases} vis_{(EB,ED)} \to R'_B \\ vis_{(EB,EA)} \to R'_B \\ vis_{(EA,EB)} \to R'_B \end{cases}$$

2. No more visitors are allowed to enter the *rooms B*, by precluding the event $vis_{(EA,EB)}$,

$$R''_B = \square \begin{cases} vis_{(EB,ED)} \to R''_B \\ vis_{(EB,EA)} \to R''_B \end{cases}$$

We encapsulate the initial processes for the components of *room B, D* in locations $l_b\langle R'_B\rangle$, and $l_d\langle R'_B\rangle$ respectively. The adaptation procedure can adapt between the two functionalities through higher-order outputs to l_b and l_d. We define the internal state of *room D* as the process $C_D(0)$ presented in first approach and similarly the process $C_B(0)$ tracks the number of visitors in *room B*.

We now define the adaptation procedure Π_{2B}. We assume the class of events $\{\!|\star_{2B}|\!\} = \{\star_{2B}\langle n\rangle \mid n \geq 0\}$ to be the communication of the number of visitors in *rooms B, D* in Π_{2B} between the *adaptation pattern* and *adaptation function*. We encode the adaptation procedure as the parallel composition of the adaptation pattern and function synchronizing on the $\star_{2B}$-events and *ack* event.

$$\Pi_{2B} = (\nu \{\!|\star_{2B}|\!\}, ack) \left(P_{2B}(0) \underset{\{\!|\star_{2B}|\!\}, ack}{\|} F_{2B} \right)$$

The adaptation pattern in process $P_{2B}(0)$ tracks the number of visitors in *rooms B, D*. The process communicates through a distinguished $\star_{2B}$-events the state when the system *must*-adapt. For these

cases, we trigger adaptation once the total number of visitors in the rooms is 10. The adaptation pattern may trigger adaptation more frequently e.g., for all movements or when the count is equal to or greater than 8. The adaptation function can decide appropriately how to handle adaptation. For simplicity, here we trigger adaptation when the rooms are full and further entrance should be precluded.

$$P_{2B}(0) = \Box \begin{cases} vis_{(EA,EB)} \rightarrow \text{if}(n+1 \geq 10) \text{ then } \star_{2B}\langle 10\rangle \rightarrow ack \rightarrow P_{2B}(10) \text{ else } P_{2B}(n+1) \\ vis_{(EB,EA)} \rightarrow \text{if}(n \geq 10) \text{ then } \star_{2B}\langle n-1\rangle \rightarrow ack \rightarrow P_{2B}(n-1) \text{ else if } n > 0 \text{ then } P_{2B}(n-1) \text{ else } P_{2B}(0) \\ vis_{(ED,c)} \rightarrow \text{if } n \geq 10 \text{ then } \star_{2B}\langle n-1\rangle \rightarrow ack \rightarrow P_{2B}(n-1) \text{ else if } n > 0 \text{ then } P_{2B}(n-1) \text{ else } P_{2B}(0) \\ vis_{(EB,ED)} \rightarrow P_{2B}(n) \end{cases}$$

At every execution point, the pattern either accepts all first-order system events or triggers adaptation. The process F_{2B} represents the adaptation function. The process F_{2B} only adapts *room B* but monitors *room D* to determine adaptations. The behaviour of *room B* is determined according to the number of visitors in the rooms, which is communicated with the $\star_{2B}$-event,

$$F_{2B} = \Box_{n \in \mathcal{I}} \star_{2B}\langle n\rangle \rightarrow \text{if } n \geq 10 \text{ then } l_b!R''_B.ack \rightarrow F_{2B} \text{ else } l_b!R'_B.ack \rightarrow F_{2B}$$

The encoding of the adaptation function is stateless, mirroring adaptation functions in SAA and only affect the system through adaptation, i.e., does not exhibit any first-order events external to Π_{2B},

The *unary cluster* for the requirement is defined as,

$$S_{2B} = (\nu\, l_B, l_D)\left(\left(l_D\langle R'_D\rangle \underset{E_D}{\parallel} C_D(0)\right) \underset{E_{(B,D)}}{\parallel} \left(l_B\langle R'_B\rangle \underset{E_B}{\parallel} C_B(0)\right) \underset{E_B, E_D}{\parallel} \Pi_{2B}\right)$$

where $E_{(B,D)} = \{vis_{(EB,ED)}\}$ is the set of events representing the connectivity between *room B* and *room D*.

We need to update the specification $Spec_2$ to include the behaviour of *room B*. To be effective, specifications needs to mirror faithfully the requirements. Since the requirement does not constraint the behaviour of *room B*, we encode the most permissive behaviour of *room B*,

$$\begin{aligned} Spec'_2 = &\text{let} \\ &T(n) = \Box \begin{cases} n < 10 \,\&\, vis_{(EB,ED)} & \rightarrow T(n+1) \\ n > 0 \;\&\, vis_{(ED,c)} & \rightarrow T(n-1) \\ vis_{(EA,EB)} & \rightarrow T(n) \\ vis_{(EB,EA)} & \rightarrow T(n) \end{cases} \\ &\text{within } T(0) \end{aligned}$$

Requirement 4: ***No more than 30 visitors (in total) should be in the exhibition area at the same time*** In section 1.2, we outline two approaches to satisfy Req. 4. In the first approach, described Ad. Proc. 4.1, we implement a simple access control, where we define an adaptation procedure that controls the entrance to the *exhibition area*, the door connecting *corridor 1* to *room A*, to ensure the number of visitors in the exhibition area never exceeds 30. The adaptation procedure monitors the exits from the exhibition area to track the number of visitors in the area.

In the second approach (Ad. Proc. 4.2), we define an adaptation procedure that ensures at

most 15 visitors are in rooms A and B and we also define an adaptation procedure that ensures at most 15 visitors in rooms B and D. Here, we overview the encoding for both approaches.

The first approach (Ad. Proc. 4.1) We follow Steps 2 and 3 from the approach presented in section 4.1. First, we identify the components that affect the satisfaction of Req. 4. Visitors can access and exit the *exhibition area* through *rooms A* and *D*. From the containment relation, we know that the *exhibition area* also contains *room B*. This room connects *room A* and *D*. Therefore guided by the topology, we infer that the components that affected the satisfaction of this requirement are *rooms A, B* and *D*. We define the set of first-order events E_{ea} to represent the connectivity of the rooms within the *exhibition area* as

$$E_{ea} = \{t_{(c,EA)}, t_{(EA,c)}, t_{(EB,EA)}, t_{(EA,EB)}, t_{(EB,ED)}, t_{(ED,c)}\} = E_A, E_B, E_D$$

Here t can take the form vis or grd to represent visitors or guard movements respectively; for simplicity we only consider the value vis in our encoding. We only present the encoding of *room A*, the encoding for *room B* and *room D* follow the same structure and has been presented for the encoding of Req. 2 (second approach). To satisfy this requirement, the adaptation procedure adapts *room A* to control the entrance to the *exhibition area* but monitors the behaviour of *rooms B, D* to track the number of visitors in the *exhibition area*.

The component *room A*, that is adapted to satisfy Req. 4, can be in one of the following functionalities:

1. Visitors are allowed in the *exhibition area* by moving through the door connecting the *corridor* to *room A*

$$R'_A = \Box_{t \in \{EB,c\}} \begin{cases} vis_{(t,EA)} \rightarrow R'_A \\ vis_{(EA,t)} \rightarrow R'_A \end{cases}$$

2. Visitors are *not* allowed in the *exhibition area* and thus we preclude the movement from the *corridor* to *room A*

$$R''_A = \Box \begin{cases} vis_{(EB,EA)} \rightarrow R''_A \\ vis_{(EA,EB)} \rightarrow R''_A \\ vis_{(EA,c)} \rightarrow R''_A \end{cases}$$

We assume a process C_A to track the internal state of *room A* similar to the process C_D presented for Req. 2.

We define an adaptation procedure Π_4 that tracks the number of visitors in the *exhibition area* and refuses entry to more visitors when the number of visitors in the *exhibition area* is 30. The adaptation procedure re-opens the door after a visitor leaves the *exhibition area*. In our encoding, an adaptation procedure comprises an adaptation pattern P_4 that tracks the number of visitors in the *exhibition area* and triggers adaptation when the number of visitors reaches 30 and an adaptation function F_4 that identifies the execution points where a system may-adapt and the adaptation—whether visitors are allowed entrance from *corridor* to *room A*.

$$\Pi_4 = (\nu \ \{|\star|\}, ack) \left(P_4(0) \underset{\{|\star|\},ack}{\|} F_4 \right)$$

The adaptation pattern P_4 communicates the number of visitors in the *exhibition area* to the adaptation function, when the number of visitors gets close to 30. The count is derived by monitoring the movements in *rooms A, B* and *D*.

$$P_4(n) = \Box \begin{cases} vis_{(c,EA)} \rightarrow \text{if}(n+1 \geq 16) \text{ then } \star\langle n+1\rangle \rightarrow ack \rightarrow P_4(n+1) \text{ else } P_4(n+1) \\ vis_{(EA,c)} \rightarrow \text{if}(n = 30) \text{ then } \star\langle n-1\rangle \rightarrow ack \rightarrow P_4(n-1) \text{ else if } n > 0 \text{ then } P_4(n-1) \text{ else } P_4(0) \\ vis_{(ED,c)} \rightarrow \text{if}(n = 30) \text{ then } \star\langle n-1\rangle \rightarrow ack \rightarrow P_4(n-1) \text{ else if } n > 0 \text{ then } P_4(n-1) \text{ else } P_4(0) \\ vis_{(EB,ED)} \rightarrow P_4(n) \\ vis_{(EA,EB)} \rightarrow P_4(n) \\ vis_{(EB,EA)} \rightarrow P_4(n) \end{cases}$$

The adaptation function F_4 precludes entrance from the *corridor* to *room A* when the number communicated with the $\star$-event is greater or equal to 30.

$$F_4 = \Box_{n \in \mathcal{I}} \star\langle n\rangle \rightarrow \text{if } n \geq 30 \text{ then } l_a!R''_A.ack \rightarrow F_4 \text{ else } l_a!R'_A.ack \rightarrow F_4$$

The encoding for the unary cluster for Req. 4 comprises components *rooms A, B, D* and adaptation procedure Π_4,

$$S_4 = (\nu\, l_A, l_B, l_D)\left(\left(l_A\langle R'_A\rangle \underset{E_A}{\parallel} C_A(0)\right) \underset{E_{(A,B)}}{\parallel} \left(l_B\langle R'_B\rangle \underset{E_B}{\parallel} C_B(0)\right) \underset{E_{(B,D)}}{\parallel} \left(l_D\langle R'_D\rangle \underset{E_D}{\parallel} C_D(0)\right)\right) \underset{E_A,E_B,E_D}{\parallel} \Pi_4$$

where the event sets $E_{(A,B)}$ and $E_{(B,D)}$ encodes the connectivity through doors between *rooms A, B* and *rooms B, D* respectively. Through the verification approach in Chapter 6, we later verify that the above model refines $Spec_4$,

$$Spec_4 = \text{let } P(n) = \Box \begin{cases} n < 30\,\& & vis_{(c,EA)} \rightarrow P(n+1) \\ n > 0\,\& & vis_{(EA,c)} \rightarrow P(n-1) \\ n > 0\,\& & vis_{(EA,EB)} \rightarrow P(n) \\ n > 0\,\& & vis_{(EB,EA)} \rightarrow P(n) \\ n > 0\,\& & vis_{(EB,ED)} \rightarrow P(n) \\ n > 0\,\& & vis_{(ED,c)} \rightarrow P(n-1) \end{cases}$$

$$\text{within } P(0)$$

Consider, the scenario where tail-gating is discovered and the system designer decides to reduce the number of visitors allowed in the *exhibition area* to 28. For this change, the system designer needs to check that when the 18^{th} visitor enter the *exhibition area*, adaptation is triggered and the correct adaptation is communicated to *room A*. The system designer needs to verify that the adaptation pattern broadcasts an $\star$-event when 18 visitors are in the *exhibition area*, which our encoding of P_4 does and the adaptation function to preclude visitor entrance from the corridor to *room A* when the number of visitors reaches 18,

$$F'_4 = \Box_{n \in \mathcal{I}} \star\langle n\rangle \rightarrow \text{if } n \geq 28 \text{ then } l_a!R''_A.ack \rightarrow F'_4 \text{ else } l_a!R'_A.ack \rightarrow F'_4$$

The second approach (Ad. Proc. 4.2) To satisfy this approach, we define two adaptation procedures—Π_{4A} ensuring at most 15 visitors in *rooms A* and *B* and an adaptation procedure Π_{4B}

that guarantees at most 15 visitors in *rooms B* and *D*. The encoding for Π_{4B} is similar to Π_{2B} in Ad. Proc. 2.2. We thus only show the encoding for Π_{4A}.

From fig. 4.1, we know that *room A* is connected to the *corridor* and *room B* through a door and *room B* is connected through a door with *room A* and a one-way door to *room D*. We define a family of first-order events for the movement from and to *room A* and *room B*— $E_A = \{t_{(c,EA)}, t_{(EA,c)}, t_{(EA,EB)}, t_{(EB,EA)}\}$ and $E_B = \{t_{(EB,ED)}, t_{(EA,EB)}, t_{(EB,EA)}\}$ where t can be *vis*, *grd* or *emp* to represent a visitor, guard and employee movement respectively. Again, for simplicity, we only consider *vis*

1. The entrance to the *rooms* is open to visitors. We include the event $vis_{(c,EA)}$

$$R'_A = \Box_{t\in\{EB,c\}} \begin{cases} vis_{(t,EA)} \to R'_A \\ vis_{(EA,t)} \to R'_A \end{cases}$$

2. No more visitors are allowed to enter the *rooms A* and *B* respectively,

$$R''_A = \Box \begin{cases} vis_{(EB,EA)} \to R''_A \\ vis_{(EA,EB)} \to R''_A \\ vis_{(EA,c)} \to R''_A \end{cases}$$

We encapsulate the initial processes for the components of *room A, B* in locations $l_a\langle R'_A\rangle$, and $l_b\langle R'_B\rangle$ respectively. The adaptation procedure can adapt between the two functionalities through higher-order outputs to l_a and l_b. We assume the definition of the internal state processes of *room A* as $C_A(0)$, *room B* as $C_B(0)$ and *room D* as the process $C_D(0)$.

We now define the adaptation procedure Π_{4A} as the composition of adaptation pattern and function. We assume the class of events $\{\!|\star_{4A}|\!\} = \{\star_{4A}\langle n\rangle \mid n \geq 0\}$ to be the communication of the the number of visitors in *rooms A, B* between the *adaptation pattern* and *adaptation function*.

$$\Pi_{4A} = (\nu\, \{\!|\star_{4A}|\!\}, ack) \left(P_{4A}(0) \underset{\{\!|\star_{4A}|\!\}, ack}{\|} F_{4A} \right)$$

The adaptation pattern $P_{4A}(0)$ tracks the number of visitors in the *rooms A, B*. The process communicates through a distinguished $\star_{4A}$-events the state when the system *must*-adapt. For these cases, we trigger adaptation once the number of visitors in the rooms is 15. The adaptation pattern may trigger adaptation more frequently e.g., for all movement when the count is equal to or greater than 8. The pattern may also broadcasts an $\star_{4A}$-event at every step and the adaptation function can decide appropriately how to handle adaptation outcomes. For simplicity, here we trigger adaptation when the rooms are full and the entrance should be locked.

$$P_{4A}(0) = \Box \begin{cases} vis_{(c,EA)} \to \text{if}(n+1 \geq 15) \text{ then } \star_{4A}\langle 15\rangle \to ack \to P_{4A}(15) \text{ else } P_{4A}(n+1) \\ vis_{(EA,c)} \to \text{if}(n \geq 15) \text{ then } \star_{4A}\langle n-1\rangle \to ack \to P_{4A}(n-1) \text{ else if } n > 0 \text{ then } P_{4A}(n-1) \text{ else } P_{4A}(0) \\ vis_{(EB,ED)} \to \text{if } n \geq 15 \text{ then } \star_{4A}\langle n-1\rangle \to ack \to P_{4A}(n-1) \text{ else if } n > 0 \text{ then } P_{4A}(n-1) \text{ else } P_{4A}(0) \\ vis_{(EA,EB)} \to P_{4A}(n) \\ vis_{(EB,EA)} \to P_{4A}(n) \end{cases}$$

At every execution point, the pattern either accepts all first-order system events or triggers adaptation. The process F_{4A} represents the adaptation function. The process F_{4A} only adapts *room A* but monitors the behaviour of *room B* to determine the outcome of the adaptation. The

behaviour of *room A* is determined by the number of visitors in the rooms, which is communicated with the $\star$-event,

$$F_{4A} = \Box_{n \in \mathcal{I}} \star_{4A}\langle n \rangle \rightarrow \text{if } n \geq 15 \text{ then } l_a!R''_A.ack \rightarrow F_{4A} \text{ else } l_a!R'_A.ack \rightarrow F_{4A}$$

The encoding of the adaptation function is stateless, mirroring adaptation functions in SAA and only affect the system through adaptation, i.e., does not exhibit any (external) first-order events. The encoding for the adaptation procedure Π_{4B} that constraint the number of visitors in *rooms B, D* to 15 is defined similarly to Π_{4A} and Π_{2B}. We thus skip the presentation of its encoding. The *unary cluster* for the Req. 4 is defined as the composition of *rooms A, B* and *D* and the adaptation procedures Π_{4A} and Π_{4B}.

$$S_{4.2} = (\nu\, l_A, l_B)\left(\left(\left(l_A\langle R'_A\rangle \underset{E_A}{\parallel} C_A(0)\right) \underset{E_{(A,B)}}{\parallel} \left(l_B\langle R'_B\rangle \underset{E_B}{\parallel} C_B(0)\right)\right) \underset{E_{(D,B)}}{\parallel} \left(l_D\langle R'_D\rangle \underset{E_D}{\parallel} C_D(0)\right)\right) \underset{E_A, E_B, E_D}{\parallel} \left(\Pi_{4A} \underset{E_B}{\parallel} \Pi_{4B}\right)\Big)$$

where $E_{(A,B)} = \{vis_{(EB,EA)}, vis_{(EA,EB)}\}$ is the set of events representing the connectivity between *room A* and *room B*. Similarly $E_{(B,D)} = \{vis_{(EB,ED)}\}$ is the set of events representing the connectivity between *room B* and *room D*.

Specifications should reflect as closely as possible the requirement. Even though, we are enforcing a stricter access control on the *exhibition area*, we still verify that the above process $S_{4.2}$ trace refines the specifications $Spec_4$. If the system designer wants to verify a stricter specification, she can refine the original specification and verify the refinement in FDR. For instance, consider the process $Spec'_4$

$$Spec'_4 = Spec_{4A} \underset{E_B}{\parallel} Spec_{4B}$$

$$Spec_{4A} = \text{let } P(a,b) = \Box \begin{cases} a > 0 & \& \, vis_{(EA,EB)} \rightarrow P(a-1, b+1) \\ b > 0 & \& \, vis_{(EB,EA)} \rightarrow P(a+1, b-1) \\ b > 0 & \& \, vis_{(EB,ED)} \rightarrow P(a, b-1) \\ a > 0 & \& \, vis_{(EA,c)} \rightarrow P(a-1, b) \\ a + b < 15 & \& \, vis_{(c,EA)} \rightarrow P(a+1, b) \end{cases}$$

$$\text{within } P(0,0)$$

$$Spec_{4B} = \text{let } P(b,d) = \Box \begin{cases} d + b < 15 & \& \, vis_{(EA,EB)} \rightarrow P(b+1, d) \\ b > 0 & \& \, vis_{(EB,EA)} \rightarrow P(b-1, d) \\ b > 0 & \& \, vis_{(EB,ED)} \rightarrow P(b-1, d+1) \\ d > 0 & \& \, vis_{(ED,c)} \rightarrow P(b, d-1) \end{cases}$$

$$\text{within } P(0,0)$$

Using FDR, we verify that $Spec_4 \sqsubseteq_{T(ACSP)} Spec'_4$ and thus by transitivity, we know

$$Spec'_4 \sqsubseteq_{T(ACSP)} S_{4.2} \quad \text{implies } Spec_4 \sqsubseteq_{T(ACSP)} S_{4.2}$$

Requirement 5: ***The people in the building should be able to reach the nearest emergency exit*** This requirement comprises the whole building, which to verify entails translating

and verifying the whole art gallery. In order to preserve the compositionality in our model, we change the requirement for each room to say

Requirement E. In the case of an emergency, all doors should be open ◊

Irrespective of the number of visitors in the rooms, visitors should not be precluded entrance to any room as it may be the only path to an exit. This specification can be verified compositionally for each room. Here, we only show the encoding and specification for *room D*. The encoding and specification for all the other rooms in the art gallery is similar.

The components affecting the satisfaction of this requirement (for *room D*) are the room and the fire alarm panel. The alarm panel broadcasts non-deterministically an *emergency* event when the art gallery needs to be evacuated. The first-order events for the components are $E_F = \{emergency\}$.

Recall that the first-order events of *room D* are $E_d = \{t_{(ED,c)}, t_{(EB,ED)}\}$ that represents the rooms connectivity through a door with the *corridor* (c) and *room B* (B). Here t can be vis, grd or emp to represent a visitor, guard and employee movement. For simplicity we only consider vis. *Room D* can be in one of two functionalities:

1. The entrance to *room D* is open and visitors are allowed to move from *room B* to *room D* represented by the event $vis_{(EB,ED)}$

$$R'_D = (vis_{(EB,ED)} \rightarrow R'_D) \,\Box\, (vis_{(ED,c)} \rightarrow R'_D)$$

2. No more visitors are allowed to enter *room D*. We encode this by precluding the event $vis_{(EB,ED)}$,

$$R''_D = vis_{(ED,c)} \rightarrow R''_D$$

The internal state of *Room D* tracks the number of visitors entering and leaving *room D* to make sure that the number of visitors leaving is less than or equal to the number of visitors entering *room D*. We assume the definition of the process C_D from Req. 2 as the internal state process. The initial configuration of *room D* is encapsulated by a location l_d, with the process R'_D—$l_d\langle R'_D\rangle$. Theoretically, this process satisfies the requirement as the doors are already open. However, we know that Req. 2 will at specific execution points close the entrance to room D for visitors. We define an adaptation procedure that adapts the behaviour of *room D* through a higher-order output to l_d, on the broadcast of the *emergency* event

$$P_{5D} = \Box \begin{cases} e \rightarrow P_{5D} & \text{where } e \in E_d \\ emergency \rightarrow \star \rightarrow ack \rightarrow P_{5D} \end{cases}$$

The adaptation function opens the doors connecting *room B* to *room D* on an $\star$-event,

$$F_{5D} = \star \rightarrow l_d!D_0.ack \rightarrow F_{5D}$$

The adaptation procedure is defined as

$$\Pi_{5D} = (\nu\star, ack)\left(P_{5D} \underset{\{\star,ack\}}{\parallel} F_{5D}\right)$$

The composition of the adaptation procedure and the component in its scope to be verified for this requirement is defined below. The process S_{5D} is referred to as the unary cluster for Req. 5 for *room D*.

$$S_{5D} = (\nu\, l_d)\left(\left(l_d\langle R'_D\rangle \underset{E_a}{\parallel} C_D(0)\right) \underset{E_D\{emergency\}}{\parallel} \Pi_{5D}\right)$$

We later verify that S_5 refines by the *failure* semantic the following specification, [1]

$$Spec_{5D} = R'_D \triangle (emergency \rightarrow RUN(E_d))$$

This specification is a liveness property, we want to show that in the case of an emergency, visitors are not refused an entrance or exit to/from *room D*.

Requirement 6: *HVAC software updates should be installed within 3 hours* In Req. 3, we disconnect the *HVAC* once a visitor connects to the *access point*. However, in the presence of a pending update, we temporarily disconnects visitors to give the *HVAC* a chance to install important updates. We assume an event *update* that the *HVAC* broadcasts when an update is overdue. We satisfy the requirement by following the approach presented in section 4.1. We first identify the components that affect the satisfaction of Req. 6—*HVAC* and the *access point*. We assume the sets of first-order events E_H and E_{AP} are the set of events for the *HVAC* and *access point* components respectively, presented for Req. 3. The adaptation procedure adapts the behaviour of the *access point* and monitors the behaviour of the *HVAC* components. For simplicity, we only show the encoding for the *access point*.

The *access point* can be in one of the following functionalities,

1. The *HVAC* is connected and connections from both employees and visitors are pending. This is the initial configuration of the *access point*, where only the *HVAC* is connected

$$A0 = conn_{hvac} \rightarrow \underset{t\in\{vis,emp\}}{\Box} \begin{cases} conn_t \rightarrow A0 \\ disconn_t \rightarrow A0 \end{cases}$$

2. Visitors are disconnected, and thus the *HVAC* is reconnected. Thereafter, both visitors and employees can reconnect to the access point intermittently

$$A4 = disconnect \rightarrow conn_{hvac} \rightarrow \underset{t\in\{vis,emp\}}{\Box} \begin{cases} conn_t \rightarrow A4 \\ disconn_t \rightarrow A4 \end{cases}$$

We assume the definition of the process C_{ap} to track the internal state of the access point to be the same as the process presented for Req. 3. We define an adaptation procedure Π_6 that adapts the *access point* to $A4$ on the *update* event,

$$\Pi_6 = (\nu\ \{|\star|}, ack)\left(P_6 \underset{\{|\star,ack|\}}{\parallel} F_6\right)$$

The process P_6 is the adapation pattern that tracks the state of the *access point* and the *hvac*. The process P_6 triggers adaptation on the *update* event. As explained in section 5.1.1, adaptation procedures are the composition of an adaptation pattern and an adaptation function synchronizing

[1] the function $RUN(A)$ offers the choice over all events in A perpetually [59].

over $\star$-events and *ack* event.

$$P_6 = \Box \begin{cases} e \rightarrow P_6 & \text{where } e \in E_{ap}, E_{hvac} \backslash \{update\} \\ update \rightarrow \star \rightarrow ack \rightarrow P_6 \end{cases}$$

On the $\star$-event, the process F_6 adapts, through higher-order outputs to *ap*, the behaviour of the *access point* to $A4$,

$$F_6 = \star \rightarrow ap!A4.ack \rightarrow F_6$$

The overall process that guarantees the satisfaction of Req. 6, known as the *unary cluster*, is

$$S_6 = (\nu\, ap, h) \left(\left(ap\langle A0 \rangle \underset{E_{(ap,h)}}{\parallel} h\langle H_0 \rangle \right) \underset{E_H, E_{AP}}{\parallel} \Pi_6 \right)$$

We later verify that the process S_6 trace refines

$$Spec_6 = \underset{t \in \{vis, emp\}}{\Box} \begin{cases} disconn_t \rightarrow Spec_6 \\ conn_t \rightarrow Spec_6 \\ update \rightarrow disconnect \rightarrow Spec_6 \end{cases}$$

Appendix D

Encoding for the Smart Stadium Case-Study

Requirement D: ***During a match, visitors are allowed to roam to other non-empty sections*** Once a match starts, represented by the *during* event, we expect most visitors to be seated. This allows us to relax the access control between sections to improve user experience. Any visitor is free to access other *non-empty* sections or VIP sections, despite not having a ticket to it. If a lower or upper section is empty by the start of a match (on the *during* event) the section is closed off to reduce energy usage across the stadium. Nonetheless, ticket holders to an empty section are still allowed to enter, in case they arrive late. Then the section is then opened to all visitors as it is no longer empty.

For satisfying this requirement, we adapt the behaviour of the *access controller* and monitor the *match status* component. The access controller can be in one of two functionalities

1. Visitors are allowed to enter s as it is non-empty,

$$SRoam = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases} goto_{v,a,s} & \rightarrow SRoam \\ goto_{v,s,a} & \rightarrow SRoam \end{cases}$$

2. Only ticket holders are allowed to enter , as s is empty and s can be used as a backup by other sections through the $open_{s,s'}$ event,

$$AC_3 = SOpen \;|||\; \Box_{s' \in SectionID \setminus s} \Big\{ open_{s,s'} \rightarrow SKIP$$

We now define an adaptation procedure Π_D as the composition of an adaptation pattern P_D and adaptation function F_D. The process P_D triggers adaptation on the *during* event or for an empty section when the first visitor enters the section. A section is empty, if there are no *goto* events entering the section between the $before$ and the *during* events. The adaptation pattern communicates to the decision process F_D whether the section is empty.

$$
\begin{aligned}
P_D = {} & \text{let} \\
& B(emp, dur) = \Box_{\substack{v \in Visitors \\ a \in adj}} \begin{cases}
goto_{v,a,s} & \rightarrow \text{if } dur \wedge emp \\
 & \quad \text{then } \star\langle F\rangle \rightarrow ack \rightarrow B(F, dur) \\
 & \quad \text{else } B(F, dur) \\
goto_{v,s,a} & \rightarrow B(emp, dur) \\
before & \rightarrow B(T, F) \\
after & \rightarrow B(emp, F) \\
open & \rightarrow B(emp, dur) \\
during & \rightarrow \star\langle emp\rangle \rightarrow ack \rightarrow B(emp, T) \\
empty & \rightarrow B(T, dur)
\end{cases} \\
& \text{within } B(T, F)
\end{aligned}
$$

Because an adaptation pattern comprises only first-order events, we check using FDR that the pattern does not affect the component behaviour but just monitors. We verify that the adaptation pattern does not, unless adapting, refuses any events from the components eq. (ID.1) or introduces events not in the interface of the components eq. (ID.2),

$$RUN(E_{ac}, E_m) \quad \sqsubseteq_{F(ACSP)} \quad P_D \setminus \{|\star, ack|\} \tag{ID.1}$$

$$P_D \setminus \{|\star, ack|\} \quad \sqsubseteq_{T(ACSP)} \quad RUN(E_{ac}, E_m) \tag{ID.2}$$

The adaptation function, encoded in the process F_D opens a section to all visitors if not empty or is a VIP section otherwise access remains for ticket holders only

$$F_D = \star\langle emp\rangle \rightarrow \text{if } emp \wedge s \notin \{|VIP|\} \text{ then } ac!AC_3.ack \rightarrow F_D \text{ else } ac!SRoam.ack \rightarrow F_D$$

The adaptation procedure is defined as the composition of the processes F_D and P_D

$$\Pi_D = (\nu \star, ack)\left(F_D \underset{\{|\star, ack|\}}{\parallel} P_D\right)$$

The *unary cluster* encoding the satisfaction of Req. D for s is defined as

$$S_D = (\nu\, ac, ms)\left((AccessController \;|||\; MatchStatus) \underset{E_{ac}, E_m}{\parallel} \Pi_D\right)$$

We now discuss the specification for Req. D. The process R encodes the most permissive behaviour for s that applies *before* or *after* a match, when the requirement does not apply. The process D describes the behaviour during a match according to the requirement, where if s is empty only ticket holders are allowed to enter, otherwise all visitors are allowed to enter.

$$
\begin{aligned}
&Spec_D = \text{let} \\
&\quad R(emp) = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases} before & \rightarrow R(T) \\ during & \rightarrow D(emp) \\ after & \rightarrow R(T) \\ goto_{v,s,a} & \rightarrow R(emp) \\ goto_{v,a,s} & \rightarrow R(F) \\ open & \rightarrow R(emp) \\ empty & \rightarrow R(emp) \end{cases} \\
&\quad D(emp) = \Box_{\substack{a \in adj \\ v \in Visitors \\ s' \in SectionID \setminus s}} \begin{cases} goto_{v,s,a} & \rightarrow D(emp) \\ emp \,\&\, goto_{s,a,s} & \rightarrow D(emp) \\ \neg emp \,\&\, goto_{v,a,s} & \rightarrow D(emp) \\ before & \rightarrow R(T) \\ after & \rightarrow R(T) \\ during & \rightarrow D(emp) \\ open_{s,s'} & \rightarrow D(emp) \\ emp \;\&\; open_{s',s} & \rightarrow D(emp) \\ empty & \rightarrow D(emp) \end{cases} \\
&\quad \text{within } R(T)
\end{aligned}
$$

With our verification approach, we verify that for s, the unary cluster S_D trace refines $Spec_D$.

$$Spec_D \sqsubseteq_{T(ACSP)} S_D$$

Requirement E: ***On a windy day, the system should attempt to empty the upper area***
This requirement aims to protect visitors from strong winds. Sections in the upper area are particularly exposed to wind. Upon detecting strong winds during a match, the system changes the access control in the upper sections so that only ticket holders (and employees) are allowed to enter. The system also aims to find an alternative, more sheltered, section in the lower area for visitors to sit in. Once an alternative section is opened, visitors are directed to the new section. We model this by closing the exposed section. In sold-out matches, there would not be any empty sections in the stadium. In this case, only ticket holders are allowed to enter an upper section for the remainder of the match.

To satisfy this requirement, we adapt the behaviour of the *access controller* and monitor the *match status* and *wind monitor* components.

The *access controller* can be in one of two functionalities

1. A section s is closed to visitors if s is exposed to strong winds and an alternative seating has been identified for the ticket holders. Note how visitors are only allowed to leave s to move to the *corridor*. Adjacent sections would also be exposed to the wind and thus visitors movement between sections is precluded,

$$SExit = \Box_{v \in Visitors} \left\{ goto_{v,s,Corr} \rightarrow SExit \right.$$

2. Strong wind has been detected and we restrict access to only ticket holders until an alternative section, that is not exposed to the wind itself, is found. The process requests an alternative section through the event $open_{s,s'}$ to all lower sections,

$$AC_4 = SOpen \;|||\; \Box_{s' \in \{\!| L |\!\}} \left\{ open_{s,s'} \rightarrow SKIP \right.$$

We now define an adaptation procedure Π_E that comprises an adaptation pattern P_E and adaptation function F_E.

The adaptation pattern encoded in process P_E monitors the *access controller*, *match status* and *wind monitor* components. Adaptation is triggered on the *during* event if wind has been detected

prior the start of a match, on the *wind* event during a match and once an alternative section has been found for an exposed section. We communicate with the $\star$-event if an alternative section has been found for an exposed section in the form of a boolean variable.

$$
\begin{aligned}
&P_E = \text{let} \\
&\quad B(w, ev) = \Box_{\substack{a \in adj \\ s' \in SectionID \setminus s}} \begin{cases} goto & \rightarrow B(w, ev) \\ before & \rightarrow B(F, F) \\ during & \rightarrow \text{if } w \text{ then } \star\langle F\rangle \rightarrow ack \rightarrow B(w, T) \text{ else } B(w, T) \\ after & \rightarrow B(w, F) \\ open_{s,s'} & \rightarrow \text{if } ev \wedge w \wedge s' \notin \{\!|H|\!\} \text{ then } \star\langle T\rangle \rightarrow ack \rightarrow B(w, ev) \text{ else } B(w, ev) \\ open_{s',s} & \rightarrow B(w, ev) \\ empty & \rightarrow B(w, ev) \\ wind & \rightarrow \text{if } ev \text{ then } \star\langle F\rangle \rightarrow ack \rightarrow B(T, ev) \text{ else } B(T, ev) \end{cases} \\
&\quad \text{within } B(F, F)
\end{aligned}
$$

Because adaptation patterns comprises only of first-order events, we check using FDR that the pattern P_E does not affect the components behaviour. We verify that the adaptation pattern does not, unless adapting, refuses any events from the components eq. (ID.3) or introduces events not in the interface of the components eq. (ID.4),

$$RUN(E_{ac}, E_m, E_w) \quad \sqsubseteq_{F(ACSP)} \quad P_E \setminus \{\!|\star, ack|\!\} \tag{ID.3}$$

$$P_E \setminus \{\!|\star, ack|\!\} \quad \sqsubseteq_{T(ACSP)} \quad RUN(E_{ac}, E_m, E_w) \tag{ID.4}$$

The adaptation function performs a higher-order output to *ac* to adapt the behaviour of the *access controller*,

$$
\begin{aligned}
F_E = \star\langle backup\rangle \rightarrow &\text{if } backup \wedge s \in \{\!|H|\!\} \text{ then } ac!SExit.ack \rightarrow F_E \\
&\text{else if } s \in \{\!|H|\!\} \text{ then } ac!AC_4.ack \rightarrow F_E \\
&\text{else } ack \rightarrow F_E
\end{aligned}
$$

The adaptation procedure is defined as the composition of the processes F_E and P_E synchronizing over the scoped first-order events $\star$ and *ack*,

$$\Pi_E = (\nu \star, ack)\left(F_E \underset{\{\!|\star, ack|\!\}}{\parallel} P_E\right)$$

The *unary cluster* for a section s also comprises the *wind monitor* component. In ACSP, this is modelled as a non-deterministic event. We thus do not include its encoding here,

$$S_E = (\nu\, ac, ms)\left((AccessController \;|||\; MatchStatus) \underset{E_{ac}, E_m}{\parallel} \Pi_E\right)$$

We now discuss the specification for Req. E. The process R describes the most permissive behaviour that applies before and after a match, whereas the process W describes the behaviour during a match where access to s is affected by the wind level.

$$Spec_E = \text{let}$$

$$R(w) = \Box \begin{cases} goto & \to R(w) \\ wind & \to R(s \in \{|H|\}) \\ open & \to R(w) \\ before & \to R(F) \\ during & \to W(w,F) \\ after & \to R(w) \\ empty & \to R(w) \end{cases}$$

$$W(w, backup) = \Box_{\substack{v \in Visitors \\ a \in adj \\ s' \in SectionID}} \begin{cases} w \wedge \neg backup \,\&\, goto_{s,a,s} & \to W(w, backup) \\ \neg w \,\&\, goto_{v,a,s} & \to W(w, backup) \\ goto_{v,s,a} & \to W(w, backup) \\ empty & \to W(w, backup) \\ after & \to R(w) \\ before & \to R(F) \\ during & \to W(w, F) \\ wind & \to W(s \in \{|H|\}, F) \\ open_{s,s'} & \to W(w, w \wedge s' \notin \{|H|\}) \\ \neg w \;\&\; open_{s',s} & \to W(w, backup) \end{cases}$$

$$\text{within } R(F)$$

We verify that the process S_E trace refines $Spec_E$ for a section s.

$$Spec_E \sqsubseteq_{T(ACSP)} S_E$$

Requirement F: ***During a match, the system should aim to keep noise levels below a threshold.*** The stadium has a noise level detector in each section. The level of noise in the stadium is a ratio between the measured noise and the number of sections open. Thus, if during a match the noise level in s is very high, the system tries to open an empty section to encourage visitors to distribute themselves more evenly around the stadium as means to reduce the overall noise level. As an immediate response to an increase in noise levels, we also restrict the access to s to ticket holders only.

For satisfying this requirement, we define an adaptation procedure Π_F that adapts the *access controller* of a section and monitors the *noise detector* and the *match status* components.

The *access controller* functionality on detecting a high level of noise is adapted so only ticket holders are allowed in the section and an alternative section is opened.

$$AC_5 = SOpen \;|||\; \Box_{s' \in SectionID \setminus s} \; open_{s,s'} \to SKIP$$

We now define an adaptation pattern that monitors the *access controller*, *noise detector* and *match status* components. The adaptation pattern triggers adaptation on the *noisy* event during a match.

$$P_F = \text{let}$$

$$B(ev) = \Box \begin{cases} goto & \to B(ev) \\ before & \to B(F) \\ after & \to B(F) \\ during & \to B(T) \\ open & \to B(ev) \\ noisy\langle T \rangle & \to \text{if } ev \text{ then } \star \to ack \to B(ev) \text{ else } B(ev) \\ empty & \to B(ev) \\ noisy\langle F \rangle & \to B(F) \end{cases}$$

$$\text{within } B(F)$$

The process F_F adapts the *access controller* by performing a higher-order output on ac when

the section becomes too noisy during a match.

$$F_F = \star \rightarrow ac!AC_5.ack \rightarrow F_F$$

The adaptation procedure is the composition of the processes P_F and F_F

$$\Pi_F = (\nu \star, ack)\left(F_F \underset{\{\star,ack\}}{\parallel} P_F\right)$$

The unary cluster for the satisfaction of Req. F for s also comprises the *noise detector*. This is modelled as a non-deterministic stream of alternating noisy events,

$$Noisy(b) = noisy\langle b\rangle \rightarrow Noisy(\neg b)$$

We encapsulate the process for the *noise detector* in location np,

$$NoisePanel = np\langle Noisy(T)\rangle$$

The *unary cluster* is defined as the composition of the adaptation procedure Π_F and the components in its scope— the *access controller, match status* and *noise detector*.

$$S_F = (\nu\, ac, np, ms)\left((AccessController \;|||\; MatchStatus \;|||\; NoisePanel) \underset{E_{ac},E_m,E_n}{\parallel} \Pi_F\right)$$

We now discuss the specification for Req. F. The process R describes the most permissive behaviour of the components before and after a match, when the *noisy* event has no effect on the access to s. During a match, the requirement is encoding by process Dur, which on the *noisy* event restricts the access to ticket holders.

$Spec_F =$ let

$$R = \Box \begin{cases} goto & \rightarrow R \\ noisy\langle T\rangle & \rightarrow R \\ open & \rightarrow R \\ before & \rightarrow R \\ during & \rightarrow Dur(F) \\ after & \rightarrow R \\ empty & \rightarrow R \\ niosy\langle F\rangle & \rightarrow R \end{cases} \qquad Dur(n) = \underset{\substack{v \in Visitors \\ a \in adj \\ s' \in SectionID}}{\Box} \begin{cases} n \;\&\; goto_{s,a,s} & \rightarrow Dur(n) \\ \neg n \;\&\; goto_{v,a,s} & \rightarrow Dur(n) \\ goto_{v,s,a} & \rightarrow Dur(n) \\ empty & \rightarrow Dur(n) \\ after & \rightarrow R \\ before & \rightarrow R \\ during & \rightarrow Dur(n) \\ \neg n \;\&\; open_{s',s} & \rightarrow Dur(n) \\ n \;\&\; open_{s,s'} & \rightarrow Dur(n) \\ noisy\langle b\rangle & \rightarrow Dur(b) \end{cases}$$

within R

Requirement G: ***A section may be re-opened as a backup section if it is not noisy, not exposed to strong wind, empty and its fire alarm is off.*** A section must be empty, not exposed to strong winds and not in an emergency state to be used as a backup. During a match, a backup section permits all movements to and from s, whereas after a match visitors from adjacent sections can exit to the *corridor* through s. A section s' can request to open s as a backup through the event $open_{s',s}$.

To satisfy this requirement, we define an adaptation procedure Π_G that adapts the behaviour of the *access controller* and monitors the *noise detector, fire alarm, match status* and *wind panel* components.

We now define the behaviour of the *access controller* on adaptation. The access controller can be in one of the two functionalities

1. During a match, on the $open_{s',s}$ event, s is opened to all visitors,

$$SRoam = \Box_{\substack{a \in adj \\ v \in Visitors}} \begin{cases} goto_{v,a,s} & \to SRoam \\ goto_{v,s,a} & \to SRoam \end{cases}$$

2. After a match, if the section is eligible to be used a backup, i.e., it is empty, not exposed to wind and the fire alarm is not triggered, then visitors from adjacent sections can exit through the empty section,

$$SExitAdj = \Box_{\substack{a \in adj \setminus Corr \\ v \in Visitors}} \begin{cases} goto_{v,s,Corr} & \to SExitAdj \\ goto_{v,a,s} & \to SExitAdj \end{cases}$$

The process P_G encodes an adaptation pattern where adaptation is triggered on an $open_{s',s}$ if all the criteria are met. We communicate with the $\star$-event the *match status* s this determines the access rights.

$$\begin{aligned} P_G = & \text{let} \\ & applies(alrm, wnd, n) = \neg(alrm \vee wnd \vee n) \\ & B(a, w, n, emp, dur) = \\ & \Box_{\substack{v \in Visitors \\ a \in adj \\ s' \in SectionID}} \begin{cases} goto_{v,a,s} & \to B(a, w, n, False, dur) \\ goto_{v,s,a} & \to B(a, w, n, emp, dur) \\ before & \to B(F, F, F, T, F) \\ after & \to B(a, w, n, emp, F) \\ open_{s,s'} & \to B(a, w, n, emp, dur) \\ open_{s',s} & \to \text{if } applies(a, w, n) \wedge emp \\ & \qquad \text{then } \star\langle dur \rangle \to ack \to B(a, w, n, emp, dur) \\ & \qquad \text{else } B(a, w, n, emp, dur) \\ noisy\langle b \rangle & \to B(a, w, b, emp, dur) \\ during & \to B(a, w, n, emp, T) \\ empty & \to B(a, w, n, emp, dur) \\ alarm\langle b \rangle & \to B(b, w, n, emp, dur) \\ wind & \to B(a, s \in \{\!|H|\!\}, n, emp, dur) \end{cases} \\ & \text{within } B(F, F, F, T, F) \end{aligned}$$

The adaptation function adapts the behaviour of the *access controller* through a higher-order output on *ac*. The communicated behaviour depends on the match status: during a match a

section becomes accessible by all visitors and after a match visitors from adjacent sections can exit the stadium from s. We thus communicate with the $\star$-event a boolean that is *true* if a match is ongoing and false otherwise,

$$F_G = \star\langle dur\rangle \rightarrow \text{if } dur \text{ then } ac!SRoam.ack \rightarrow F_G \text{ else } ac!SExitAdj.ack \rightarrow F_G$$

We define an adaptation procedure Π_G as the composition of the processes F_G and P_G that represent the adaptation function and adaptation pattern respectively.

$$\Pi_G = (\nu \star, ack)\left(F_G \underset{\{|\star, ack|\}}{\parallel} P_G\right)$$

The verification for a section s is encapsulated in the *unary cluster*,

$$S_G = (\nu\, ac, np, ms, f)\left((AccessController \,|||\, MatchStatus \,|||\, NoisePanel \,|||\, AlarmPanel) \underset{E_{ac}, E_m, E_n, E_f}{\parallel} \Pi_G\right)$$

We now discuss the specification for Req. G. We define three processes *Bef, Dur, Aft* that describe the behaviour before, during and after a match respectively. The *Bef* process does not include the event $open_{s',s}$ implying that the section cannot be used as a backup. In the Dur and Aft process, the $open_{s',s}$ is only allowed if the section can be used as a backup. The processes differ because in the Dur processes visitors are allowed to move from the corridor into the section, which is not allowed after a match.

$$
\begin{array}{l}
Spec_G = \text{let} \\
\quad Dur\left(\begin{array}{c} alrm, \\ wnd, n, \\ emp, opn \end{array}\right) = \underset{\substack{a \in adj \\ v \in Visitors \\ s' \in SectionID}}{\square} \begin{cases} goto_{v,s,a} & \to Dur(alrm, wnd, n, emp, opn) \\ \neg emp \vee opn \vee s \in \{\!|VIP|\!\} \ \& \ goto_{v,a,s} & \to Dur(alrm, wnd, n, False, opn) \\ noisy\langle b\rangle & \to Dur(alrm, wnd, b, emp, opn) \\ wind & \to Dur(alrm, s \in \{\!|H|\!\}, n, emp, opn) \\ alarm\langle b\rangle & \to Dur(b, wnd, n, emp, opn) \\ after & \to Aft(alrm, wnd, n, emp, opn) \\ empty & \to Dur(alrm, wnd, n, emp, opn) \\ open_{s,s'} & \to Dur(alrm, wnd, n, emp, opn) \\ \neg(alrm \vee wnd \vee n) \wedge emp \ \& \ open_{s',s} & \to Dur(alrm, wnd, n, emp, T) \end{cases} \\
\quad Aft\left(\begin{array}{c} alrm, \\ wnd, n, \\ emp, opn \end{array}\right) = \underset{\substack{a \in adj \\ v \in Visitors}}{\square} \begin{cases} goto_{v,s,a} & \to Aft(alrm, wnd, n, emp, opn) \\ a \neq Corr \wedge opn \ \& \ goto_{v,a,s} & \to Aft(alrm, wnd, n, emp, opn) \\ before & \to Aft(F, F, F, T, F) \\ noisy\langle b\rangle & \to Aft(alrm, wnd, b, emp, opn) \\ wind & \to Aft(alrm, s \in \{\!|H|\!\}, n, emp, opn) \\ alarm\langle b\rangle & \to Aft(b, wnd, n, emp, opn) \\ empty & \to Aft(alrm, wnd, n, T, opn) \\ open_{s,s'} & \to Aft(alrm, wnd, n, emp, opn) \\ \neg(alrm \vee wnd \vee n) \vee emp \ \& \ open_{s',s} & \to Aft(alrm, wnd, n, emp, T) \end{cases} \\
\quad Bef\left(\begin{array}{c} alrm, \\ wnd, n, \\ emp, opn \end{array}\right) = \underset{\substack{a \in adj \\ v \in Visitors}}{\square} \begin{cases} goto_{Emp,s,a} & \to Bef(alrm, wnd, n, emp, opn) \\ goto_{v,a,s} & \to Bef(alrm, wnd, n, F, opn) \\ during & \to Dur(alrm, wnd, n, emp, opn) \\ noisy\langle b\rangle & \to Bef(alrm, wnd, b, emp, opn) \\ wind & \to Bef(alrm, s \in \{\!|H|\!\}, n, emp, opn) \\ alarm\langle b\rangle & \to Bef(b, wnd, n, emp, opn) \\ empty & \to Bef(alrm, wnd, n, emp, opn) \\ open_{s,s'} & \to Bef(alrm, wnd, n, emp, opn) \end{cases} \\
\text{within } R(F)
\end{array}
$$

Using our verification approach we verify that the process S_G trace refines $Spec_G$,

$$Spec_G \sqsubseteq_{T(ACSP)} S_G$$

Requirement H: ***To minimize energy usage, noise level sensors are switched off before and after match or if the section they are in is empty.*** In order to minimize energy usage in the stadium, we only switch on noise detectors in non-empty sections and during a match. We model the switching off a noise detector by broadcasting a $noisy\langle F\rangle$ event if the last event was a $noisy\langle T\rangle$ and stop further broadcasting of *noisy* events. If the last event was a $noisy\langle F\rangle$ or there was no prior *noisy* event, then no *noisy* events is broadcast until the noise detector is switched back on. When on, the noise detector is a non-deterministic cycle of alternating *noisy* events. We define an adaptation procedure Π_H to ensure the satisfaction of Req. H which adapts the *noise detector* component and monitors the *match status, access component* components.

The *noise detector* component can be in one of the three functionalities

1. The noise sensor is switched on and thus the process broadcast (non-deterministically) *noisy* events, alternating between T and F,

$$Noisy(b) = noisy\langle b\rangle \to Noisy(\neg b)$$

2. If a section is noisy and the *noise detector* is to be switched off, represented by suppressing the *noisy* events, we broadcast a $noisy\langle F\rangle$ event followed by the $STOP$ process

$$Noisy_1 = noisy\langle F\rangle \rightarrow STOP$$

3. The sensor is to be switched off and the section is not noisy, then we do not broadcast the $noisy\langle F\rangle$ event

$$Noisy_2 = STOP$$

We encapsulate the noise panel inside location np so the adaptation procedure can adapt its behaviour. Initially, the noisy monitor is switched off

$$NoisePanel = np\langle STOP\rangle$$

We define an adaptation procedure Π_H that guarantees the satisfaction of Req. H. The adaptation procedure comprises an adaptation pattern P_H and function F_H. The pattern triggers adaptation when the match status changes to switch on or off the *noise detector* and when the section becomes empty or is no longer empty during a match.

$$\begin{aligned}
P_H = &\,\text{let}\\
&B(n, emp, ev) = \Box_{\substack{a \in adj\\ v \in Visitors}} \begin{cases}
before & \rightarrow \star\langle n, T\rangle \rightarrow ack \rightarrow B(n, T, before)\\
during & \rightarrow \text{if } emp \text{ then } \star\langle n, emp\rangle \rightarrow ack \rightarrow B(n, emp, during)\\
 & \qquad \text{else } B(n, emp, during)\\
after & \rightarrow \star\langle n, T\rangle \rightarrow ack \rightarrow B(n, emp, after)\\
noisy\langle b\rangle & \rightarrow B(b, emp, ev)\\
goto_{v,a,s} & \rightarrow \text{if } ev = during \wedge emp \text{ then } \star\langle n, F\rangle \rightarrow ack \rightarrow B(n, emp, ev)\\
 & \qquad \text{else } B(n, emp, ev)\\
goto_{v,s,a} & \rightarrow B(n, emp, ev)\\
open & \rightarrow B(n, emp, ev)
\end{cases}\\
&\text{within } B(F, T, before)
\end{aligned}$$

We communicate with the $\star$-events two boolean values: the last value of the *noisy* event and if a section is empty. If a section is empty and the last *noisy* event was T, then the process should adapt to $Noisy_1$, otherwise to $STOP$. If the section is not empty, the component is adapted to $Noisy_0$, initialized to the correct parameter.

$$\begin{aligned}
F_H = \star\langle n, emp\rangle \rightarrow &\,\text{if } emp \wedge n \text{ then } np!Noisy_1 \rightarrow ack \rightarrow F_H\\
&\text{else if } \neg emp \wedge n \text{ then } np!Noisy_0(F) \rightarrow ack \rightarrow F_H\\
&\text{else if } \neg emp \wedge \neg n \text{ then } np!Noisy_0(T) \rightarrow ack \rightarrow F_H\\
&\text{else if } emp \wedge \neg n \text{ then } np!Noisy_2 \rightarrow ack \rightarrow F_H\\
&\text{else } SKIP
\end{aligned}$$

The adaptation procedure is defined as the composition of processes F_H and P_H

$$\Pi_H = (\nu \star, ack)\left(F_H \underset{\{\mid \star, ack \mid\}}{\parallel} P_H\right)$$

The unary cluster for verifying Req. H in s is

$$S_H = (\nu\, np, ac, ms)\left((NoisePanel \mid\mid\mid AccessController \mid\mid\mid MatchStatus) \underset{E_{ac}, E_m, E_n}{\parallel} \Pi_H\right)$$

We now discuss the specification for Req. H, where we never broadcast consecutive *noisy* events with the same boolean value, the first *noisy* event after the *before* event is always $noisy\langle T\rangle$ and no *noisy* events are sent following the *after* event.

$$\begin{aligned} Spec_H = &\text{let} \\ &B(n, emp) = \underset{\substack{a \in adj \\ v \in Visitors}}{\square} \begin{cases} before & \rightarrow B(n, T) \\ during & \rightarrow B(n, emp) \\ after & \rightarrow B(n, T) \\ \neg emp\ \&\, noisy\langle \neg n\rangle & \rightarrow B(\neg n, emp) \\ emp \wedge n\ \&\, noisy\langle F\rangle & \rightarrow B(F, emp) \\ goto_{v,a,s} & \rightarrow B(n, F) \\ goto_{v,s,a} & \rightarrow B(n, emp) \\ open & \rightarrow B(n, emp) \\ empty & \rightarrow B(n, T) \end{cases} \\ &\text{within } B(F, T) \end{aligned}$$

With our verification approach, we verify that for a section s, the process S_H trace refines $Spec_H$.

$$Spec_H \sqsubseteq_{T(ACSP)} S_H$$

Requirement I: *Floodlights usage should kept to a minimum* The last requirement for our case-study concerns the efficient use of the floodlights. The floodlights should be turned off during the day when there is sunlight, and if a section remains empty by the start of a match. If a section becomes temporarily empty during a match, which may happen if a section is scarcely filled, the lights should not be switched off as this may affect user experience during a match. In our process language, we cannot accurately model the time of the day and thus the encoding of the flood light component is a non-deterministic broadcast of $lights\langle on\rangle$ and $lights\langle off\rangle$ events. We can only verify that the lights are switched off when a section is empty.

We satisfy this requirement by adapting the behaviour of the *floodlights* component and monitoring the *match status* and *access controller* components.

The *floodlight* component that can be in one of the two functionalities

1. The lights should be turned on: $floodlights\langle T\rangle \rightarrow STOP$

2. The lights should be turned off: $floodlights\langle F\rangle \rightarrow STOP$

We encapsulate the component in location lp so it can be adapted by adaptation procedures,

$$LightPanel = lp\langle STOP\rangle$$

The adaptation procedure guarantees the satisfaction of Req. I and comprises an adaptation pattern P_I and function F_I. The pattern triggers adaptation on the $before$ so lights are switched on, *during* event so if a section is empty the lights are switched off or during a match when a ticket holder enters s meaning that s is no longer empty,

$$\begin{aligned}
&P_I = \text{let}\\
&B(l, emp, ev) = \Box_{\substack{a \in adj\\ v \in Visitors}} \begin{cases}
before & \rightarrow \star\langle l, F\rangle \rightarrow ack \rightarrow B(l, T, before)\\
during & \rightarrow \star\langle l, emp\rangle \rightarrow ack \rightarrow B(l, emp, during)\\
after & \rightarrow\!\rightarrow B(l, emp, after)\\
floodlights\langle b\rangle & \rightarrow B(b, emp, ev)\\
goto_{v,a,s} & \rightarrow \text{if } ev \neq before \wedge emp \text{ then } \star\langle n, F\rangle \rightarrow ack \rightarrow B(l, F, ev) \text{ else } B(l, F, ev)\\
goto_{v,s,a} & \rightarrow B(l, emp, ev)\\
open_{s,s'} & \rightarrow B(l, emp, ev)\\
open_{s',s} & \rightarrow \star\langle l, F\rangle \rightarrow ack \rightarrow B(l, F, ev)\\
empty & \rightarrow \text{if } ev = after \text{ then } \star\langle l, T\rangle \rightarrow ack \rightarrow B(l, emp, ev) \text{ else } B(l, emp, ev)
\end{cases}\\
&\text{within } B(F, T)
\end{aligned}$$

The adaptation function encoded in the ACSP process F_I receives the status of the floodlight and the section – whether the floodlights are on/off and whether the section is empty. From this information, the function decides to switch the lights on or off or leaves them in their current status (the $STOP$ process)

$$\begin{aligned}
F_I = \star\langle l, emp\rangle \rightarrow &\text{if } emp \wedge l \text{ then } lp!floodlights\langle F\rangle \rightarrow STOP \rightarrow ack \rightarrow F_I\\
&\text{else if } \neg emp \wedge \neg l \text{ then } lp!floodlights\langle T\rangle \rightarrow STOP \rightarrow ack \rightarrow F_I\\
&\text{else } lp!STOP \rightarrow ack \rightarrow F_I
\end{aligned}$$

The adaptation procedure is defined as the composition of processes F_I and P_I

$$\Pi_I = (\nu \star, ack)\left(F_I \underset{\{\!|\star, ack|\!\}}{\parallel} P_I\right)$$

The *unary cluster* for a section s that is verified is

$$S_I = (\nu\, lp, ac, ms)\left((LightPanel \;|||\; AccessController \;|||\; MatchStatus) \underset{E_{ac}, E_l, E_m}{\parallel} \Pi_I\right)$$

We now discuss the specification for Req. I. We verify that the lights are switched on after the *before* event and switched off if the section remains empty by the time the match starts. The lights are switched on if during a match the section is no longer empty, either if a ticket holders enters the section or the section is used as a backup.

$$
\begin{aligned}
&Spec_I = \text{let} \\
&\qquad B(l, emp, ev) = \underset{\substack{a \in adj \\ v \in Visitors}}{\square} \begin{cases}
before & \rightarrow B(l, T, before) \\
during & \rightarrow B(l, emp, during) \\
after & \rightarrow B(l, emp, after) \\
\neg emp \wedge \neg l \ \& \ floodlights\langle \neg l \rangle & \rightarrow B(\neg l, emp, ev) \\
\neg l \wedge e = before \ \& \ floodlights\langle T \rangle & \rightarrow B(T, emp, ev) \\
emp \wedge l \wedge ev \neq before \ \& \ floodlights\langle F \rangle & \rightarrow B(F, emp, ev) \\
goto_{v,a,s} & \rightarrow B(l, F, ev) \\
goto_{v,s,a} & \rightarrow B(l, emp, ev) \\
open_{s',s} & \rightarrow B(l, F, ev) \\
open_{s,s'} & \rightarrow B(l, emp, ev) \\
empty & \rightarrow B(l, emp \vee ev = after, ev)
\end{cases} \\
&\qquad \text{within } B(F, T)
\end{aligned}
$$

With our verification approach, we verify that for sections s, the process S_I trace refines $Spec_I$.

$$Spec_I \sqsubseteq_{T(ACSP)} S_I$$

Bibliography

[1] Croke park - stadium map. https://crokepark.ie/matchday/stadium-map.

[2] Self-adaptive systems artifacts and model problems. https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/.

[3] Bahareh Abolhasanzadeh and Saeed Jalili. Towards modeling and runtime verification of self-organizing systems. *Expert Systems with Applications*, 44(Supplement C):230 – 244, 2016.

[4] Yousef Abuseta. An investigation of the monitoring activity in self adaptive systems. *CoRR*, abs/1802.03667, 2018.

[5] Yousef Abuseta and Khaled Swesi. Design patterns for self adaptive systems engineering. *arXiv preprint arXiv:1508.01330*, 2015.

[6] Rasmus Adler, Ina Schaefer, Tobias Schuele, and Eric Vecchié. From model-based design to formal verification of adaptive embedded systems. In *Proceedings of the Formal Engineering Methods 9th International Conference on Formal Methods and Software Engineering*, ICFEM'07, pages 76–95, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. MDSE@R: Model-Driven Security Engineering at Runtime. pages 279–295. Springer, Berlin, Heidelberg, 2012.

[8] Jesper Andersson, Luciano Baresi, Nelly Bencomo, Rogério de Lemos, Alessandra Gorla, Paola Inverardi, and Thomas Vogel. *Software Engineering Processes for Self-Adaptive Systems*, pages 51–75. Springer, Berlin, Heidelberg, 2013.

[9] Sreram Balasubramaniyan, Seshadhri Srinivasan, Furio Buonopane, B. Subathra, Jri Vain, and Srini Ramaswamy. Design and verification of cyber-physical systems using truetime, evolutionary optimization and uppaal. *Microprocessors and Microsystems*, 42:37 – 48, 2016.

[10] Björn Bartels and Moritz Kleine. A CSP-based framework for the specification, verification, and implementation of adaptive systems. In *Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems - SEAMS '11*, page 158. ACM Press, 2011.

[11] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[12] Sergey Berezin, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, COMPOS'97, pages 81–102, London, UK, UK, 1998. Springer-Verlag.

[13] Marco Bernardo, Rocco De Nicola, and Michele Loreti. Revisiting Trace and Testing Equivalences for Nondeterministic and Probabilistic Processes. *Logical Methods in Computer Science*, 10(1), mar 2014.

[14] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, March 1991.

[15] Viviana Bono, Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Data-driven adaptation for smart sessions. *Journal of Logical and Algebraic Methods in Programming*, 90(Supplement C):31 – 49, 2017.

[16] Aimee Borda and Vasileios Koutavas. Self-adaptive automata. In *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, FormaliSE '18, pages 64–73, New York, NY, USA, 2018. ACM.

[17] Aimee Borda, Liliana Pasquale, Vasileios Koutavas, and Bashar Nuseibeh. Compositional verification of self-adaptive cyber-physical systems. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '18, pages 1–11, New York, NY, USA, 2018. ACM.

[18] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel. Morph: A Reference Architecture for Configuration and Behaviour Self-adaptation. In *Proc. 1st International Workshop on Control Theory for Software Engineering*, pages 9–16. ACM, 2015.

[19] Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Adaptable processes. In *Proceedings of the Joint 13th IFIP WG 6.1 and 30th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'11/FORTE'11, pages 90–105, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[21] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. *Engineering Self-Adaptive Systems through Feedback Loops*, pages 48–70. Springer, 2009.

[22] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Adaptable Transition Systems. pages 95–110. Springer Berlin Heidelberg, 2013.

[23] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. *Science of Computer Programming*, 99:75–94, 2015.

[24] Sven Burmester, Holger Giese, and Matthias Tichy. Model-driven development of reconfigurable mechatronic systems with mechatronicuml. In Uwe Aßmann, Mehmet Aksit, and Arend Rensink, editors, *Model Driven Architecture*, pages 47–61, Berlin, Heidelberg, 2005. Springer.

[25] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly. Engineering trustworthy self-adaptive software with dynamic assurance cases. *IEEE Transactions on Software Engineering*, 44(11):1039–1069, Nov 2018.

[26] Radu Calinescu, Simos Gerasimou, and Alec Banks. Self-adaptive software with decentralised control loops. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, pages 235–251, Berlin, Heidelberg, 2015. Springer.

[27] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, September 2012.

[28] Radu Calinescu, Shinji Kikuchi, and Kenneth Johnson. Compositional reverification of probabilistic safety properties for large-scale complex it systems. In Radu Calinescu and David Garlan, editors, *Large-Scale Complex IT Systems. Development, Operation and Management*, pages 303–329, Berlin, Heidelberg, 2012. Springer.

[29] Javier Cámara and Rogério De Lemos. Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In *SEAMS*, pages 53–62. IEEE, June 2012.

[30] Matteo Camilli, Angelo Gargantini, and Patrizia Scandurra. Specifying and verifying real-time self-adaptive systems. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 303–313. IEEE, November 2015.

[31] Nicolas Cardozo, Sebastian Gonzalez, Kim Mens, Ragnhild Van Der Straeten, and Theo DHondt. Modeling and Analyzing Self-Adaptive Systems with Context Petri Nets. In *2013 International Symposium on Theoretical Aspects of Software Engineering*, pages 191–198. IEEE, jul 2013.

[32] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pages 1–26. Springer, Berlin, Heidelberg, 2009.

[33] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, pages 468–483, Berlin, Heidelberg, 2009. Springer.

[34] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, June 1989.

[35] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, 1993.

[36] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31:1–6, 2006.

[37] Rodolfo Conde and Sergio Rajsbaum. An introduction to the topological theory of distributed computing with safe-consensus. *Electronic Notes in Theoretical Computer Science*, 283:29 – 51, 2012. Proceedings of the workshop on Geometric and Topological Methods in Computer Science (GETCO).

[38] Maxime Cordy, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model Checking Adaptive Software with Featured Transition Systems. pages 1–29. Springer Berlin Heidelberg, 2013.

[39] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. Aiocj: A choreographic framework for safe adaptive distributed applications. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, pages 161–170, Cham, 2014. Springer International Publishing.

[40] Alexandre David, Kim. G. Larsen, Axel Legay, Mikael H. Møller, Ulrik Nyman, Anders P. Ravn, Arne Skou, and Andrzej Wasowski. Compositional verification of real-time systems using ecdar. *International Journal on Software Tools for Technology Transfer*, 14(6):703–720, Nov 2012.

[41] C. Daws. Symbolic and Parametric Model Checking of Discrete-Time Markov Chains. In *Proc. 1st International Colloquium on Theoretical Aspects of Computing*, volume 3407, pages 280–294. Springer, 2004.

[42] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*, pages 1–32. Springer, Berlin, Heidelberg, 2013.

[43] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.

[44] Rocco De Nicola and Matthew Hennessy. Testing equivalence for processes. In *Automata, Languages and Programming, 10th Colloquium, Barcelona, Spain, July 18-22, 1983, Proceedings*, pages 548–560, 1983.

[45] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. *Safety, Liveness and Run-Time Refinement for Modular Process-Aware Information Systems with Dynamic Sub Processes*, pages 143–160. Springer International Publishing, Cham, 2015.

[46] Giovanna Di Marzo Serugendo, Marie-Pierre Gleizes, and Anthony Karageorgos. Self-organization in multi-agent systems. *Knowl. Eng. Rev.*, 20(2):165–189, 2005.

[47] Zuohua Ding, Yuan Zhou, and Mengchu Zhou. Modeling Self-Adaptive Software Systems With Learning Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 46(4):483–498, April 2016.

[48] Nicolas D'Ippolito, Víctor Braberman, Jeff Kramer, Jeff Magee, Daniel Sykes, and Sebastian Uchitel. Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In *Proc. of the 36th International Conference on Software Engineering - ICSE 2014*, pages 688–699. ACM Press, 2014.

[49] Igor Brasileiro Duarte. CSP-GRAMMAR. https://github.com/igorbrasileiro/CSP-GRAMMAR, 2018.

[50] Hartmut Ehrig, Claudia Ermel, Olga Runge, Antonio Bucchiarone, and Patrizio Pelliccione. *Formal Analysis and Verification of Self-Healing Systems*, pages 139–153. Springer Berlin Heidelberg, 2010.

[51] Naeem Esfahani and Sam Malek. *Uncertainty in Self-Adaptive Software Systems*, pages 214–238. Springer, Berlin, Heidelberg, 2013.

[52] Yaser P. Fallah, ChingLing Huang, Raja Sengupta, and Hariharan Krishnan. Design of cooperative vehicle safety systems based on tight coupling of communication, computing and physical vehicle dynamics. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 159–167, New York, NY, USA, 2010. ACM.

[53] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time Efficient Probabilistic Model Checking. In *Proc. 33rd International Conference on Software Engineering*, pages 341–350, 2011.

[54] A. Filieri and G. Tamburrelli. Probabilistic Verification at Runtime for Self-Adaptive Systems. *Assurances for Self-Adaptive Systems*, 7740:30–59, 2013.

[55] Antonio Filieri, Giordano Tamburrelli, and Carlo Ghezzi. Supporting Self-Adaptation via Quantitative Verification and Sensitivity Analysis at Run Time. *IEEE Transactions on Software Engineering*, 42(1):75–99, jan 2016.

[56] Michael Fisher, Louise Dennis, and Matt Webster. Verifying autonomous systems. *Commun. ACM*, 56(9):84–93, September 2013.

[57] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004.

[58] Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, Adam Hujecek, Frantisek Plasil, and Dominik Skoda. Strengthening adaptation in cyber-physical systems via meta-adaptation strategies. *ACM Trans. Cyber-Phys. Syst.*, 1(3):13:1–13:25, April 2017.

[59] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika brahm and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[60] Heather J. Goldsby and Betty H. C. Cheng. Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty. In *Model Driven Engineering Languages and Systems*, pages 568–583. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[61] Heather J. Goldsby, Betty H. C. Cheng, and Ji Zhang. Amoeba-rt: Run-time verification of adaptive software. In Holger Giese, editor, *Models in Software Engineering*, pages 212–224, Berlin, Heidelberg, 2008. Springer.

[62] Thomas Göthel and Björn Bartels. *Modular Design and Verification of Distributed Adaptive Real-Time Systems*, pages 3–12. Springer International Publishing, Cham, 2015.

[63] Thomas Göthel, Nils Jähnig, and Simon Seif. *Refinement-Based Modelling and Verification ofDesign Patterns forSelf-adaptive Systems*, pages 157–173. Springer International Publishing, Cham, 2017.

[64] M. Hachicha, R. B. Halima, and A. H. Kacem. Modeling and verifying self-adaptive systems: A refinement approach. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 003967–003972, Oct 2016.

[65] M. Hachicha, R. B. Halima, and A. H. Kacem. Formalizing compound mape patterns for decentralized control in self-adaptive systems. In *2018 12th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–10, May 2018.

[66] Marwa Hachicha, Riadh Ben Halima, and Ahmed Hadj Kacem. Designing compound mape patterns forself-adaptive systems. In Ajith Abraham, Pranab Kr. Muhuri, Azah Kamilah Muda, and Niketa Gandhi, editors, *Intelligent Systems Design and Applications*, pages 92–101, Cham, 2018. Springer International Publishing.

[67] Allen. Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[68] Regina Hebig, Holger Giese, and Basil Becker. Making control loops explicit when architecting self-adaptive systems. *Proceedings of the second international workshop on Self-organizing architectures*, pages 21–28, 2010.

[69] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[70] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[71] M. Usman Iftikhar and Danny Weyns. Towards runtime statistical model checking for self-adaptive systems, 2016.

[72] Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Towards an assume-guarantee theory for adaptable systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*, pages 106–115, 2009.

[73] Yoshinao Isobe and Markus Roggenbach. Csp-provera proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1):32–39, 2010.

[74] Kenneth Johnson, Radu Calinescu, and Shinji Kikuchi. An incremental verification framework for component-based software systems. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 33–42, New York, NY, USA, 2013. ACM.

[75] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[76] Yonit Kesten and Amir Pnueli. A compositional approach to ctl* verification. *Theoretical Computer Science*, 331(2):397 – 428, 2005. Formal Methods for Components and Objects.

[77] Narges Khakpour, Saeed Jalili, Carolyn Talcott, Marjan Sirjani, and MohammadReza Mousavi. Formal modeling of evolving self-adaptive systems. *Science of Computer Programming*, 78(1):3–26, November 2012.

[78] Narges Khakpour, Marjan Sirjani, and Ursula Goltz. Context-based behavioral equivalence of components in self-adaptive systems. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, pages 16–32, Berlin, Heidelberg, 2011. Springer.

[79] Annabelle Klarl. Engineering Self-Adaptive Systems with the Role-Based Architecture of Helena. In *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 3–8. IEEE, June 2015.

[80] Vasileios Koutavas and Matthew Hennessy. First-order reasoning for higher-order concurrency. *Computer Languages, Systems & Structures*, 38(3):242 – 277, 2012.

[81] Vasileios Koutavas and Matthew Hennessy. Symbolic bisimulation for a higher-order distributed language with passivation. In Pedro R. D'Argenio and Hernán Melgratti, editors, *CONCUR 2013 – Concurrency Theory*, pages 167–181, Berlin, Heidelberg, 2013. Springer.

[82] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17:184 – 206, 2015. 10 years of Pervasive Computing In Honor of Chatschik Bisdikian.

[83] W. H. Lim, N. A. M. Isa, S. S. Tiang, T. H. Tan, E. Natarajan, C. H. Wong, and J. R. Tang. A self-adaptive topologically connected-based particle swarm optimization. *IEEE Access*, pages 1–1, 2018.

[84] Malte Lochau, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. Deltaccs: A core calculus for behavioral change. In *Part I of the Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - Volume 8802*, pages 320–335, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[85] Iftikhar M. Usman and Weyns Danny. A case study on formal verification of self-adaptive behaviors in a decentralized system. *Electronic Proceedings in Theoretical Computer Science*, (Proc. FOCLASA 2012):45, 2012.

[86] Frank D Macías-Escrivá, Rodolfo Haber, Raul del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18):7267–7279, December 2013.

[87] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.

[88] Emanuela Merelli, Nicola Paoletti, and Luca Tesei. Adaptability checking in complex systems. *Science of Computer Programming*, 115:23–46, 2016.

[89] R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[90] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[91] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[92] Niall Moran. Croke Park: Sound and weather data monitoring within a smart stadium — Microsoft Technical Case Studies. https://microsoft.github.io/techcasestudies/iot/2016/10/28/CrokePark.html, 2017.

[93] G. A. Moreno, J. Cmara, D. Garlan, and B. Schmerl. Efficient decision-making under uncertainty for proactive self-adaptation. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 147–156, July 2016.

[94] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 1–12, New York, New York, USA, 2015. ACM Press.

[95] J. Morse, D. Araiza-Illan, K. Eder, J. Lawry, and A. Richards. A fuzzy approach to qualification in design exploration for autonomous robots and systems. In *2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–6, July 2017.

[96] V. Nallur and R. Bahsoon. A decentralized self-adaptation mechanism for service-based applications in the cloud. *IEEE Transactions on Software Engineering*, 39(5):591–612, May 2013.

[97] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002.

[98] Pierluigi Nuzzo, Jiwei Li, Alberto L. Sangiovanni-Vincentelli, Yugeng Xi, and Dewei Li. Stochastic assume-guarantee contracts for cyber-physical system design. *ACM Trans. Embed. Comput. Syst.*, 18(1):2:1–2:26, 2019.

[99] Charles L. Ortiz, Régis Vincent, and Benoit Morisset. Task inference and distributed task management in the centibots robotic system. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, pages 860–867, New York, NY, USA, 2005. ACM.

[100] Rob Oshana. Chapter 1 - principles of parallel computing. In Rob Oshana, editor, *Multicore Software Development Techniques*, pages 1 – 30. Newnes, Oxford, 2016.

[101] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[102] L. Pasquale, C. Ghezzi, C. Menghi, C. Tsigkanos, and B. Nuseibeh. Topology Aware Adaptive Security. In *Proc. 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 43–48. ACM, 2014.

[103] Liliana Pasquale, Claudio Menghi, Mazeiar Salehie, Luca Cavallaro, Inah Omoronyia, and Bashar Nuseibeh. SecuriTAS: A Tool for Engineering Adaptive Security. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, page 1, New York, New York, USA, November 2012. ACM Press.

[104] . Piel, A. Gonzalez-Sanchez, H. Gross, and A. J. C. v. Gemund. Spectrum-based health monitoring for self-adaptive systems. In *2011 IEEE Fifth International Conference on Self-Adaptive and Self-Organizing Systems*, pages 99–108, Oct 2011.

[105] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[106] Roy S. Rubinstein and John N. Shutt. Self-modifying finite automata: An introduction. *Information Processing Letters*, 56(4):185–190, November 1995.

[107] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):1–42, May 2009.

[108] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[109] Davide Sangiorgi. From π-calculus to higher-order π-calculus — and back. In M. C. Gaudel and J. P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, pages 151–166, Berlin, Heidelberg, 1993. Springer.

[110] Bryan Scattergood. *The semantics and implementation of machine-readable CSP*. PhD thesis, Citeseer, 1998.

[111] Bryan Scattergood and Philip Armstrong. Cspm: A reference manual, 2011.

[112] I. Schaefer and A. Poetzsch-Heffter. Compositional reasoning in model-based verification of adaptive embedded systems. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 95–104, Nov 2008.

[113] A. Schmitt and J.-B. Stefani. The kell calculus: A family of higher-order distributed process calculi. In C. Priami and P. Quaglia, editors, *Global Computing, IST/FET International Workshop, GC 2004, Rovereto, Italy, March 9-12, 2004, Revised Selected Papers*, volume 3267 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2004.

[114] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[115] Andreas Schroeder, Sebastian S. Bauer, and Martin Wirsing. A contract-based approach to adaptivity. *Journal of Logic and Algebraic Programming*, 80(3-5):180–193, apr 2011.

[116] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar. Opportunities and obligations for physical computing systems. *Computer*, 38(11):23–31, Nov 2005.

[117] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[118] Daniel Sykes, Jeff Magee, and Jeff Kramer. Flashmob: Distributed adaptive self-assembly. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 100–109, New York, NY, USA, 2011. ACM.

[119] Bent Thomsen. Plain chocs a second generation calculus for higher order processes. *Acta Informatica*, 30:1–59, 1993.

[120] Christos Tsigkanos, Liliana Pasquale, Carlo Ghezzi, and Bashar Nuseibeh. On the Interplay Between Cyber and Physical Spaces for Adaptive Security. *IEEE Transactions on Dependable and Secure Computing*, PP(99), 2017.

[121] Christos Tsigkanos, Liliana Pasquale, Claudio Menghi, Carlo Ghezzi, and Bashar Nuseibeh. Engineering topology aware adaptive security: Preventing requirements violations at runtime. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pages 203–212. IEEE, August 2014.

[122] W. M. P. van der Aalst, A. K. Alves de Medeiros, and A. J. M. M. Weijters. Process equivalence: Comparing two process models based on observed behavior. In *Proceedings of the 4th International Conference on Business Process Management*, BPM'06, pages 129–144, Berlin, Heidelberg, 2006. Springer-Verlag.

[123] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On interacting control loops in self-adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 202–207, New York, NY, USA, 2011. ACM.

[124] Jun Wang and Gregory M. Provan. A comparative analysis of specific spatial network topological models. In *Complex*, 2009.

[125] Danny Weyns and M. Usman Iftikhar. Model-Based Simulation at Runtime for Self-Adaptive Systems. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 364–373. IEEE, jul 2016.

[126] Danny Weyns, Sam Malek, and Jesper Andersson. On decentralized self-adaptation: Lessons from the trenches and challenges for the future. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 84–93, New York, NY, USA, 2010. ACM.

[127] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. *On Patterns for Decentralized Control in Self-Adaptive Systems*, pages 76–107. Springer, Berlin, Heidelberg, 2013.

[128] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. In *Proceedings of the 19th National Conference on Innovative Applications of Artificial Intelligence - Volume 2*, IAAI'07, pages 1752–1759. AAAI Press, 2007.

[129] J. Wuttke, Y. Brun, A. Gorla, and J. Ramaswamy. Traffic routing for evaluating self-adaptation. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 27–32, June 2012.

[130] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 371. ACM Press, 2006.

[131] Y. Zhao, S. Oberthür, M. Kardos, and F.J. Rammig. Model-based Runtime Verification Framework for Self-optimizing Systems. *Electronic Notes in Theoretical Computer Science*, 144(4):125–145, May 2006.

[132] Yongwang Zhao, Dianfu Ma, Jing Li, and Zhuqing Li. Model Checking of Adaptive Programs with Mode-extended Linear Temporal Logic. In *2011 Eighth IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 40–48. IEEE, April 2011.